

PARALLEL COMPUTING TECHNOLOGIES

Proceedings of the International Conference

PARALLEL COMPUTING TECHNOLOGIES



Novosibirsk, USSR September 7 - 11, 1991



Editor

N N Mirenkov

Computing Center, Novosibirsk, USSR



World Scientific

Singapore • New Jersey • London • Hong Kong

Published by

World Scientific Publishing Co. Pte. Ltd.

P O Box 128, Farrer Road, Singapore 9128

USA office: Suite 1B, 1060 Main Street, River Edge, NJ 07661

UK office: 73 Lynton Mead, Totteridge, London N20 8DH

Library of Congress Cataloging-in-Publication data is available.

**PARALLEL COMPUTING TECHNOLOGIES —
Proceedings of the International Conference**

Copyright © 1991 by World Scientific Publishing Co. Pte. Ltd.

All rights reserved. This book, or parts thereof, may not be reproduced in any form or by any means, electronic or mechanical, including photocopying, recording or any information storage and retrieval system now known or to be invented, without written permission from the Publisher.

ISBN 981-02-0698-4

Printed in Singapore by JBW Printers and Binders Pte. Ltd.

PREFACE

We are currently entering an era of *developed parallelism*. Parallelism provides the high performance needed in science and engineering, the responsiveness required in real-time control, the fault-tolerance necessary for high reliability systems, etc. These capabilities become available for use in an increasing number of applications but offer new challenges to the system designer and programmer.

The advanced research in parallel computing technologies as well as the development of high industrial quality products are now especially important for the USSR trying to do perestroika's steps to be tightly integrated with the world community. We believe that increasing research contacts and scientific cooperation is one of the first steps to take. So, in spite of the considerable number of international conferences and workshops on parallel systems, we decided to establish a Soviet series on parallel computing technologies (the PaCT series). As a result you have in your hands the proceedings of the PaCT-91 Conference, which is organized by the Computing Centre of the Siberian Division of the USSR Academy of Sciences in cooperation with the Soviet Parallel Computing Association, and held in Novosibirsk in September 1991. This is the first time such a big international conference on parallel computing technologies is taking place in the USSR. The aim of the conference is to give an overview of new developments, applications and trends in the field of parallelism.

The program committee has selected 45 presentations from over 85 submissions from 11 countries. Unfortunately, the final versions of some selected papers have not reached us in time to be included in the proceedings.

The organizers would like to express their special thanks to the program committee members and to other reviewers who have given their very valuable time in selecting and reviewing the papers.

Novosibirsk, June 1991

Nikolay N. Mirenkov

COMMITTEES

ORGANIZING COMMITTEE

A. Alekseev	- Academy Novosibirsk (USSR) General Chairman
V. P. Bhatkar	- CDAC, Pune (IND)
Ch. Boutzev	- INCCA, UNESCO, Paris (FR)
N. Duplensky	- UNESCO, Moscow (USSR)
St. Markov	- Academy Sofia (BUL)
V. Melnikov	- Academy Moscow (USSR)
N. Mirenkov	- Academy Novosibirsk (USSR) Vice-Chairman
St. Sedukhin	- Academy Novosibirsk (USSR) Scientific Secretary
D. Wilson	- M.C.U., London (UK)

PROGRAM COMMITTEE

N. N. Mirenkov	- Academy Novosibirsk (USSR) Chairman
O. L. Bandman	- Academy Novosibirsk (USSR)
V. P. Bhatkar	- CDAC, Pune (IND)
Ch. Boutzev	- INCCA, UNESCO, Paris (FR)
H. Burkhart	- Univ. Basel (CH)
W. W. Chu	- UCLA, Los Angeles (USA)
M. Clementi	- IBM, Kingston (USA)
M. Cosnard	- LIP, Lyon (FR)
P. Degano	- Univ. Pisa (It)
W. Handler	- Univ. Erlangen (FRG)
W. P. Ivannikov	- Academy Moscow (USSR)
Ch. Jesshope	- Univ. Surrey (UK)
G. Joubert	- Philips, Eindhoven (NL)
V. E. Kotov	- Academy Novosibirsk (USSR)
H. T. Kung	- Carnegie-Mellon Univ. (USA)
V. P. Kutepov	- Energy Institute, Moscow (USSR)
Ch. Lengauer	- Univ. Edinburgh (UK)
V. K. Levin	- Academy Moscow (USSR)
St. Markov	- Academy Sofia (BUL)
B. A. Melnikov	- Academy Moscow (USSR)
R. H. Perrott	- Univ. Belfast (UK)
P. Quinton	- INRIA, Rocquencourt (FR)

B. Rebel	– ZKI, Berlin (FRG)
K. D. Reinartz	– Univ. Erlangen (FRG)
A. A. Samarsky	– Academy Moscow (USSR)
M. Vaitersic	– Academy Bratislava (CSFR)
D. R. Wilson	– M. C. U., London (UK)
G. Wolf	– ZKI, Berlin (FRG)

**LOCAL ORGANIZING
COMMITTEE**

S. G. Sedukhin	– Academy Novosibirsk (USSR) Chairman
A. P. Vazhenin	– Academy Novosibirsk (USSR) Technical Secretary
S. M. Achasova	– Academy Novosibirsk (USSR)
Y. I. Kolosova	– Academy Novosibirsk (USSR)
V. E. Malyshkin	– Academy Novosibirsk (USSR)
V. P. Markova	– Academy Novosibirsk (USSR)
O. G. Monakhov	– Academy Novosibirsk (USSR)

ACKNOWLEDGEMENTS

For substantial help we are obliged to the following referees:

D. K. Arvind (CS)	A. V. Larin (USSR)
O. L. Bandman (USSR)	A. L. Lastovetsky (USSR)
R. M. Chamberlain (UK)	Ch. Lengauer (UK)
A. P. Cherniaev (USSR)	V. P. Markova (USSR)
I. S. Duff (UK)	N. N. Mirenkov (USSR)
R. Degano (It)	F. F. Martens (FRG)
G. S. Elisarov (USSR)	M. van der Meer (FRG)
Ya. Fet (USSR)	C. Morgan (UK)
P. J. Fleming (UK)	J. M. Muller (FR)
S. J. Flavell (UK)	P. Quinton (FR)
V. N. Fal'k (USSR)	P. Rajzsní (CSFR)
A. I. Galushkin (USSR)	K. D. Reinartz (FRG)
R. Gorrieri (It)	L. Ricci (It)
S. S. Gajarian (USSR)	F. Romani (It)
W. Handler (FRG)	S. A. Simonov (USSR)
S. V. Jablonsky (USSR)	L. A. Sedukhina (USSR)
Xue Jingling (CS)	M. C. Schwem (FRG)
I. E. Kaporin (USSR)	P. Shedko (USSR)
Yu. T. Katalov (USSR)	P. Thanish (CS)
V. V. Korneev (USSR)	M. Vajtersic (CSFR)
V. E. Kotov (USSR)	I. B. Virbitskaite (USSR)
A. B. Kucherov (USSR)	M. Yu. Vasil'yev (USSR)
V. Kutepov (USSR)	G. Wolf (FRG)
M. Lucka (CSFR)	V. V. Zolotarev (USSR)

CONTENTS

Introduction

Nature Needed Billions of Years...	1
<i>W. Händler</i>	

Keynote Remarks	17
<i>V. A. Melnikov</i>	

Invited Papers

Parallel Programming Language Developments	20
<i>R. H. Perrott</i>	

A View of Systolic Design	32
<i>Ch. Lengauer</i>	

High Performance Computer System "Electronica SSBIS"	47
<i>V. A. Melnikov, Ju. I. Mitropolski, V. Z. Shnitman, V. P. Ivannikov, A. N. Tomilin and S. S. Gajsarian</i>	

Vertical Processing in Parallel Computing Systems	56
<i>W. Händler and Ya. I. Fet</i>	

Methods of Parallel Information Processing in the Architecture of Soviet High-Performance Computers	76
<i>B. A. Babajan</i>	

Influence of Communications on the Complexity of Givens Factorization on a Network of Processors	82
<i>M. Cosnard, E. M. Daoudi and B. Tourancheau</i>	

Systolic Systems

Massively Parallel Systolic Processors for High-Speed Recognition of Simple Patterns	98
<i>R. Männer, J. GläB and F. Klefenz</i>	

On the Design of Systolic Array Architectures for the Iterative Solution of Linear Systems <i>R. Wyrzykowski</i>	109
A Systematic Approach to the Design of Processor Arrays <i>Ju. S. Kanevsky and D. V. Korchev</i>	119
Design of Systolic Arrays for Iterative Algorithms: Eigenvalue Computations <i>N. A. Likhoded, P. I. Sobolevskii and A. A. Tiountchik</i>	129
The Design of a Linear Systolic Array for the Gauss-Jordan Algorithm with Partial Pivoting <i>V. V. Kopyanchuk and V. P. Yakush</i>	139
Parallel Logic and Functional Programming	
Parallel Constraint Satisfaction in a Logic Programming Framework <i>M. Meyer</i>	148
Integrated Tools for Functional, Logical and Data Flow Parallel Programming and Controlling Parallel Computations on Computer Systems <i>V. Kutepov and V. Falk</i>	158
Parallelized Quicksort and Radixsort with Optimal Speedup <i>D. M. W. Powers</i>	167
Knowledge-Based Parallel Scheduling Strategies in Parallel Logic Programming Systems <i>Gao Yaoqing, Wang Dingxing, Qiu Xiaolin, Shen Meiming, Zheng Weimin, Wen Dongchan and Hu Shouren</i>	177
Very-High-Level Language PARIS <i>V. D. Korneev, N. N. Mirenkov and A. Sh. Nepomniaschaya</i>	186
Distributed-Memory Programs Development in Declarative & Procedural Environment <i>S. P. Gorlatch and G. N. Gorlatch</i>	195

Massively Parallel Processing

Recognizing Faces with a Massively Parallel System <i>P. Flocchini, F. Gardin, G. Mauri, M. P. Pensini and P. Stofella</i>	203
The Use of a PRAM Architecture for Parallel Image Reconstruction <i>W. L. Nowinski</i>	215
A Comparison of Interconnection Networks for Large SIMD Parallel Computers <i>M. Schwehm and A. Strey</i>	223
Space-Time Transformation of Cellular Computations <i>O. L. Bandman</i>	235
High-Performance Computing Systems of Combined Architecture <i>A. P. Vazhenin, S. G. Sedukhin and Ya. I. Fet</i>	246
Language STAR for Associative and Parallel Computation with Vertical Data Processing <i>A. Sh. Nepomniaschaya</i>	258
 Parallel Programming and Tools	
The Kernel of a Graphic Environment for Analyzing Distributed Systems <i>P. Degano, R. Gorrieri, L. Zamboni and P. Zanotti</i>	266
A Design and Specification Environment for Distributed Memory Multiprocessors <i>T. Bemmerl, T. Ludwig and B. Ries</i>	280
Multiple Function Multiple Data Procedures and Parallel Programming <i>N. N. Mirenkov and L. A. Bulysheva</i>	292
A Tool to Display Hierarchical Acyclic Dataflow Graphs <i>S. Mitrovic and St. Murer</i>	304

Assemble Parallel Programming System INYA <i>V. A. Anisimov and V. E. Malyshkin</i>	316
Parallel Algorithms for Graph Topology Analysis and their Implementation for a Vector Supercomputer <i>A. M. Wabishchevich, N. S. Kovalenko, M. I. Ovseets and L. I. Shevchenko</i>	325
A Programming Methodology for Effective Data Exchanges in Macroconveyer Programs <i>A. E. Doroshenko</i>	330
Linearized Mass Computation <i>V. E. Malyshkin</i>	339
Applications, Algorithms and Systems	
Kinetical-Consistent Finite-Difference Gasdynamic Schemes and their Solution on Multiprocessor Computers <i>T. G. Elizarova, A. E. Dujsekulov and L. V. Kosarev</i>	354
Construction of Parallel Programs for Multiprocessors in Computational Physics Problems <i>D. P. Kostomarov, A. V. Pedorenko and A. M. Popov</i>	363
Algorithm Based Fault-Tolerant Technique for Matrix Inversion <i>M. K. Stojcev, E. I. Milovanovic and I. Z. Milovanovic</i>	375
Software and Hardware for Simulation of Physical-Mechanical Fields <i>V. L. Rvachev, A. N. Shevchenko, A. I. Koshelenko, I. I. Levin and S. Yu. Fomin</i>	385
Simulation, Evaluation and Analysis	
Performance Evaluation of Parallel SIMD-Systems by Interpreting Automatically Transformed Algorithms on Simulated Architectures <i>M. M. Gutzmann</i>	395

The Combined Approach to the Distributed Computer System Simulation	407
<i>R. L. Smeliansky, Y. P. Kazakov and Y. V. Bakalov</i>	
Synthesis of Models for Multiprocessor System Evaluation	418
<i>V. Khabas</i>	
Highly Efficient Algorithms for Static Analysis of Parallel Programs	428
<i>N. N. Mirenkov and S. A. Simonov</i>	
A Petri Net Entity as a Formal Model for LOTOS, a Specification Language for Distributed and Concurrent Systems	440
<i>N. A. Anisimov</i>	
Optimal Circulant Computer Networks	450
<i>E. A. Monakhova</i>	
Resource Controlling and Tools	
Parallel Isotropic Access to Tree-Like Data Structures	459
<i>R. Creutzburg and L. Andrews</i>	
Parallel Algorithm for Mapping of Program Graphs into Parallel Computers	471
<i>O. G. Monakhov</i>	
Dynamic Parallelism in Transputer Systems	477
<i>A. V. Anisimov and Yu. E. Boreisha</i>	
HINTERCUBE: The Project of a New Generation Multiprocessor System	487
<i>S. A. Rakov, A. A. Shvein and V. A. Signaevskii</i>	
Parallel Programming Technology and Resource Management in Multiprocessor Systems	495
<i>L. K. Babenko, O. B. Makarevich and A. G. Chefranov</i>	

Nature needed billions of Years

Wolfgang Händler

Friedrich-Alexander-Universität
Erlangen-Nürnberg
Institut für Mathematische Maschinen und
Datenverarbeitung (Informatik)
Martensstrasse 3, D-8520 Erlangen, Germany

1. Nature as a pattern: Evolution

We describe best the evolution of our world by talking about the "Big Bang" and get then a certain imagination of the cosmos. We talk about "Origin of life" and have a certain feeling regarding living beings. If we follow DARWIN we explain the evolution of life by some procedure of Trial and Error. We recognize a line, which begins with simple unicellular organisms or monads, continues with simple microorganisms with a multitude of cells and proceeds to more complex living entities, ending for the present with mammals and in particular with man. It consumed some billions of years.

The fascinating emergence of digital computers 50 years ago offers us the opportunity to compare carefully these new man-made structures with structures of organisms. That does not necessarily mean that we identify computers with Creation in the philosophical or religious sense. But there are some obvious considerations which might be helpful to illustrate today's "state of the art" with respect to computers.

Some decades ago we began with von-Neumann-Computers which were an answer to a question "How simple can be a computer structure?" I think it is not unrealistic to compare this minimal type with the unicellular organisms mentioned earlier. Since a couple of years we have been under the way to exploit all kinds of compositions of elements of the von-Neumann-structure - Arithmetic and Logic Units (or ALUs) on the one hand and whole Processors on the other hand. The shape of the resulting entities then become similar to multicellular organisms which are so typical for the creatures in nature. There is always a replication of elements and there is always a certain redundancy. This redundancy gives the living organisms the opportunity to survive in the case, where one cell - or even more - are destroyed. The same chance of substituting faulty elements in runtime - either ALUs or Processors - can be achieved now particularly in multiprocessors.

It will nevertheless consume some decades more to achieve the same in computers what nature exemplifies to an admirable extent - self-healing forces for instance. Evidently there are many things more to exploit carefully. So it will be always rewarding to consider living entities while dealing with computer architecture, computer programming, operating systems etc.

On the other side it seems to be not impossible that physiologists come to new insight if they study computer structures and architecture. A colleague of mine, W. D. Keidel, and me we studied with a small team structures and behaviour of computer structures and organisms. It is not quite easy to come to conclusions in this borderland. Nevertheless we could derive some more insight from the investigations. The results were interesting for both areas - the physiology and the computer science. Let me say distinctly that we felt that always the difference was quite important, while the similarities e.g. led us to computer structures which were structured on the one hand according to a retina-like-pattern with nearest-neighbour arrays and on the other hand according to the hierarchical control structure of our brain. We could verify that these basic forms - combined in a common structure - which is known as EGPA-Pyramid - has an advantageous and unique capability to run a broad spectrum of applications, including image-processing, matrix computations and finally non-regularly structured problems. We will come back to this structure called EGPA or MEMSY, reflecting a certain level of computer evolution.

The two basic structures we just mentioned - the nearest-neighbour topology on the one hand and the hierarchical steplike structure on the other hand are feasible concepts with respect to powerful multiprocessors. The proximity to organic structures and behaviorism seems to be evident.

.

2. History of parallel processing - the better structure will survive

Already Ch. BABBAGE made a statement that it would be possible to compute a set of values for a - say - nautic table - at the same time. Computing n values from n arguments would mean that n computing devices have to run at the same time, which evidently amounts to n -fold the price of one. Nevertheless we know that Babbage failed to build up one - as he called it - "analytical machine" and make it work. So he failed evidently to make it in parallel. Apparently the parallelism in view was a very primitive one, which even is not applicable as in the mentioned nautic table for instance the value i is dependent on the value $i - 1$, which is sometimes the most efficient way to compute tables.

BABBAGE's statement seems to be quite evident and it does not demand mentioning at all. A very similar idea is the practice of astronomical institutions at the end of the 19th century. Astronomers distributed at that time huge computations to friendly institutions on the continent. These institutions then executed the computation (or say: program) by their staff utilizing a dozen desk calculators or more. The results were finally composed by the institution which originally distributed the material. Since the capacity in each of the institutions were restricted with respect to staff and the number of desk calculators, this practice led to a shorter execution time - one of the first examples of real parallel processing.

K. ZUSE finally proposed in 1956 a structure which he called the "Feldrechenmaschine" (Array-Computer) where many identical computations are executed at the same time - we sometimes characterize this type of computer as SIMD (single-instructions-multi-data). About 1970 K. ZUSE extended his ideas to what he is calling the "Rechnender Raum" (Computing Universe). Also this structure bases on a great number of equal elements. In this case these elements were allocated in a discrete k-dimensional space like particles in a crystal.

As you probably know SLOTNICK came up with an array computer in 1962 the so-called SOLOMON-Computer. It was the idea of an array-computer the elements of which were allocated in a 32-times-32-array of one-bit ALUs mainly to solve problems shaped like membrans or vaults. As far as is known this SOLOMON remained in a paper state. Some years later SLOTNICK was given an opportunity to realize his ideas in a computer called ILLIAC IV. While SOLOMON was based on a one-bit ALU-array, ILLIAC IV had more powerful ALUs. Unfortunately these ALUs were only built up in a number of 64 (not as originally planned 256). The ALUs could be allocated linearly or in an 8-times-8-array. Admittedly the ALUs had each 64 bit wordlength. This huge computer was in operation till 1983. It brought a lot of experience for the type of S I M D - computation. The successful operation was possible since Burroughs-Corp. finished the design and made the computer ready for an extended research at NASA-Ames. Unfortunately Burroughs did not succeed to make an industrial product from its experience with ILLIAC IV. The promising Burroughs BSP (Burroughs Scientific Processor) never reached the market.

It was nearly at the same time when some multiprocessors came up, e.g. from Burroughs-B 5500 , IBM 360 / Modell 67 , General Electric GE 645 and some others. The M I M D -type Multiprocessors had at most four processors so these computers were later on called the "timids", since they only followed the concept of distributing the jobs (or the job stream) onto the processors as they arrived in the I/O-sections of these computers. The aim was seldomly to compute one huge problem on the multiprocessor in order to get a speed-up of k , if k is

approximately the number of processors ($k=4$). Indeed the result - mainly rated according to performance and efficiency - was mostly depressing. Sometimes the speed-up of a Four-Processor reached only about 2.3 (instead of 4) or even less. So it was understandable to decide to go back to a separate operation of the four processors. In this case we also could say: to separate four computers.

Nevertheless the depression in the early seventies had some other reasons too. On the one hand some more theoretical considerations came up, which cannot be fully discussed in this note. There was first MINSKY who estimated the portion of time where in a parallel device a full parallelism can take place is only about $\log^2 p$, if p is the number of processors (or processor elements). This would mean, that with 64 elements ILLIAC IV would come only to a speed-up of 6 instead of about 64 one would expect by common sense.

More seriously AMDAHL presented his "Law" (AMDAHL's Law), where he argued that there are always some parts of the computation, which are not parallelisable at all. Regarding a certain portion f which is characteristic for a specific program or problem (where f is the portion of non-parallelisable code) then it is possible numbering the speed-up. For instance if a tenth of the code is non-parallelisable, then $f = 0.1$, and the maximal speed-up is at the most $s = 10$, how large ever the number of processors (or processor elements) becomes. Similarly if only half of the program can be parallelised ($f = .5$) the speed-up will never exceed $s = 2$.

Nevertheless one has to investigate these Laws carefully. MINSKY's estimation - some-times called MINSKY's conjecture - proved out to be too pessimistic, so it was laid off from most experts in practice. Regarding AMDAHL's Law is a hard restriction mainly in the case of vector-processing. The percentage of code which can be vectorized is always much smaller than the percentage which can be parallelised. Beyond it endeavours to parallelise a code can always be balanced regarding the multiprocessor-structure on one hand and the size of the problem on the other hand in order to get a satisfying efficiency. This can be seen also as an argument to concentrate all our endeavours on parallelisation and not only on vectorization.

GROSCH's Law finally says that if we spend double the cost for the computer then we get four times the performance or speed. Similarly if we spend three times the cost then we get nine times the performance or speed (performance is the square of cost). Evidently the consequence would be then that parallelism is not a reasonable concept. Nevertheless this law apparently cannot be valid in the case of highly parallel devices. The increase is rather linear or even somewhat lower in practice. There are apparently many parameters and facts which determine performance and efficiency of parallel computers. To create more realistic models is the challenge of our days.

Good models mean that good predictions can be made for a new architecture. Good models also offer the hope that the mythoi which came up with the seventies, that parallel processing devices are too complex, consume too much time for communication, cannot be easily programmed etc. (compare A. JONES) may be forgotten.

In spite of these and other obstacles research was continuing during the seventies and eighties. At the National Bureau of Standards LEINER, NOTZ et. al. conceived the PILOT-Project (1972), a multiprocessor composed from three processors each of which taking over a specific function and working like a team. At the Carnegie-Mellon-University WULF, BELL et. al. combined 16 processors of PDP 11 via a crossbar-switch with the same number of memory blocks. The multiprocessor, called C.mmp (according to a notation PMS, which was created nearly at the same time at Carnegie-Mellon), was for a while a research vehicle for parallelisation.

Regarding the expensive crossbar-switch (cost is increasing with the square of connected elements) Carnegie-Mellon University later paid more attention to a cluster connection topology. The CM* computer (CM-Star) had a hierarchical three-level topology. In general one of the key problems of multiprocessor-architecture became now the connection topology. CM* was not a breakthrough regarding this problem. Nevertheless a lot of papers referred to this multiprocessor of Carnegie-Mellon. At least CM* exceeded at that time the magic threshold of 16. Up to 16 processors one can restrict the topology problem connecting all processors to one bus. The proper problem begins with the necessity of allocating more than one bus. CM* containing about 50 processors had several buses in a multi-level tree-like configuration.

After all, Carnegie-Mellon made some statements of the kind: What we learned from the multiprocessor CM*. At least some people of other institutions and universities considered CM* as a starting signal for own research.

It became evident that some physical facts would delimit the further increase of speed or performance in a single processor. This and a rather more realistic rating of "laws" we mentioned already, led to a widespread research in parallel processing, array processing and vector processing. Vector processing in particular became the most favoured area of some known computer manufacturers like Cray, Amdahl, Fujitsu, Hitachi, IBM, Siemens etc. They were encouraged to go into this field, since it seemed rather simple to extend the conventional habits of programming to the vector-type of supercomputing, as it then was called.

Computing Centres or research people running both types of parallel devices - vector-processors and multiprocessors made the experience that also vectorprocessors cause some problems in utilizing fully the pipelines, the more

as AMDAHL's Law sets just for this type unavoidable limits. The mentioned corporations were originally arguing that in designing and utilizing vectorprocessors it would not be necessary to go into the field of multiprocessing. Now they approach what they call today multitasking with their huge vectorprocessors. They are coming just from the other side. They combine 4 , 8 or 16 powerful vectorprocessors, while most of nowadays multiprocessors e.g. from ALLIANT, PARSYTEC, SEQUENT, n Cube, Intel Touchstone etc. are ensembles, arrays or clusters composed from 64 , 256, 1024 or even more rather parsimonious processors. Possible the two lines will grow together one day.

In this respect J. ROLLWAGEN, the President of Cray-Research Inc., said some months ago "Rather, massively parallel processing can, for some specialized applications, enhance general purpose computing. Thus we will pursue integration of the two architectures." The view is that both lines will grow together. The massively parallel concept will not merely substitute vector-multiprocessing - citing J. ROLLWAGEN. Probably not all computer architects will agree with this statement without some minor restrictions.

The line of Array processors like Feldrechenmaschine, SOLOMON and ILLIAC IV had a couple of successors which are still alive and work successfully on applications with regular structures.

STARAN as the very beginner in the line should be mentioned as also MPP and DAP for instance. Nevertheless at the time a consideration begins that the applicability of these SIMD-computers is dependant of the vectorizability of the application and that this is rather restrictive as compared with the MIMD-computers or multiprocessors. These latter can also compute all problems of the SIMD-type but an array processor cannot substitute a real MIMD-computer. In that way our multiprocessors are more general in their applicability. The simple necessary changeability of MIMD-structures comes from the fact that the multitude of (self-contained) processors can always be provided with a unique program. They then degrade in a way to a SIMD-structure. This explains the preference of a majority of computer-architects given to MIMD-type computers today.

"The better structure will survive" we stated at the beginning of the brief historical review. In technology a permanent selection process like in nature is going on. Some of today's parallel computers and multiprocessors will disappear from the production lines. Others which even are not to be seen now will prove out to be the more viable structures.

3. State of the art - in particular multiprocessors.

There is rather no doubt that most of our challenging future problems cannot be solved without any parallel approach. So we will have, at least for another decade several promising parallel approaches simultaneously on the market. Some general statements nevertheless can be made in order to show at the same time the most important points in architecture, in programming and operating parallel computing systems.

In the following we will grant a certain preference to < highly parallel > MIMD-systems or multiprocessors because of their generality (compare earlier remarks). At the end of this century systems with 256 up to 1024 processors will be quite normal. The specific problem of multiprocessors is the need for a good communication among these 256 or 1024 processors.

It is impossible to connect each of the processors to each other. In the case of e.g. 1024 this would result in the order of 1 million connection lines and control switches for them. The more realistic concept is to choose a favourable connection topology between the processors, where the number of lines decreases to $k \times 1024$ according to the number of edges of the 10-dimensional cube ($k = 10$). Lines in this order of magnitude are controllable. It is assumed in this concept that for each of the processors k lines (edges) go to other processors directly. The other non-direct connections are then to be switched step by step by taking the few direct connections taking mostly k elementary periods in the case of a k -cube. No doubt this consumes more time according to the number of intermediate steps than for the hypothetical case of a totally connected graph. But many applications just of the physical world are by nature of the nearest-neighbour type. Influence of more distant quantities to the computation of a local value will disappear strongly. In many types of numerical computations one can consider the nearest-neighbour-topology as a quite natural one.

The restricted connectivity schema which reduces for instance - as pointed out - the million connections to some thousands is unfavourable for nonnumerical and logic problems. If there are no direct connections for what is actually demanded a certain penalty has to be paid for a transfer.

Communication in a multiprocessor something which has to be taken into account like branches, loops and calls for subroutines became quite natural in the early monoproductors of the von-Neumann-type.

According to the exposition given above the problem of restricted connectivity we have to face with is a technological and topological question. Regarding the fact that optical elements for computation and connection are not available yet, we have to face topological problems taking the contemporary network-technology as a fact.

Five prominent representatives are today:

MIMD - INTEL -iPSC/860

8 - 128 nodes in hyper-cube topology
 60 MFLOPS per node (peak performance/64 bit)
 8 or 16 MBYTES per node
 7.68 GFLOPS peak performance for 128 nodes

MIMD - NCUBE 2

64 - 8192 nodes (Model 6400) in hypercube topology
 or: 16 - 1024 nodes (Model 3200)
 3.3 MFLOPS per node
 64 MBYTES - 512 GBYTES main memory
 27 GFLOPS peak performance

MIMD - PARSYTEC (PARACOM)

(Transputer-based)

64 - 512 nodes (processors) (switchable topology to a certain degree)
 1.5 MFLOPS per node
 1.6 MBYTES memory per node
 600 MFLOPS peak performance

SIMD-Connection machine 2

4 k - 64 k bit-slice processors (in hypercube topology)
 (MFLOPS per node are not noted in this case!)
 128 MBYTES - 2 GBYTES memory
 2.5 GFLOPS peak performance
 (64 k processors , 64 bit)

As a SIMD-machine the repertory of problems which can be handled is restricted.

MIMD - QCDPAX

488 nodes (or more), 2D-nearest-neighbour-mesh structure
 GBYTES memory
 13.75 GFLOPS peak performance

The figures present only a rough impression what a performance is really achieved. The different solutions which compete today each have immanently a certain preference regarding the spectrum of applications.

In this category may be mentioned an Erlangen Project called MEMSY. MEMSY is not an industrial product up to now. Nevertheless this computer is interesting with respect to its topology. MEMSY does not follow the concept of n -dimensional cube, or a similar connection schema. In a k -cube each one of the processors has k connections to its (direct) neighbours. With each new dimension one more connection comes to the single element (vertex) of a cube. One can call it the degree of the graph which represents the k -cube. The degree which is equal to k also may be characterized by saying: k is the local complexity of the multiprocessor.

In contrast with most other topologies and with n -cube the local complexity of MEMSY's nodes never exceeds eight. This fact is identical with the property of scalability. The MEMSY-structure is essentially a pyramid. Theoretically we can expand the basic topology to an infinite structure without coming into trouble with the technological facts. The general schema of a MEMSY-pyramid provides a reduction (rejuvenescence) of the number of processors from level to level in the pyramid till the peak is reached with 1 processor (or element). If we begin at the bottom we may have for instance 1024 processors. For the next higher level we will have then 256, then 64, 16 etc. This corresponds to a reduction by four from level to level.

Inside one level - for instance regarding the 256 level - there is a nearest-neighbour-topology closed to a torus. Apparently the schema may be continued by setting 4096, 8192 etc. processor-levels. The essential property is - as pointed out above - that the local complexity (or connectivity) is limited to eight. This is the strongest definition of scalability.

MIMD - MEMSY (Erlanger Multiprocessor System)

1024 (or more) nodes, 2D-nearest neighbour mesh topology (application or working section) plus pyramid-superstructure (global control section)
 20 MFLOPS per node (provided with arithmetic coprocessor)
 20 - 30 MBYTES memory per node
 20 GFLOPS peak performance (20 GBYTES or more total main memory)
 (21 - pyramid prototype available in 1992)

Aside this property the pyramid-structure brings up good other properties e.g. with respect to broadcasting (of programs and data).

The PARSYTEC-structure is more general or flexible than the hypercube or n -cube solutions. It is expandable in portions of 16 processors. These are

internally fully interconnected. INMOS-Corp. announces a multiprocessor-line where switches allow a rather general connectivity. Nevertheless it is not strongly scalable like MEMSY.

Presenting a rough definition of the term strongly <scalable> we can state:

"A system is strongly **scalable** if performance can be increased by a factor of 1000 or more (in fact: unlimitedly) without changing technology and topology."

This general property is fulfilled with the MEMSY-Topology. Similar topologies fulfilling the same strong condition can be built up from transputer-like elements.

S c a l a b i l i t y is the first of a couple of attributes the computer architect has to obey.

H o m o g e n e i t y is the second one. With respect to the worker processors (devoted to the proper applications) we cannot admit seams of any kind; that means, we do not change technology and topology in the context of the scalability condition. So we do not change the kind of nearest-neighbour connections in the basic array which will configure the preeminent structure of the multiprocessors. In the context of a topology for the working space as part of a multiprocessor this property of homogeneity was also characterized as a seamless topology.

The most obvious and at the same time most simple topology is a torus where there is nowhere a border and nowhere a seam which would make up any inhomogeneity.

Why do we need such a strong homogeneity of this kind? We have on the software side an infrastructure which takes its pattern from data dependences between the parts (or tasks) of the program to be processed on a multiprocessor in parallel. Such data dependences are in general not avoidable and so the infrastructure is.

If the inherent structure of software is distributed on a given hardware then this will interfere in runtime with an infrastructure which is characteristic for the hardware. Therefore it is legitimate to search for a topology which has as few seams or borders as possible. The degree of freedom we have with respect to software-modifications is minimal since the software-structure is owing to the specific problem we are faced with. Therefore the computer architect has to center his attention on hardware homogeneity. It remains nevertheless a most challenging mission to map a given problem or program with respect to the inherent software-structure onto the hardware available. Software has its infrastructure according to data dependences, while hardware has its discrete

processor-oriented structure. Matching both means to solve an NP-hard problem, which can be otherwise characterized by saying, that we have to search for a subgraph-isomorphism. A general solution therefore cannot be presented. Nevertheless a heuristic procedure very often will help to find a practicable solution. This will be the more the case the more homogeneous the underlying hardware has been designed.

4. Future Evolution - paradigmatic approach

When I was asked in 1977 to give some highlights in a 'Conference on Parallel Computers - Parallel Mathematics' in Munich then I gave a couple of theses. These theses are still valid and let me therefore present them with some of today's comments. Thereafter I will present some additional remarks which may complement and update what has been said in 1977.

The seven theses from 1977 were:

- " 1. Computer architecture must be oriented to evolution.
- 2. Contemporary processors are suitable for use as components in the synthesis of more advanced systems.
- 3. Computer design must be concentrated on systems comprising very many identical simple components.
- 4. Programs can be partitioned into tasks which can be computed partially in sequence one after the other, and partially in parallel at the same time. As a result a flow diagram can be expanded into a cluster which can be mapped onto a rectangular processor array.
- 5. The concept of quantitative modularity will replace the current concept of the family of computers.
- 6. The overheads and complexity of a multiprocessor system with limited connectivity do not necessarily increase more than linearly with the number of processors.

7. Content oriented processing does not necessarily require a real associative memory. In many applications vertical data processing is sufficient. "

Today's comments:

ad 1: Evolution - Paradigm :

Parallel computer concepts may start successfully from the proven pattern of monoproductors - often characterized by the term 'von-Neumann-Computer' (or Processor). On the one hand we have a lot of experience with programs and programming languages and on the other hand the concept of coarse grain parallelism is a very promising one. We can follow the way of nature in holding the 'cell' as a selfcontained entity or one of the known processors to build up more powerful and more versatile machine beings. This does not prohibit that some promising other concepts like the dataflow processor will penetrate one day into the field of multiprocessors. Nevertheless the pattern is always the one presented us from nature - to configure more powerful and versatile beings from the basic cells.

ad 2: Universality - Paradigm :

The proven processors have a lot of more capabilities than only to compute. They organize I/O, they carry out an operating system (in a manyfold of facets), they translate formal and natural languages, and they manage data bank operations - in order to mention some. They will therefore also be capable to communicate with other processors on the progress of a common team-work and they will manage the necessary operations to exchange data and programs. There is nothing which is not known from the selfcontained 'life' of a single processor.

ad 3: Homogeneity / Simplicity - Paradigm :

Mass production, fault-tolerant operation, simplicity of communication, synchronisation and operating system as well as the simplicity of programming tools, compilers etc. demand for a very homogeneous and simple hardware. It is nevertheless not a contradiction to provide most versatile elements, i.e. in our case: processors (compare: ad 2).

ad 4: CSP - Paradigm :

Parallel processing means always communication between parts of the program - or between the tasks of the job, running in parallel on a

multiprocessor. The communication is minimised by the concept of coarse grain or by the CSP-Paradigm (Communicating-Sequential-Processes Paradigm). In correspondence with paradigm 3 (compare ad 3) we may base the elementary process to bring a problem or program onto a multiprocessor upon a regular processor array.

ad 5: Scalability - Paradigm :

A regular array secures in any case a restricted neighbourhood of connections and this again brings what is today called the scalability (at the time the thesis, or paradigm, was written the comments were related to 'size-independant topology' and 'quantitative modularity').

To correspond the need of performance, in 1977, there was still the problem of choosing the right member of a 'Family' of computers. Manufacturers of computers created a family - e.g. the IBM 360-Family - in a couple of models which differed in many hardware-aspects. While they were quite different in hardware they nevertheless were compatible regarding FORTRAN , ALGOL etc.

From model to model (of the same family) the hardware technology changed. In a very simple case a multiplication was performed by a table-operation, in an other model the multiplication was based on a subroutine and in more powerful models this operation was performed by a microprogram or by a most sophisticated circuit. Accordingly the customer had to pay a price for the respective technology of the chosen model.

In the era of multiprocessors the price will correspond the configurations of 16 , 64 , 256 or say 1024. So it will be nearly proportional to the number of processors or elements. The technology will be the same - at least in the case of genuine (strong) scalability. In appropriately designed multi-processors exists a 'family' in a quite natural way, in many cases proportional to the 16 , 64 , 256 , 1024 etc. pattern.

ad 6: Nearest-Neighbour-Paradigm :

Limited connectivity may be considered to be identical to the constant local complexity which again is identical to the degree of the underlying graph of the multiprocessor concept. As long as we have no optical elements of a speed much faster than today's electronic switches the necessity of a restricted connectivity exists (The restricted connectivity concept in accordance with the need for regularity is equivalent to the result of scalability.).

ad 7: Space-Sharing-Paradigm:

Space - that means the set of distributed elements, being in particular processors - may be shared between an application section (working area) and a global control section in order to achieve a good overall performance. Various applications beyond it may have a demand each for different numbers of elements (processors). In order to make the best use of available processors the application section may be partitioned into several parts, working then separately.

This concept supports the paradigm 2 (ad 2) which claims for universality of processor elements. It must be possible to individualise the processor elements in favour of the tasks they are faced with. The lowest level of this individualisation is the usual programmability. The highest level would be a temporary individualisation (or modification) with respect to an operation mode like 'Associative-Parallel Processing', 'Reduction Machine-Processing', 'Digital Differential Processing' (DDA-mode), etc. As an example may be mentioned in the following an individualisation (in run-time) with respect to 'Associative-Parallel Processing' or 'Vertical Processing'.

In this high level the concept of individualisation (space-) may be considered a proper extension or corollary of the Space Sharing-Paradigm.

As shown for instance in the context of the associative / parallel computer STARAN and some successors associative operation does not necessarily mean the existence of a specific associative structure. Rather a conventional memory shows very important associative characteristics provided it is driven appropriately (In this context compare also this volume: Ya. I. Fet and W. Händler: Vertical Processing in Parallel Computing Systems). Slight modifications of quite conventional microprocessors will offer the possibility of utilising or evaluating the information associatively. The term 'vertical processing' corresponds to the fact that in this case the information has to be converted to the vertical direction (for instance: in the case of a 32-bit processor 32 items of information are located vertically in the memory waiting for being processed in one-bit slices (machine words) in parallel. An extra-chip design may be forgotten in the case of microprogrammable microprocessors. Three typical implementations of vertical processing in microprogramming technology led to successful experience.

The most prominent problem today is subsumed or even cached under the paradigm 4 (ad 4). In order to formulate it more explicitly there are two main problems in the design and utilisation of multiprocessors.

First the *d e s i g n e r* of a processor-chip has to obey the need for the most general connectivity ever achievable at the time and then to realise the most general and homogeneous topology for a modular multiprocessor which is applicable for a broad range of size, i.e. for 16 , 64 , 256 etc. -processors.

Secondly the *p r o g r a m m e r* or the software-provider has to be equipped with software-tools which makes it practically possible, to partition given jobs into parts resp. tasks which can be computed in parallel and, to distribute these tasks then onto the available processors. Nevertheless this is equivalent to the problem to search an isomorphic mapping from one graph (data dependence graph) to another graph (hardware structure of an existing multiprocessor). This is mathematically rated a NP-hard problem. It means that there is no general solution to the problem. Nevertheless for many important problems can be found an acceptable solution. It is apparent how the two problems are related one to the other. There is basic research necessary from the algorithmic side, from the compiler side and from the side of appropriate languages.

In a long range structures similar to *n e r v e n e t s* are a most promising alternative. These are the most consequent parallel devices. When we began to consider nature in its cellular structure then we recommended to choose today's computers or processors as elements. At the same time we suggested to compose multiprocessors from the computing elements we know best from about 40 years in practice and to take for this the most simple ones but in a big number of pieces (elements). Nevertheless after having more complete insight into the functionality of nerve nets it may be possible to compose our multiprocessors from tiny nerve-like elements in a huge number which we have to allocate in order to function hierarchically in a multitude of layers. At the present nerve nets in nature are incompletely decoded, so that in a majority of cases a direct application of a nerve-net - concept does not seem to be possible. Parts of the business can be realised on EGPA-like structures (compare above: MEMSY-Pyramid) with its levels or layers, which are suited to stepwise abstraction or extraction of concepts. This consideration suggests that multiprocessors and structures similar to nerve nets will come together more and more. Sometimes today the possibilities or potentials of such structures are rather overestimated.

5. Final remarks

Parallelism is not a problem per se. It is rather a way to the proper problems. Nevertheless it is evident that in the majority of future problems the conventional mainframe processor will not satisfy the need of computer performance and dependability (fault tolerance). The size of microstructures and speed of light set serious limits of physical nature to what can be achieved via a single processor.

The real problem is therefore to make available e.g. artificial intelligence, expert systems, and the like. And it must be offered in a most efficient and reliable way. The way to these targets - and it is necessary, to say: the only way - is to become parallel with all thoughts about computing.

The Teraflop-Computer will come in a remarkable number at the end of this century (10^{12} Flops). There is no doubt about it. The Teraflop-Multiprocessors will represent a means to an end - to numerical simulation, to huge data bases, to artificial intelligence, to expert systems and to many things else. The main question will nevertheless be whether we will wisely utilize these tools to preserve our world and its natural sources from destruction and to prevent the humanity from serious conflicts or wars.

There will also be a serious limit in what we can do. The Chaos-Theory e.g. says in its essence that it is impossible to predict the weather more than for some days. A rough estimation can only be given but no more. The famous example is the story that a tiny butterfly in Paris with its teeny weeny flapping of wings may contribute decisively to the weather in Budapest.

No Teraflop-Multiprocessor and no program whatsoever can overcome this utmost limit, set by the Chaos-Theory - a limit which is more real than those from AMDAHL, MINSKY, and others.

KEYNOTE REMARKS

Vladimir A.Melnikov

The Institute for Problems of Cybernetics

USSR Acad.Sci.

Vavilova st., 37, Moscow, 117312, USSR

It is difficult to overestimate the needs in high-speed computing (or supercomputing). The huge amount of calculations is needed for solving such problems as creation of new chemical materials (new drugs for example), visual and sound pattern recognition, weather forecasting, aircraft engineering, space exploring, simulation of complex physical processes (including real-time models), etc. Since the appearance of the first electronic computers, working frequency of electronic circuits is approximately twice increasing every two years. It is considered that such a growth can keep at most until the beginning of the next millennium (in case of introducing in computer industry of new methods such as opticfiber and organic technologies). Architectural supporting, and first of all *parallel data processing*, is another way for the performance speedup.

Nowadays parallel computers (or parallel environments) are widespread around the world of calculations. Parallelism is used at several levels: parallel execution of different machine instructions, simultaneous execution of similar functions upon homogeneous data in pipelined and array processors, simultaneous execution of independent machine instructions in processors with Very Long Instruction Word, simultaneous execution of different parts of the same program on different processors in multiprocessor environment. Several levels of parallelism are usually used in computer systems, but sometimes it is difficult to precisely categorize them (systolic array is the good example). Modern supercomputers usually have one or a few vector-pipelined processors.

On the other hand, *multiprocessor systems with distributed memory* give the potential possibility of the unlimited growth of calculation

performance due to the increasing number of processor elements. Simplicity of dynamic reconfiguration of interprocessor links makes multiprocessor environment architecture adaptable for solving the concrete problem. Dynamic adaptation enables us to partly compensate inevitable performance decrease which arises in connection with complication of interprocessor communications. One of the most accessible parallel architecture now is a multitransputer system with some kind of Personal Computer or Workstation as host machine. The price/performance index for such systems is sufficiently low if comparing with powerful supercomputers.

Parallel computer systems demand special features of its *software*. Operating system must support effective concurrent execution of communicating processes. Programs for a parallel computer have to take into account the object system architecture. Problem solving algorithm is the basis of every program. Consequently, programming of the parallel computer begins with choosing the algorithm and mathematical methods according to object architecture. This consideration explains the growth of importance of parallel algorithms development. To solve these problems parallel programming languages are needed. To enable software portability it is necessary to increase the level of parallel language (in comparing with object architecture). Existence of the large fund of programs created during the last 40 years ("dusty decks") requires the development of automatic vectorizing/parallelizing software tools which make it easier for the user to transfer programs from the conventional von-Neumann architecture to parallel/vector environment.

Good enough *automatic vectorizing systems* already exist for vector supercomputers. Although a human factor is minimized in the vectorization process we cannot avoid its intervention completely. Today most of *automatic parallelizing systems* for the multiprocessor environment are being at the experimental stage. Parallelizing at the level of subroutine calls requires analyzing of the whole program text, i.e. the global optimization. Now it is important to expand the class of automatically parallelized programs. Programs which make use of dynamically controlled memory is the case of great interest. It is

desirable for the automatic parallelizing system be able to modify algorithms for parallel execution. It seems very promising to use Artificial Intelligence methods for the partial restoration of program semantics and optimal reflection of this semantics onto the object computer architecture taking into account features of the object operating system.

Effective algorithms for configuring and routing of calculations are needed for the better performance of parallel programs. Promising parallel high-level languages and automatic parallelization technique will be integrated into parallel programming systems which support all stages of parallel program life cycle.

To solve the above mentioned R&D problems new ideas and their practical implementations are necessary. Of course, it is necessary to have human beings who are able to do this. Thus, I think that the *social aspect of supercomputing* will be very important during the next years. It is necessary to foresee incorporation of special subject on supercomputing into programmes of the universities and technical colleges. Also, it is very important to provide a wide access to modern supercomputing for young researchers and engineers.

PARALLEL PROGRAMMING LANGUAGE DEVELOPMENTS

R H Perrott
Department of Computer Science
The Queen's University
Belfast BT7 1NN
N. Ireland

Abstract

This paper is concerned with the topic of parallel programming language design and implementation. An outline of the main issues affecting the spread and acceptance of parallel computing is given followed by consideration of specific approaches to parallel programming.

1. Issues Affecting Parallel Systems

In the case of sequential computers the architectural model, the programming paradigms and the method of constructing algorithms all have a single model of computation to work from. In the case of parallel computers there is no longer a single architectural model to represent parallelism but rather a variety of different parallel architectures.

The main issue affecting the architectural model is how to organise multiple processors to execute in parallel. Two decades of research and implementation have produced a variety of different machines, each with its own model of computation. In the case of parallel software the choice of programming language is no longer confined to a single approach. The main

division of these languages is into either imperative or declarative languages. The declarative group can be further divided into logic and functional languages while the imperative group consist of procedural and object oriented languages.

All the various languages that have been proposed offer some different way of capitalising on the power of parallel machines. To date it is not clear if any one approach is substantially better than any other as enough experience has not yet been accumulated. In many cases the concepts have not been efficiently implemented on parallel machines.

The third important aspect of programming parallel systems is the choice of algorithm. Studies have shown that transferring an efficient sequential algorithm to a parallel machine results in an inefficient parallel algorithm. It is now apparent that the design and construction of a new parallel algorithm for a particular application area can produce major performance improvements.

Hence, in the case of parallel systems there are three important and contributing factors, namely, the architectural model, the programming language and the choice of algorithm. The following sections concentrate on the programming language.

2. Approaches For Parallel Programming

Essentially there has been three main methods used to promote the wider use of parallel processing, namely

- (i) extend an existing sequential language with features to represent parallelism.

The advantage of extensions is that existing software can be transferred to the new parallel machine with relative ease. This is possible because programmers are already trained in the base language and can introduce the extensions gradually as they become more familiar with the situation in which they should be used and the effect they produce. However, experience to date has shown that extension languages have been limited to a certain range of hardware and to machines with a small number of processors.

- (ii) use a sequential language but rely on the compiler to detect which parts of the program can be executed in parallel.

Most of the work in this area is based on Fortran and examines the DO loops of the program to determine if it is possible to spread the iterations of the loop across different processors. The advantage of such an approach is that existing sequential programs can be moved relatively inexpensively and quickly to the parallel machine. This can represent a substantial saving in development costs and is an attractive proposition for many purchasers of a new parallel machine. In the case of the construction of new programs it is advisable that a programmer has some knowledge of the detection techniques if as much parallelism as possible is to be detected. Such an approach inhibits the development of parallel languages and algorithms as it is confined to a sequential notation.

- (iii) develop a new parallel language

In this case a completely new parallel language is developed ignoring all existing languages and applications. The main advantage of this approach is that a coherent approach to parallelism is presented. The parallel notation will enable a user to express directly the parallelism in an application and, in addition, will assist with the development of new parallel algorithms. However, it does mean that a user will have to rebuild the entire software base in the new language which is a labour intensive, expensive and perhaps an error prone exercise.

These three approaches have been widely used over the years, and the following sections highlight some of the main examples.

3. Types Of Parallel Languages

Since the early 1960s several proposals have been made in order to classify machine architectures, this has resulted in a 'periodic table' of computer architectures. It has not proved as easy to do this for programming languages since their characteristics do not lend themselves to easy classification. For the purposes of this article the following classification is used.

3.1 *Imperative Languages*

The imperative languages have the longest history of any language group. It was this approach which was used to program the early von Neumann machines and which has been refined over the years. The main characteristic is that the language enables a user to specify a sequence of operations to perform the calculation; the sequence of operations changes the state of the machine hence the description as 'state oriented' languages. The central operation of these languages is that of assignment which enables any location in the machine to be changed. Within this broad category there have been separate developments under the headings of procedural languages and object oriented languages.

3.1.1 *Procedural Languages*

The procedural approach in a language means that constructs are provided for modularising the program code; this capitalises upon the experience which has been gained from the design and implementation of sequential languages.

(i) *Implicit Approach*

This is an attempt to make the transfer from sequential to parallel programming as easy as possible for a user by placing the burden of parallelism detection on the compiler. In this way a user does not have to learn a new language or get involved with the complexities of parallel programming. The majority of this work is Fortran based.

In general, the main strategy of a parallelisation system is to examine nested DO loops, with the objective of vectorising the innermost loop and parallelising the outermost loop (assuming hardware support). The methods rely on data dependence analysis techniques which determine the flow of data in a program. This, in turn, enables statements which can be executed in parallel to be identified.

At present the challenge in the field of parallelisation is to perform whole program optimisation. This requires full

interprocedural analysis (the tracking of data across procedure calls) to be performed on a user program.

(ii) **Explicit Approach - Array and Vector Processing Languages**
Languages for array and vector processor machines should provide features which enable a programmer to express the parallelism in the data of the application as well as its subsequent manipulation.

This approach has been particularly widely promoted by array processors such as the Illiac IV^{2,8}, AMT Distributed Array Processor¹⁵, Connection Machine⁷ etc. Typical of these languages are CFD Fortran, Vectran, Fortran Plus. Exceptions to this Fortran trend are the languages Actus¹⁸ which is Pascal based and the language Booster¹⁶. The first two projects were carried out in the USA, the last three in Europe.

If past experience is any guide, then the new Fortran standard referred to as Fortran 90 will be the dominant parallel programming language in this area as the majority of manufacturers will implement the standard, and the majority of users will prefer to use a standard programming language.

(iii) **Explicit Approach - Multiprocessor/Distributed Systems**
In a concurrent programming environment, a programmer requires not only program and data structures similar to those required in a sequential programming environment but also mechanisms to control the interaction of the processes - processes which are proceeding at fixed but unknown rates. The situations in which processes interact can be divided into two categories. The first situation occurs whenever several processes wish to access a resource at the same time - **mutual exclusion**. The second situation occurs whenever processes are co-operating, they must be correctly synchronised with respect to each other's activities - **conditional synchronisation**.

Experience and research produced a series of solutions such as semaphores, critical regions and conditional critical regions, finally resulting in the monitor¹⁹. The monitor was influenced by the class concept of Simula. Several languages have been

developed using this technique, for example, Concurrent Pascal⁴ and Pascal Plus²².

A second technique which can be used to solve these problems is based on message passing. This technique has been adopted in languages such as Ada¹ and Occam¹². In Ada, for example, tasks first synchronise their activities and then communicate directly without the help of an intermediate data structure. Hence one process may have to wait for another process to arrive, when it does the processes can exchange messages directly; such an encounter is referred to as a rendezvous. In addition, non deterministic selection has been introduced so that given a choice between several alternatives one is chosen at random.

3.1.2. *Object Oriented Languages*

In the last decade an approach to imperative programming which has gained widespread popularity is that of object oriented programming. In this approach an object is used to integrate both data and the means of manipulating that data. Objects interact exclusively through message passing and the data contained in an object are visible only within the object itself. Objects are intended for structuring programs in a clear and understandable way.

The behaviour of an object is defined by its class, which comprises a list of operations that can be invoked by sending a message to the object. All objects must belong to a class. Objects in a class have the same properties and can be manipulated using similar operators. The definition of an object class can act as a template for creating instances of the class. Each instance has a unique identity but has the same set of data properties and responds to the same set of operators.

Inheritance, allows a class to be defined as an extension of another (previously defined) class. Typically when a new object class is created a place for it is defined within the class hierarchy. The effect of this is that the new class inherits the state and operators of its superclass in the hierarchy.

There are several possibilities for the introduction of parallelism, namely,

- (i) objects are active without having received a message;
- (ii) objects continue to execute after returning results;
- (iii) messages are sent to several objects at the same time;
- (iv) senders proceed in parallel with receivers.

These possibilities can be realised by associating a process with each object. The introduction of processes means that several processes can be active at the same time. Facilities which provide synchronisation and mutual exclusion are usually supplied as built-in classes, for example, the semaphore. Examples of languages which have followed this approach are Concurrent Smalltalk²⁴, Emerald³.

3.2 *Declarative Languages*

Another major approach to language design is that represented by the declarative approach. In this case a notation is used to represent what is to be calculated rather than how the calculation is to be performed. How the calculation is performed depends on the implementation of the language and is not meant to be a concern of a programmer. A major difference between this approach and the imperative approach is that in the declarative approach there is no concept of state; a programmer does not work or think in terms of state transitions.

Essentially there are two main models

- (i) the logic model based on the relation between entities;
- (ii) the functional model based on the use of functions.

However, a major concern is that declarative languages have not been efficiently implemented; the advent of parallel architectures represents an opportunity to explore their efficient implementation.

3.2.1. *Logic Languages*

In this type of programming the underlying computational unit is the relation and the approach concentrates on proving entities rather than building entities as in imperative languages. A logic

program is constructed as a finite set of clauses where a clause is a formula of the form $H \leftarrow B_1, B_2, \dots, B_n$ $n \geq 0$ where H is referred to as the clause's head and $B_1, B_2, \dots, B_n$ its body.

The earliest programming language based on this approach is Prolog (PROgramming in LOGic) which was developed in the early 1970s by Colmerauer and his colleagues at the University of Marseilles^{5,14}. Programming in Prolog involves the specification of a set of facts and rules to form a database of knowledge on a subject. Questions may be asked of this knowledge database via Prolog which is then responsible for making inferences.

The basic data manipulation operation in logic programs is unification which results in a substitution. Operationally a substitution can be thought of as a simultaneous assignment of values to variables except that a variable can only be assigned a value once, and the value assigned can itself be another variable or a term containing variables. Hence logic variables have the single assignment property; initially they are unbound but once they receive a value, by unification, they cannot be changed. In logic programming systems there are several opportunities for introducing parallelism, namely:

- (i) AND-parallelism comes from trying to solve many parts of the problem simultaneously, that is, pursuing solutions to a number of subgoals at the same time. The main goal is solved if the first subgoal is solved AND, the second AND, the third etc.
- (ii) OR-parallelism involves the evaluation of a problem in several ways at the same time. The problem is solved if the first clause is solved OR, the second OR, the third etc.
- (iii) Parallel pattern matching corresponds to parallel versions of the unification algorithm used by the language while processing.

In AND-parallelism because the processes are working on the same problem, they share a common set of data and variables. A change to a variable by one process may affect the computation of another process, may even make it fail. In addition, when two processes share an unbound variable and both wish to assign it a value, care must be taken to ensure that they assign it a common value. If they assign different values to the variable and this

discrepancy goes undetected, an invalid computation may result - this is known as the binding conflict. Identifying code which is free of binding conflicts and could safely execute in parallel requires extra analysis at both compile and run time.

3.2.2. *Functional Languages*

A functional program comprises a set of equations describing functions and data structures which a user wishes to compute. The application of a function to its arguments is the only control structure; there are no features which deal with assignment, conditional or looping structures as in imperative languages. The application of a function to a variable can produce a new variable and a new value but never change an old value. Functions are regarded as in the mathematical sense in that they do not give rise to side effects that can occur in, for example, Fortran functions. As a consequence a value of a function is determined solely by the values of its arguments, a property which is referred to as referential transparency; side effects mean the loss of referential transparency. In this approach to programming there is no concept of state which is associated with imperative languages.

The benefits of parallel processing for functional programming are realised by means of the Church Rosser Theorem which states that any reduction sequence which yields a reduced form may be used in the evaluation of a function. More specifically, if a given lambda expression is reduced by two different reduction techniques and if both reduction sequences yield a reduced form then the reduced forms are equivalent up to renaming of bound variables. If the evaluation sequence does not matter then the reduction can be done in parallel without worrying about the order in which they finish.

If functions do not have side effects then it makes no difference in which order they are executed and it is therefore possible to evaluate them in parallel. The only delay may be that a function must wait on a result being produced by another function. The compiler should analyse the program to decide on the evaluation order.

Parallelism in functional languages is therefore manifest through data dependencies and the semantics of primitive operators in contrast to conventional languages where explicit constructs are typically used to invoke synchronisation and coordinate concurrent activities. In most systems parallelism is detected by the system and allocated to the processors automatically - implicit parallelism.

In certain classes of functional languages the mapping of a process to a processor can be determined optimally but in the general case the optimal strategy is undecidable so heuristics such as load balancing are often used instead. To allow users the possibility of influencing the behaviour of the program various researchers have designed extensions to functional programming languages, such languages are referred to as parafunctional programming languages. These languages include a mapping notation that helps spread a computation across a parallel machine; a user may indicate on which processor an expression should be executed.

Hudak¹⁰ argues that functional programming is just an evolution of the imperative approach since languages like Fortran had expressions as an important feature with a mathematical flavour and subsequent languages like Pascal built on this theme using the function. Functional programming is therefore the logical conclusion since everything is treated as an expression.

4. Summary

A complete solution to the problem of parallel computing involves several components, namely, the design of new parallel algorithms, the type of programming language and the characteristics of the underlying architecture. Currently it is the programming languages (and tools) which require the most urgent attention. This is an area of research and development which has received considerable attention in Europe over the last few decades. It is easy to attribute substantial and innovative proposals to the work of individuals and groups in European countries.

5. References

1. Ada 1983. Reference Manual for Ada Programming Language.(ANSI/MIL-STD 1815), United States Department of Defense, Washington, D.C.
2. Barnes G.H., Brown N.M., Kato M., Kuck D., Slotnick D., and Stokes N.A. 1968. *'The Illiac IV Computer'*. IEEE. Trans. on Compuyers, 17 746-757.
3. Black A., Hutchinson N., Jul E., Levy H. and Carter L. 1987. *'Distribution and abstract types in Emerald'*. IEEE. Trans. Software Eng SE 13, 1 (Jan). 65-76.
4. Brinch Hansen P. 1975. *'The Programming Language Concurrent Pascal'*. IEEE Trans. on. Eng., 1, 199-207.
5. Colmerauer A., Kanoui H., Roussel P. and Pasero R. 1973. Groupe do Rechercho on I.A.Universite JAIX-Marseille.
6. Halstead R H Jr. 1985. *'Multilisp; A language for concurrent symbolic computation'*. ACM Trans Program Lang Systems, 7, 4, 501-538.
7. Hillis W D. 1985. *The Connection Machine*, MIT Press.
8. Hord R M. 1984. *The Illiac IV-The First Supercomputer*. Computer Science Press, Maryland.
9. Hudak P. and Smith L. 1986. *'Parafunctional programming: a parading for programming multiprocessors'*. In Proceedings, of the 13th ACM Symposium on Principles of Programming Landuages. 243-254.
- 10.Hudak P. 1989. *'Conception, evaluation, and Application of Functional Programming Languages'*. ACM Surveys, 21, 3, 359-411.
- 11.Hudak P., and Wadler P. Eds. 1988. Report on the Functional Programming Language Haskell. Tech. Rep. YALE U./DCS/RR656. Department of Computer Science, Yale University.
- 12.Inmos Ltd. 1984. *Occam Programming Manual*. Prentice-Hall, Englewood Cliffs, New Jersey.
- 13.Kelly P. 1989. *Functional programming for loosely coupled multiprocessors*. Pitman Press.
- 14.Kowalski R. A. 1974. *'Predicate Logic as a Programming Language'*. IFIP 74, 569-574.

15. Parkinson D. 1983. '*the Distributed Array Processor*'. Computer Physics Communications ACM, 28, 325-336.
16. Paalvast E M. and Sips H J. 1989. '*A high level language for the description of parallel algorithms*'. Parallel Computing 89 Published by North Holland.
17. Paul G. and Wilson W. 1975. '*The Vectran language: and experimental language for vector/matrix array processing*'. IBM Palo Alto Scientific Center, Report 6320-3334.
18. Perrott R H. 1979. '*A Language for Array and Vector Processors*'. ACM Trans. on Programming. Lang. Systems. 2, 177-195.
19. Perrott K. 1987. *Parallel Programming*. Addison-Wesley.
20. Stevens K. 1975. '*CFD-A FORTRAN-like Language for the ILLIAC IV*'. Sigplan Notices, 10, 72-80.
21. Sugimoto S., Agusa K., Tabata K. and Ohno Y. 1983. *A multiprocessor system for Concurrent Lisp*. In Proceedings of the 1983 International Conference on Parallel Processing (Bellaire, Michigan) 135-143.
22. Welsh J. and Bustard D W. 1979. '*Pascal Plus-Another Language for Modular Multiprogramming*'. Software-Practice and Experience, 9, 947-957.
23. Wirth N. 1977. '*Modula: a Language for Modular Multiprogramming*'. Software-Practice and Experience, 7, 3-35.
24. Yokote Y. and Tokoro M. 1986. '*the design and implementation of Concurrent Smalltalk*'. In Proceedings of Object-Oriented Programming Systems, Languages and Applications 1986. Sigplan Not (ACM), 21, 11, 331-340.

A VIEW OF SYSTOLIC DESIGN

CHRISTIAN LENGAUER

*Laboratory for Foundations of Computer Science
Department of Computer Science, University of Edinburgh
King's Buildings, Edinburgh EH9 3JZ, Scotland, U.K.*

ABSTRACT

This is an overview and bibliography of recent results and current issues in the mechanical synthesis of systolic arrays.

1. Introduction

A *systolic array*^{34, 35} is a network of processors that are locally and regularly connected. This special-purpose computing paradigm happens to support the parallel implementations of iterative algorithms in a variety of areas, e.g., numerical analysis, signal or image processing and graph theory. Among the more complex algorithms that have been expressed as systolic arrays are the algebraic path problem,^{59, 60} Toeplitz algorithms,¹⁵ Jacobi algorithms⁶⁵ and pyramidal algorithms.^{41, 43}

Mechanical methods can transform algorithmic descriptions that do not specify concurrency or communication – usually functional or imperative programs – into abstract systolic arrays described by functions that distribute the program's operations over time and space. This process is called *systolic design*. The distribution functions can then be refined further and translated into silicon for fabrication of a VLSI chip or into a distributed program for execution on a programmable processor array.

This paper is an overview of recent results and current issues in systolic design.

- Sect. 2. Most presently implemented methods can synthesise a class of (rather simple) systolic arrays autonomously with a guarantee of correctness and time-optimality (i.e. fastest possible parallel execution time). These are, essentially, systolic arrays whose source specifications are *uniform recurrences*.
- Sect. 3. For a non-uniform source, human intervention is often necessary in the systolic design process, but there are recent advances in the search for automatic schemes to turn more general forms of recurrences into uniform recurrences.
- Sect. 4. Even when the syntactic requirements on the source can be satisfied, the systolic design process may still yield an unsatisfactory solution, for example, one with more processors or execution steps than necessary. Systolic design methods guarantee time-optimality with respect to the source only. A different source may specify the same problem but lead to a faster systolic array.

- Sect. 5. A uniform source guarantees linear distribution functions. Sometimes a systolic solution can be improved by breaking the linearity.
- Sect. 6. A systolic design may be complicated by a limitation of the number or nature of processors or channel connections. Time-optimality can usually not be guaranteed in the presence of resource limitations.
- Sect. 7. A realisation in hardware often requires control signals that tell cells what to do when. Methods for synthesising such signals exist and are being improved.
- Sect. 8. A realisation in software requires the generation of computation and communication code for each cell and communication code for the injection and extraction of data to and from the array.
- Sect. 9. Theoretical issues of systolic arrays and systolic design will also be reviewed. An overview of this kind is, by necessity, biased. This survey is intended to be comprehensive, but it cannot be complete. The focus is on mechanical systolic design based on uniform recurrences. Elsewhere there is an overview that also considers other, earlier methods.¹⁹

2. Uniform Systolic Design

2.1. The Source Description

The starting point of the mechanical systolic design methods that we review is a functional program^{11, 46, 50, 57} or an imperative program,^{27, 39} both without parallel commands and communication directives. Both adhere to the format:

```

for  $i_1$  from  $lb_1$  by  $st_1$  to  $rb_1$  do
  :
  for  $i_r$  from  $lb_r$  by  $st_r$  to  $rb_r$  do
     $i_1 \cdots i_r$ 

```

The loops define a space Φ of points called the *index space* ($\Phi \subset \mathbb{Z}^r$); the coordinates of the points are the loop indices. The bounds lb_k and rb_k are linear expressions in the loop indices i_1 to i_{k-1} ($0 < k \leq r$) and in additional variables that specify the problem size; the bounds must make the index space a convex polyhedron.⁶⁴ The steps st_i are constants. We call $i_1 \cdots i_r$ the *basic operation* of the program. It can be functional (without reassignments) or imperative (with reassignments) but must be without non-local references other than to the variables indexed by the loop indices. There is a formal correspondence between the functional and imperative style.⁷

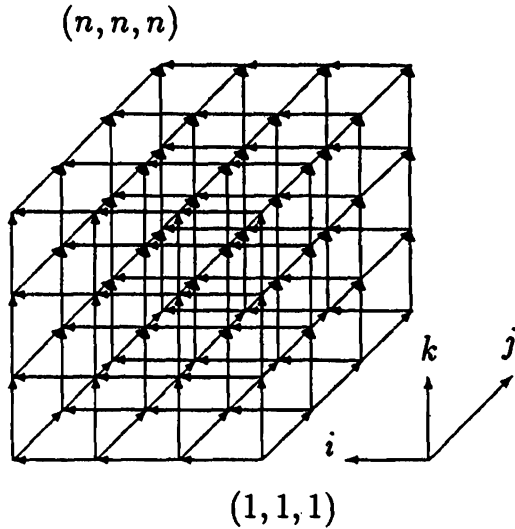
In the functional style, the basic operation must contain a set of *uniform recurrences equations* (UREs),^{32, 50} i.e., equations of the form:

$$v(I) = f_v(w(J), \dots) \quad \text{where } I = (i_1, i_2, \dots, i_r) \in \Phi \quad \text{and} \quad J = I - \vartheta_v$$

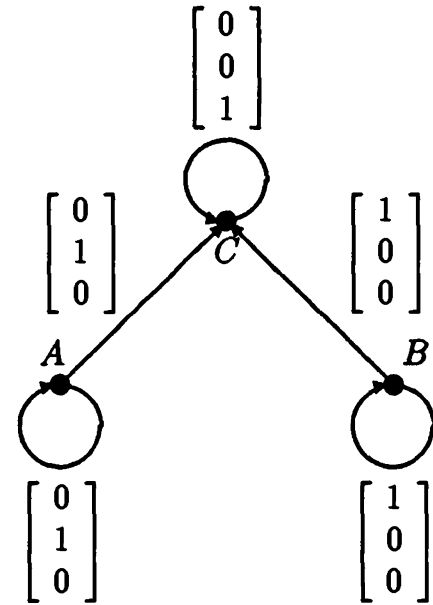
The three dots in the arguments list of f_v stand for an arbitrary but fixed number of similar arguments. The constant vectors ϑ_v are called *dependence vectors*; the (partial) order imposed on the computation by these vectors may not be violated

Dependence Matrix: $[\vartheta_A, \vartheta_B, \vartheta_C] = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Dependence Graph ($n=4$):



Reduced Dependence Graph:



2.2. The Target Description

The systolic array is described by two functions; further functions are derived from them. Let $\mathbb{Z}$ denote the integers and $\mathbb{Q}$ the rationals:

step : $\Phi \longrightarrow \mathbb{Z}$, $\text{step}(I) = \lambda I$, $\lambda \in \mathbb{Z}^r$.

step specifies the temporal distribution. λ is the *scheduling vector*. I is computed at step λI .

place : $\Phi \longrightarrow \mathbb{Z}^{r-1}$, $\text{place}(I) = \sigma I$, $\sigma \in \mathbb{Z}^{(r-1) \times r}$.

place specifies the spatial distribution. σ is the *allocation matrix*. I is computed at location σI .

The mechanical derivation of a shortest distribution in time employs techniques of integer programming in the functional style⁵⁷ and of program transformation in the imperative style.²⁷ One chooses a compatible distribution in space by a search.^{27, 46} With this choice, one can optimise other aspects of the array, e.g., its throughput, number of processors or channels, or storage registers per processor. The square matrix in which step occupies the first row and place occupies all other rows is called the *index transformation matrix*.

If step and place are given, the direction, speed and layout of all data travelling through the array are determined. Let V be the set of variables of the program:

flow : $V \longrightarrow \mathbb{Q}^{r-1}$, $\text{flow}(v) = \sigma \vartheta_v / \lambda \vartheta_v$, $\vartheta_v \in D$.

flow specifies the velocity with which elements of a stream travel at each step. Stream v is called *moving* if $\text{flow}(v) \neq 0$ and *stationary* if $\text{flow}(v) = 0$.

$\text{input} : V \longrightarrow (\Phi \longrightarrow \mathbb{Z}^{r-1}), \quad \text{input}(v(I)) = \text{place}(v(I)) - (\text{step}(v(I)) - fs) \cdot \text{flow}(v(I)).$

input specifies the layout of the data at the first execution step, fs . (This function has also been called *pattern*.)²⁷ The layout at the k -th following step is $\text{input}(v(I)) + k \cdot \text{flow}(v(I))$.

2.3. Functional vs. Imperative

As is usually the case for functional vs. imperative programs, the functional style is based on a richer and cleaner theory. The systolisation in the functional style is independent of the problem size. Integer programming is NP-complete²³ but, in the restricted setting of systolic arrays, it can often be solved efficiently.

To systolise a program in the imperative style, its loops must be expanded for a fixed problem size. Generalisation techniques can then make the step function independent of the problem size.³⁹ The imperative method can take other semantic aspects than independence into account, like idempotence and neutrality.²⁷ Its parallelisation scheme can also generate non-systolic solutions.⁴⁰ There has been much, usually, less formal work on the parallelisation of non-uniform loop programs; we do not review it here.⁶⁷

2.4. Implementations

Early implementations of the functional style include DIASTOL^{20, 21}, ADVIS⁴⁷ and SYSTOL,⁴⁹ among others.⁶⁶ Crystal started out as a system for systolic design¹¹ and subsequently targeted other machine models like Hypercubes.¹² More modern and mature systems are PRESAGE⁶⁶ and ALPHA;^{22, 44} they can assist in the synthesis of more complex arrays (Sect. 3). PRESAGE respects resource limitations (Sect. 6). The imperative method has also been implemented and fitted with a graphic displayer that simulates the execution of systolic arrays.^{27, 28}

3. Syntactic Adaptations

Uniformity is a rather restrictive requirement on recurrences. Many loop algorithms are more easily phrased in a non-uniform way. Research is presently being conducted in transforming non-uniform into uniform recurrences. In cases where the deviation from uniformity is not too serious, mechanical methods can be hoped for.

Recently, methods of transforming affine recurrence (AREs) into uniform recurrences have appeared.^{51, 52, 54, 68} An *affine recurrence equation* differs from a uniform recurrence equation only in the format of the arguments' index vectors. In an affine recurrence, it is $J = A_v I - \vartheta_v$, where A_v is a constant $r \times r$ matrix; in a uniform recurrence, A_v is the identity matrix. The non-identity of A_v may cause the dependence to be non-injective; only injective dependences enforce the full data pipelining required in systolic arrays. Uniformisation methods amount to the introduction of more pipelining by adding compatible dependences. These methods can also handle data reflections in the systolic array, i.e., the case where flow is not constant for a variable.

A further mild deviation from uniformity is a set of nested loops with several basic operations (where scoping is indicated by indenting):^{41, 43}

```

for  $i_1$  from  $lb_1$  by  $st_1$  to  $rb_1$  do
  for  $i_2$  from  $lb_2$  by  $st_2$  to  $rb_2$  do
    for  $i_3$  from  $lb_3$  by  $st_3$  to  $rb_3$  do
       $i_1:i_2:i_3$ 
     $i_1:i_2$ 
  
```

This can be made uniform by letting the inner loop apply $i_1:i_2$ in an added step.

In the functional setting, a source of this form corresponds to a set of AREs in which the length of the index vector varies between different variables. There are also techniques for obtaining a linear step function directly from this kind of AREs.⁴⁵ They may yield a faster execution than can be obtained via a uniformisation. However, only place functions for UREs, not for AREs, enforce local communication (i.e., a bound on the communication length).

4. Semantic Adaptations

The time-optimality of the systolic solution is only with respect to the source program. Different source programs with the same input/output behaviour may prescribe different dependences and therefore lead to systolic solutions of varying quality. This occurs, for example, in the presence of commutative or associative operations.

1. $n \times n$ Matrix Product:

The cumulative additions ("inner product steps")³⁴ in a matrix product can be performed in any order, but one must choose a fixed accumulation order in both the imperative²⁷ and the applicative source.^{20, 51} In certain situations (e.g., band matrices),²⁷ the choice of order makes a difference to the execution time of the systolic solution.

2. Dynamic Prog.: $(\forall i, j : 0 < i, j \leq n : c_{i,j} = w_{i,j} + (\min k : i < k < j : c_{i,k} + c_{k,j}))$

The minimum must be accumulated from the middle to the two ends of the interval (i, j) in order to make a systolisation possible; this keeps the treatment of i and j symmetric.^{21, 26, 52}

There are other examples. If the expense in processors and communication is justified, a sequence of associative operations can be turned into a tree, in which each level of nodes can be performed in parallel, from the leaves to the root.^{41, 43} A small collection of semantic transformation techniques has been published recently.⁵³

The size of the search space makes a systematic identification of useful application orders a particular challenge which has not been addressed by systolic design so far. A desirable specification would be a minimally ordered source program, as is possible in GAMMA^{2, 3} in the functional and Unity¹⁰ in the imperative style. But these languages are too general to support an implemented scheme of systolic design. They leave too many implementation choices because they are targeted at a much wider range of architectures than systolic arrays. To support the mechanical

development of systolic implementations, we need to address issues in the specification that need not be addressed in GAMMA or Unity programs. One requirement is that a dependence graph must be constructible from the specification.

5. Non-Linear Distributions

Uniform systolic design delivers linear distribution functions. In certain cases, non-linear distribution functions improve a systolic solution or make it at all possible. Non-linearity usually means piecewise linearity. We give two examples:

1. Processor-Time Minimal $n \times n$ Matrix Product:

Cappello⁹ presents a systolic array for matrix-product that has $\lceil (3/4)n^2 \rceil$ processors and requires $3n-2$ execution steps. The best linear distribution requires n^2 processors. Cappello saves a quarter of the processors by overlapping the corner of the array that is inactive in the first few steps with the corner that is inactive in the last few steps. To keep communications neighbouring, the two-dimensional array must be made cylindrical, i.e., curved in the third dimension. $\lceil (3/4)n^2 \rceil$ processors is minimal: the step with the most parallelism requires that many.

2. The Algebraic Path Problem

The algebraic path problem,^{59, 60} instances of which are the shortest paths problem and reflexive transitive closure, is solved by Gauss-Jordan elimination. It can be divided into three successive phases:⁶⁰

1. the decomposition of the input matrix into two triangles,
2. the inversion of the two triangles, and
3. the multiplication of the inverted triangles.

The step function of the according systolic solutions is piecewise linear with one linear piece per phase.^{28, 36} The previously mentioned uniformisation methods have been tested on the algebraic path problem.^{51, 52}

The algebraic path problem has been the most popular object of systolic design; new solutions are still emerging. Arrays have been proposed that are processor-time minimal for the standard source, with roughly $5n$ steps and $\lceil n^2/3 \rceil$ processors for an $n \times n$ matrix.^{5, 63}

6. Resource Limitations

Systolic design methods that return a time-optimal solution disregard resource limitations. Resource limitations fall into three classes:

1. Absence of Dimensions from the Processor Space.

If the systolic design postulates more dimensions of processors than are available, a *projection* of the processor space becomes necessary. Dimensions of the processor space can be traded off to time, yielding a multi-dimensional step function, which can then be made one-dimensional again.

- The simplest way of projecting is by eliminating loops from the source program, or hiding them inside the basic operation.^{41, 43}
- There are a number of methods that accept arbitrarily many nested loops. Most work in this area has focussed on the synthesis of arrays with a one-dimensional processor layout – either approaching one-dimensional time directly^{38, 55, 69} or via multi-dimensional time.^{30, 57, 68}
- When a software representation is targeted, one can impose the projection on the distributed program rather than earlier in the design process. While it is advisable to impose resource limitations as late in the programming process as possible, there are indications that this is at too low a level; the multitude of channel communications poses theoretical problems.⁴²

2. Absence of Processors in a Given Dimension.

If a given dimension of the processor layout prescribes more processors than are available, a partitioning or wrapping of the array is required. In *partitioning*, several neighbouring cells in the ideal layout are mapped to the same processor.⁸ In *wrapping*, adjacent portions of the array are superimposed and opposite border processors are connected together.^{8, 48}

3. Absence of Channels.

If channels postulated by the systolic design do not exist, a *re-routing* along existing channels must be performed. Simple cases of re-routing have been addressed by uniformisation methods.⁵¹ On programmable processor networks, the underlying operating system may be able to perform the routing.

7. Hardware Realisation

If the systolic array is realised in hardware and requires a cell to perform different operations at different steps, the cell must be told when to perform what operation. It was first suggested a decade ago that this could be done by communicating *control signals* in addition to the variables needed for the computation.²⁶

Later, formal treatments of this idea emerged.^{11, 51, 52} These methods aim at a time-optimal array with a dimension one less than that of the index space and the control signals are not derived until the space-time mapping has been determined. The methods make no allowances for resource limitations and do not apply for the generation of fixed-size or projected systolic arrays.

We are working on a method without these drawbacks.^{70, 71} It specifies the control signals at the source level by a system of UREs called *uniform control recurrence equations* (UCREs). This makes the specification of control signals independent of the space-time mapping. By further exploiting properties of the space-time mapping, UCREs for arrays of any dimension and respecting any resource limitations can be obtained.

The minimisation of control hardware is still an unsolved problem.

8. Software Realisation

Systolic arrays can be viewed as synchronous programs: the index transformation yields recurrence equations over space-time¹¹ that can be coded in programming languages like ALPHA,^{22, 44} GAMMA^{2, 3} or Unity.¹⁰ But many commercial processor networks, like Warp¹ and transputer networks,²⁹ have asynchronous parallelism.

The parallelisation of for-loops has been an active area of research for many years.^{37, 67} Most of this work focuses more on the issue of concurrency than on the issue of communication because, predominantly, vector processors and other shared memory machines have been targeted.

Recently, work on the synthesis of inter-process communication directives for architectures with asynchronous parallelism has appeared – some of it in the setting of systolic arrays. The system SDEF¹⁶ takes a set of uniform recurrence equations and the index transformation matrix derived from it and fills a distributed program skeleton in C with the appropriate computations and communications. The program code is for a fixed problem size, i.e., the loop limits must be constants.

A systolic program for the algebraic path problem was generated with an extension of another mechanical systolisation method.³⁹ This method also takes the source program and the index transformation matrix as input. But it generates abstract distributed program code that is parameterised with the problem size and that can be manipulated further, e.g., optimised or translated to any target language. The initial version of this method works only for place functions that are projections along one axis of the index space; they correspond to parallelisations of one or more loops in the source program. A newer version works for all linear place functions.⁴

Other recent work on the synthesis of distributed programs is by Fencl and Huang,¹⁸ who target both synchronous and asynchronous architectures. Ribas⁵⁸ and Ramanujam and Sadayappan⁵⁶ generate more concrete and efficient program code. They address resource limitations by projection and partitioning (Sect. 6), but put more restrictions on the source. That is, they handle a smaller class of systolic arrays.

Current commercial programmable processor networks encourage coarse-grain parallelism. Ultimately, we expect continued progress in the development of distributed computer architectures that support finer-grain parallelism.³³

9. Theoretical Issues

9.1. Computability

A set of uniform recurrence equations is called *explicitly defined*³² or *computable*,⁵¹ if none of its variable instances depends on itself. The computability of uniform recurrence equations has been investigated widely;^{15, 32, 57, 62, 72} it can be reduced to the halting problem for a Turing machine and is, therefore, undecidable.³¹ The undecidability arises out of the possible unboundedness of the index space. For a fixed problem size (and, therefore, a finite index space), a set of UREs is computable, if its dependence graph is acyclic.

Karp et al.³² point out the decidability of strict UREs. In *strict* UREs, each equation depends directly on each of its arguments. Strict UREs are computable, if the reduced dependence graph does not contain non-positive cycles.

The strong point of systolic design is that the existence of a *linear* step function for a set of uniform recurrence equations can be established automatically.

9.2. Systolic Automata

Systolic automata are languages acceptors that are models of processor networks, usually in an array, trellis or tree-like form.^{24, 25} Various basic problems concerning systolic architectures and computations have been studied on such models; examples are the computational power of various interconnection patterns (e.g., two-way vs. one-way), different ways of interacting with the external environment and different types of heterogeneity. Problems of classical language theory have also been investigated.^{13, 17} Special attention has been given to the comparative study of the recognition power of classes of systolic automata with different underlying structure. Some restricted but useful design methods can be extracted from this theory¹⁴ and have been applied in the improvement of a systolic array for string correction.⁶¹

9.3. Semantic Independence

In the imperative style, the parallelisation is based in the concept of the semantic independence of two basic operations.²⁷ The independence criterion in systolic design, the absence of shared parallel accesses to variables, enforces full pipelining.

The imperative method can also be used for non-systolic parallelisations. In this setting, it is interesting to investigate the most general criterion for a semantic independence.⁶ A consequence of semantic independence is implementability on disjoint stores. One way of achieving a disjoint implementation is by transforming the pair of operations into an equivalent pair with disjoint variables. Curiously, since $x := z$ and $y := z$ are semantically independent, they are also implementable on disjoint stores. An according variable transformation is known,⁶ but general, constructive transformation techniques are not.

10. Acknowledgements

Thanks to Mike Barnett, Davide Sangiorgi and Jingling Xue for assistance. Financial support was received from the Science and Engineering Research Council, grant no. GR/G55457, and from the USSR Academy of Sciences.

11. References

1. M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. Lam, O. Menzilcioglu, and J. A. Webb, "The Warp Computer: Architecture, Implementation, and Performance", *IEEE Trans. on Computers* C-36, 12 (Dec. 1987) 1523–1538.
2. J.-P. Banâtre, A. Coutant, and D. Le Métayer, "A Parallel Machine for Multiset Transformation and its Programming Style", *Future Generation Computer Systems* 4, 2 (Sept. 1988) 133–144.

3. J.-P. Banâtre and D. Le Métayer, "The GAMMA Model and its Discipline of Programming", *Science of Computer Programming* **15**, 1 (Nov. 1990) 55–77.
4. M. Barnett and C. Lengauer, "A Systolizing Compilation Scheme: Abstract", in *Proc. 1991 Int. Conf. on Parallel Processing, Vol. II* (Pennsylvania State University Press, 1991) 305–306. Full paper: Technical Report TR-91-03, Department of Computer Sciences, The University of Texas at Austin (Jan. 1991).
5. A. Benaini and Y. Robert, "Spacetime-Minimal Systolic Architectures for Gaussian Elimination and the Algebraic Path Problem", in *Application Specific Array Processors*, eds. S. Y. Kung and E. E. Swartzlander (IEEE Computer Society Press, 1990) 746–757.
6. E. Best and C. Lengauer, "Semantic Independence", *Science of Computer Programming* **13**, 1 (Dec. 1989) 23–50.
7. J. Bu and E. F. Deprettere, "Converting Sequential Iterative Algorithms to Recurrent Equations for Automatic Design of Systolic Arrays", in *Proc. IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP 88), Vol. IV: VLSI; Spectral Estimation* (IEEE Press, 1988) 2025–2028.
8. J. Bu, E. F. Deprettere, and P. Dewilde, "A Design Methodology for Fixed-Size Systolic Arrays", in *Application Specific Array Processors*, eds. S. Y. Kung and E. E. Swartzlander (IEEE Computer Society Press, 1990) 591–602.
9. P. R. Cappello, "A Spacetime-Minimal Systolic Array for Matrix Product", in *Systolic Array Processors*, eds. J. V. McCanny, J. McWhirter, and E. E. Swartzlander (Prentice-Hall, 1989) 347–356.
10. K. M. Chandy and J. Misra, *Parallel Program Design* (Addison-Wesley, 1988).
11. M. C. Chen, "A Design Methodology for Synthesizing Parallel Algorithms and Architectures", *J. Parallel and Distributed Computing* **3**, 4 (1986) 461–491.
12. M. C. Chen, Y. Choo, and J. Li, "Compiling Parallel Programs by Optimizing Performance", *J. Supercomputing* **2** (1988) 171–207.
13. K. Culik II, J. Gruska, and A. Salomaa, "Systolic Automata for VLSI on Balanced Trees", *Acta Informatica* **18** (1983) 335–344.
14. K. Culik II, J. Gruska, and A. Salomaa, "Systolic Trellis Automata, Part II", *Int. J. Computer Math.* **16** (1984) 3–22.
15. J.-M. Delosme and I. C. F. Ipsen, "Efficient Systolic Arrays for the Solution of Toeplitz Systems", in *Systolic Arrays*, eds. W. Moore, A. McCabe, and R. Urquart (Adam Hilger, 1987) pages 37–46.
16. B. R. Engstrom and P. R. Cappello, "The SDEF Systolic Programming System", in *Concurrent Computations*, eds. B. W. Dickinson, S. K. Tewksbury and S. C. Schwartz (Plenum Press, 1987) Chap. 15.

17. E. Fachini and M. Napoli, "C-Tree Systolic Automata", *Theoretical Computer Science* **56** (1988) 155–186.
18. H. A. Fencil and C.-H. Huang, "On the Synthesis of Programs for Various Parallel Architectures", in *Proc. 1991 Int. Conf. on Parallel Processing* (Pennsylvania State University Press, 1991).
19. J. A. B. Fortes, K.-S. Fu, and B. W. Wah, "Systematic Design Approaches for Algorithmically Specified Systolic Arrays", in *Computer Architecture – Concepts and Systems*, ed. V. M. Milutinović (North-Holland, 1988) Chap. 11.
20. P. Frison, P. Gachet, and P. Quinton, "Designing Systolic Arrays with DIASTOL", in *VLSI Signal Processing, II*, eds. S.-Y. Kung, R. E. Owen, and J. G. Nash (IEEE Press, 1987) 93–105.
21. P. Gachet, B. Joinnault, and P. Quinton, "Synthesizing Systolic Arrays using DIASTOL", in *Systolic Arrays*, eds. W. Moore, A. McCabe, and R. Urquart (Adam Hilger, 1987) 25–36.
22. P. Gachet, C. Mauras, P. Quinton, and Y. Saouter, "A Language for the Design of Regular Parallel Algorithms", in *Hypercube and Distributed Computers*, eds. F. André and J. P. Verjus (North-Holland, 1989) 189–202.
23. M. R. Garey and D. S. Johnson, *Computers and Intractability* (W. H. Freeman & Co., 1979).
24. J. Gruska, "Systolic Automata: Power, Characterization, Nonhomogeneity", in *Proc. Mathematical Foundations of Computer Science (MFCS '84)*, eds. M. P. Chytil and V. Koubek, Lecture Notes in Computer Science 176 (Springer-Verlag, 1984) 32–49.
25. J. Gruska, "Synthesis, Structure and Power of Systolic Computations", *Theoretical Computer Science* **71** (1990) 47–77.
26. L. J. Guibas, H. T. Kung, and C. D. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms", in *Proc. Caltech Conf. on VLSI* (1979) 509–525.
27. C.-H. Huang and C. Lengauer, "The Derivation of Systolic Implementations of Programs", *Acta Informatica* **24**, 6 (Nov. 1987) 595–632.
28. C.-H. Huang and C. Lengauer, "An Incremental, Mechanical Development of Systolic Solutions to the Algebraic Path Problem", *Acta Informatica* **27**, 2 (Dec. 1989) 94–124.
29. INMOS Ltd., *Transputer Reference Manual* (Prentice-Hall, 1988).
30. H. V. Jagadish, S. K. Rao, and T. Kailath, "Array Architecture for Iterative Algorithms", *Proc. IEEE* **75**, 9 (Sept. 1975) 1304–1320.

31. B. Joinnault, *Conception d'Algorithmes et d'Architectures Systoliques*, PhD thesis, l'Université de Rennes I (Sept. 1987).
32. R. M. Karp, R. E. Miller, and S. Winograd, "The Organization of Computations for Uniform Recurrence Equations", *J. ACM* 14, 3 (July 1967) 563-590.
33. T. Kean and J. Gray, "Configurable Hardware: Two Case Studies of Micro-Grain Computation. *J. VLSI Signal Processing*, 2, 1 (Sept. 1990) 9-16.
34. H. T. Kung and C. E. Leiserson, "Algorithms for VLSI Processor Arrays", in *Introduction to VLSI Systems*, eds. C. Mead and L. Conway (Addison-Wesley, 1980) Chap. 8.3.
35. S. Y. Kung, *VLSI Processor Arrays* (Prentice-Hall Int., 1988).
36. S.-Y. Kung, S.-C. Lo, and P. S. Lewis, "Optimal Systolic Design for the Transitive Closure and Shortest Path Problems", *IEEE Trans. on Computers* C-36, 5 (May 1987) 603-614.
37. L. Lamport, "The Parallel Execution of DO Loops", *Comm. ACM* 17, 2 (Feb. 1974) 83-93.
38. P. Lee and Z. Kedem, "Synthesizing Linear-Array Algorithms from Nested For Loop Algorithms", *IEEE Trans. on Computers* C-37, 12 (Dec. 1988) 1578-1598.
39. C. Lengauer, M. Barnett, and D. G. Hudson, "Towards Systolizing Compilation", *Distributed Computing* 5, 1 (1991).
40. C. Lengauer and E. C. R. Hehner, "A Methodology for Programming with Concurrency: An Informal Presentation", *Science of Computer Programming* 2, 1 (Oct. 1982) 1-18.
41. C. Lengauer, B. Sabata, and F. Arman, "A Mechanically Derived Systolic Implementation of Pyramid Initialization", in *Proc. Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*, eds. G. Brown and M. Leeser, Lecture Notes in Computer Science 408 (Springer-Verlag, 1990) 90-105.
42. C. Lengauer and J. W. Sanders, "The Projection of Systolic Programs", *Formal Aspects of Computing* 2 (1990) 273-293.
43. C. Lengauer and J. Xue, "A Systolic Array for Pyramidal Algorithms", Technical Report ECS-LFCS-90-114, Department of Computer Science, University of Edinburgh (June 1990). To appear in *J. VLSI Signal Processing*.
44. H. Leverage, C. Mauras, and P. Quinton, "A Language-Oriented Approach to the Design of Systolic Chips", in *Int. Workshop on Algorithms and Parallel VLSI Architectures* (North-Holland, 1990).

45. C. Mauras, P. Quinton, S. Rajopadhye, and Y. Saouter, "Scheduling Affine Parameterized Recurrences by Means of Variable Dependent Timing Functions", in *Application Specific Array Processors*, eds. S. Y. Kung and E. E. Swartzlander (IEEE Computer Society Press, 1990) 100–110.
46. D. I. Moldovan, "On the Design of Algorithms for VLSI Systolic Arrays", *Proc. IEEE* **71**, 1 (Jan. 1983) 113–120.
47. D. I. Moldovan, "ADVISED: A Software Package for the Design of Systolic Arrays", *IEEE Trans. on Computer-Aided Design CAD-6*, 1 (Jan. 1987) 33–40.
48. D. I. Moldovan and J. A. B. Fortes, "Partitioning and Mapping Algorithms into Fixed-Size Systolic Arrays", *IEEE Trans. on Computers C-35*, 1 (Jan. 1986) 1–12.
49. C. Mongenet and G.-R. Perrin, "Synthesis of Systolic Arrays for Inductive Problems", in *Parallel Architectures and Languages Europe (PARLE '89)*, Vol. I: *Parallel Architectures*, eds. E. Odijk, M. Rem, and J.-C. Syre, Lecture Notes in Computer Science 365 (Springer-Verlag, 1989) 260–277.
50. P. Quinton, "The systematic design of systolic arrays", in *Automata Networks in Computer Science* (Princeton University Press, 1987) Chap. 9, 229–260.
51. P. Quinton and V. van Dongen, "The Mapping of Linear Recurrence Equations on Regular Arrays", *J. VLSI Signal Processing* **1**, 2 (Oct. 1989) 95–113.
52. S. V. Rajopadhye, "Synthesizing Systolic Arrays with Control Signals from Recurrence Equations", *Distributed Computing* **3** (1989) 88–105.
53. S. V. Rajopadhye, "Algebraic Transformations in Systolic Array Synthesis: A Case Study", in *Formal VLSI Specification and Synthesis (VLSI Design Methods-I)*, ed. L. J. M. Claesen (North-Holland, 1990) 361–370.
54. S. V. Rajopadhye and R. M. Fujimoto, "Synthesizing Systolic Arrays from Recurrence Equations", *Parallel Computing* **14**, 2 (June 1990) 163–189.
55. I. Ramakrishnan, D. Fussell, and A. Silberschatz, "Mapping Homogeneous Graphs on Linear Arrays", *IEEE Trans. on Computers C-35*, 3 (Mar. 1986) 189–209.
56. J. Ramanujam and P. Sadayappan, "A Methodology for Parallelizing Programs for Multicomputers and Complex Memory Multiprocessors", in *Supercomputing '89* (ACM Press, 1989) 637–646.
57. S. K. Rao and T. Kailath, "Regular Iterative Algorithms and their Implementations on Processor Arrays", *Proc. IEEE* **76**, 3 (Mar. 1988) 259–282.
58. H. B. Ribas, *Automatic Generation of Systolic Programs from Nested Loops*, PhD Thesis, Technical Report CMU-CS-90-143, Department of Computer Science, Carnegie-Mellon University (June 1990).

59. Y. Robert and D. Trystram, "An Orthogonal Systolic Array for the Algebraic Path Problem", *Computing* **39**, 3 (1987) 187–199.
60. G. Rote, "A Systolic Array Algorithm for the Algebraic Path Problem (Shortest Paths; Matrix Inversion)", *Computing* **34**, 3 (1985) 191–219.
61. D. Sangiorgi and C. Lengauer, "An Improved Systolic Array for String Correction", Technical Report ECS-LFCS-91-153, Department of Computer Science, University of Edinburgh (May 1991). To appear in *Proc. Advanced Research Workshop on Correct Hardware Design Methodologies* (June 1991).
62. Y. Saouter and P. Quinton, "Computability of Recurrence Equations", Technical Report 1203, INRIA-Rennes (Apr. 1990). To appear in *Theoretical Computer Science* **97** (Aug. 1992).
63. C. J. Scheiman and P. R. Cappello, "A Processor-Time Minimal Systolic Array for Transitive Closure", in *Application Specific Array Processors*, eds. S. Y. Kung and E. E. Swartzlander (IEEE Computer Society Press, 1990) 19–30.
64. A. Schrijver, *Theory of Linear and Integer Programming*, Series in Discrete Mathematics (John Wiley & Sons, 1986).
65. U. Schwiegelsohn and L. Thiele, "A Systolic Array for Cyclic-by-Rows Jacobi Algorithms", *J. Parallel and Distributed Computing* **4** (1987) 334–340.
66. V. van Dongen and M. Petit, "PRESAGE: A Tool for the Parallelization of Nested Loop Programs", in *Formal VLSI Specification and Synthesis (VLSI Design Methods-I)*, ed. L. J. M. Claesen (North-Holland, 1990) 341–359.
67. M. Wolfe, *Optimizing Supercompilers for Supercomputers* (MIT Press, 1989).
68. Y. Wong and J. M. Delosme, "Optimal Systolic Implementations of N-Dimensional Recurrences", in *Proc. IEEE Int. Conf. on Computer Design (ICCD-85)* (IEEE Press, 1985) 618–621.
69. J. Xue and C. Lengauer, "On One-Dimensional Systolic Arrays", in *Proc. ACM Int. Workshop on Formal Methods in VLSI Design*, (Springer-Verlag, 1991). To appear.
70. J. Xue and C. Lengauer, "The Synthesis of Control Signals for One-Dimensional Systolic Arrays", in *Algorithms and Parallel VLSI Architectures II*, eds. P. Quinton and Y. Robert (North-Holland, 1991). To appear.
71. J. Xue and C. Lengauer, "The Systematic Derivation of Control Signals for Systolic Arrays", Technical Report ECS-LFCS-91-152, Department of Computer Science, University of Edinburgh (May 1991).
72. Y. Yaacoby and P. R. Cappello, "Scheduling a System of Nonsingular Affine Recurrence Equations onto a Processor Array", *J. VLSI Signal Processing* **1**, 2 (1989) 115–125.

HIGH PERFORMANCE COMPUTER SYSTEM "ELECTRONICA SSBIS"

V.A.Melnikov, Ju.I.Mitropolski, V.Z.Shnitman,
V.P.Ivannikov, A.N.Tomilin, S.S.Gajsarian
The Institute for Problems of Cybernetics
USSR Acad. Sci.
Vavilova st., 37, Moscow, USSR

ABSTRACT

The paper describes basic development concepts of hardware architecture and software of supercomputer "Electronica SSBIS".

1. Hardware Architecture

The design of computer systems "Electronica SSBIS" is based on the totality of works concerning creation of constructive component base that allows to integrate into a united construction a large volume of high frequency circuits and on advancing the idea of constructing a system with open architecture including problem-oriented and function-specialized subsystems.

Problem-oriented computers and subsystems provide the highest performance when solving large problems. Vector machines, for example, are considered as related to this kind. Function-specialized subsystems are: external memory subsystems (an integrated circuits and magnetic disks), data base subsystems based on them, subsystems of communication with terminal hardware, graphical subsystems, as well as hard-software of operating system, programming system and so on.

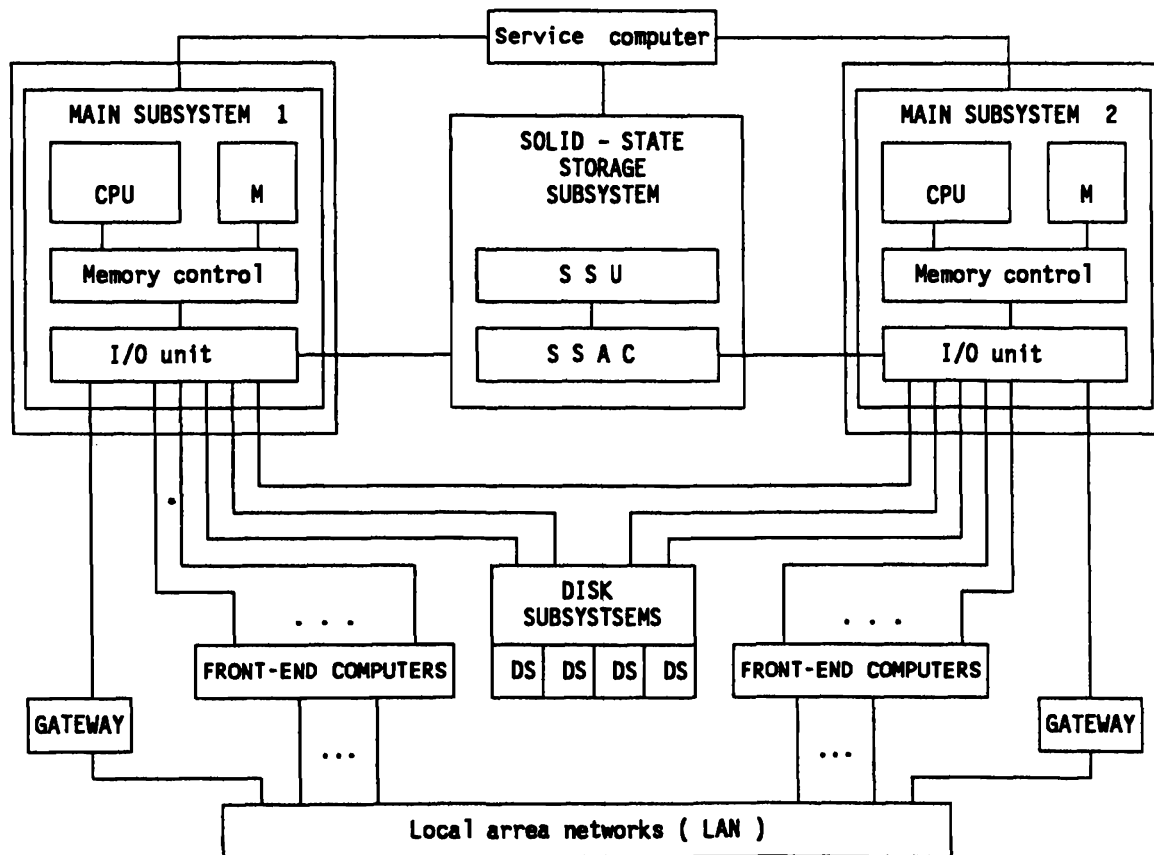
Integration of these systems into a united system requires hierarchical structure of communication channels; each of the channels has to take account of characteristics of united abonents and a possibilities of further growth of subsystem and changes in its configuration.

The main devices of the system "Electronica SSBIS" are high-capacity vector pipelined host machines, a subsystem of external semi-conductor memory of large volume, disk subsystems, as well as an interface devices with front-end and control machines and tools for construction of local area computer networks. The system can work in the mode of batch processing, in the mode of time-sharing and in the real time mode. A user can choose the optimal from his point of view system configuration with given characteristics: performance, size of main and external semi-conductor

memory, structure and size of mass memory on magnetic disks, number and nomenclature of front-end machines and so on.

In the fig.1 the basic configuration of the computer system "Electronica SSBIS" is shown, that includes:

- two vector pipelined host machines, each one having local main memory of capacity 8-32 Mb and performance up to 250 Mflops (floating point operations per second);
- a subsystem of external semi-conductor memory of capacity 256 Mb, having a two-port controller for connecting two host machines;
- from 2 to 8 disk subsystems of total capacity more than 20 Gb;
- a subsystem of front-end machines for work with peripheral hardware and for preparing tasks for host machines;
- a subsystem of control machines for control and diagnostic functions for the whole system;
- programmable devices of access, intended for building a local area network that ensures the work with workstations and front-end user computers.



M - Main memory
SSU - Solid-state storage unit
SSAC - Solid-state storage access controller
DC - Disk server

Fig. 1. "ELECTRONICA SSBIS" architecture

The high performance of host machines is received by extensive use of the vector pipelined processing principle with short synchronization time, high-speed register memory of large size and great number of special functional devices ensuring parallel data processing, integration in the mainframe of one computer of the advantages of scalar and vector processing.

In the host machines there is a two-level buffer addressable memory for scalar and address operands, synchronous pipeline functional devices allowing to receive one result in every device's output at each synchronization time (there are also two complete sets of seven devices for 64-bit scalar and vector processing and two devices for processing 24-bit address information), a command buffer of size 256 64-bit words. In the machines the mode of chaining vector operations is implemented, so in separate program sections two or three results can be received at each time. The maximum throughput of command pipelines is reached by using more registers and by possible optimization of programs in compilers from Fortran 77, Pascal and C languages.

The system external memory has two hierarchical levels. The subsystem of external semi-conductor memory of capacity 256 Mb has such parameters as size, access time and access speed in the average between operative memory and mass memory on magnetic disks. The use of external semi-conductor memory system in the host machines allows to reduce considerable the file input/output time and to ensure effective execution of large programs with intensive data communication.

The external semi-conductor memory controller has a special processor that ensures data communications between it and the main memory forming external memory addresses at the rate of communication with a program that implements the required access method. By a directive coming from the host machine the processor can be tuned practically to any access method required by an application. That increases the total performance of the work of the system, since it releases the processors of host machines from routine work, decreases the number of information communicated with external memory, for data flow between the host machine and the mass memory will contain only useful information.

Those that gain most of all from the use of mass memory are the tasks processing large arrays of information with arbitrary access to their items.

Containing double complete set of high speed synchronous input/output channels the controller allows to connect up to it two system host machines. The rate of communication through the channel is 150 Mb per second.

The next hierarchy level of external memory contains disk subsystems. There are two ways to implement the disk subsystems.

In the first way every disk subsystem is built on the basis of a special magnetic disc controller (MDC). Up to eight magnetic disc devices of capacity 317.5 Mb can be connected up to each MDC. The magnetic

disc controller ensures communication between disk devices and the main memory of the host machine through a couple of synchronous input/output channels. Depending on configuration from 1 to 4 MDC (or from 2 to 32 devices) can be connected up to one host machine.

In the second way every disk subsystem is built on the basis of two disk servers. A disk server of the capacity 4.5 Gb is implemented on the basis of the workstation "BESTA". Disk servers are adjoined directly to the asynchronous input/output channels of the host machine. Depending of configuration from 2 to 8 disk servers can be connected up to one host machine.

The work of the computer system "Electronica SSBIS" is organized for using it in a common network with front-end machines.

Front-end machines are independent computers that work under control of their own operation systems. Front-end machines are connected up to asynchronous channels of the system host machines using corresponding interface devices or programmable access devices. The interface device agree differences in width of channels and length of machine words, logical levels and control items. Since asynchronous channels of host machines have higher communication speed than that of channel (interface) of any front-end machine, the maximum communication speed is limited by the maximum channel speed of front-end machines. At programmable access devices the functions of local area network transport station are implemented.

A front-end machine adjoined to the computer system "Electronica SSBIS" can function as: a console, a local station for input/output of batched tasks, a data concentrator that multiplexes input from several workstations to one channel of the host machine, a remote station for input/output of batched tasks, an interactive communication station and so on. The tools of connecting computers of kind "Electronica 79", "Electronica 82", "SM", "BESM-6", the computer families "Elbrus" and "BESTA" -as front-end machines are developed. The connection with local area computer networks are performed by front-end machines.

Advanced requirement to the reliability of the system, impossibility to implement by hand complicated diagnostic algorithms makes necessary automatization of check and search of faults in different components of the system. For this a subsystem of control, check and diagnostic is implemented in the system; one of its main functions is to control diagnostic and check modes, to indicate the state of processors, main and external memories, channels etc. The subsystem ensures early detection and localization of a fault to within one or several packages SIS or BIS by comparing the state of checked hardware with standard information stored in the memory of the system control machines. As system control machines computers "Electronica 79" or workstations of the family "BESTA" can be used.

2. System Software

The architecture of the supercomputer complex based on the totality of function-specialized computer tools for preparing, transmitting and processing data requires the software for complex components and for the whole complex to be adaptive for connecting to the complex different hardware tools and for inserting supercomputers into different large computer centers.

According to this requirement, the system software of the supercomputer complex is to ensure the work of its components in a local area computer network that contains single- or multiprocessor host machines proposed for executing a large volume of computations and front-end computers used for preparing data and transmitting them to computation on host machines and for receiving back results of computations and giving them to output devices and also for working together with host machines in information processing systems including real-time ones. The work of the sequential-parallel information processing system - the computer pipeline can be organized with any combination of host and front-end machines in the pipeline, that is mostly important for creating real-time systems.

The considered requirements lead to decentralized architecture of the system software, where independent operative systems of different functional components are completed by means of working inside the system local area network. These means include a network transport service that ensures information exchange among processes (their communication is of the kind "remote procedure call") in different system computers and services that implement functional protocols of the network-protocols of input of tasks and output of computation results, protocols of file transfer, operator system service. User distributed systems can be created above the network transport service, for instance, the distributed system of information retrieval, systems of computer-aided engineering, systems of real-time information processing.

Host machines of the supercomputer complex have an advanced operative system, that, besides common operative system functions, executes data control on two-level external memory, supports transport and functional network protocols.

The main characteristics of vector pipelined computers of high performance - means of vector data processing, multilevel organization of processor registers, concurrently working pipeline devices - require that optimization compilers from high-level languages were constructed, that allow effective implementation. The availability of tools profiling the program behavior will help to reach the maximum possible speed of computation. The host machines have direct compilers from high-level languages.

Since there are several host machines in the supercomputer complex, compilation of programs can be charged on the "monitor machine" connected with the host machines by shared semi-conductor mass memory. For the same processor as in the host machine is used in the "monitor machine", a user task can be executed on it as well.

In any supercomputer complex structure a cross-compiler on a front-end computer can be used.

The operative environment of a computer system is the set of soft/hardware tools that ensures execution of user programs. The operative environment is defined by a set of notions - objects - and tools for manipulating them. One of the main requirements is the requirement of homogeneity of the whole operative environment, i.e. the requirement of independence of a user that wants to execute, edit or debug his program on the front-end machine by which he performs these actions. The basic object of the operative environment is a user task. A task is divided into steps. The totality of actions that are executed on one system computer is called generalized job step. Generalized steps can be executed either sequentially or concurrently. A generalized job step in its turn is divided into steps with fixed execution order (sequential or concurrent). The second main object of the operative environment is the unit of data organization - a file. The information environment of each task step is formed by files. Files are stored in a hierarchically ordered file archive on external memory.

The operating system architecture is chosen depending on its function destination, the architecture of the complex and the environment in which the OS is to work. The characteristics of the hardware of the supercomputer complex and the work modes - batched or real-time determine the choice of basic objects of the operating system - tasks (user, system) to be executed in separate address spaces of host machines of the supercomputer complex and messages as tools for communications and synchronization of tasks. The OS kernel transmits messages and controls execution of tasks.

Though the set of OS basic objects is not the same as the set of the homogeneous operating system objects, it is rather close to it and the ordinary reflection from one to another is possible.

The operating system consists of a kernel and a set of system tasks. Some system tasks that are called pseudo-tasks work in the privileged mode, can access to the whole operative memory, have no virtual complete set of basic and intermediate registers. The tasks of control of the operative memory, of the external memory and of the transport stations are designed as pseudo-tasks. Pseudo-tasks communicate with OS tasks using the standard tool of exchange of messages among tasks.

The tool of exchange of messages among tasks implements the primitives: "to send a message synchronously", "to send a message asynchronously", "to send an answer to a message", "to wait for a message", "to wait for an answer to a message", "to wait for the end of a time interval" and primitives with different combinations of such instructions.

The central OS task is "initiator-terminator" that executes the following functions:

- forming new user tasks and system not-resident OS tasks;
- informing all the system tasks about appearance of a new task; in

consequence of that all system tasks are tuned to the work with the new abonent;

- normal or abnormal termination of a task about which all the system tasks know, in consequence of that they are tuned out of the abonent;

- scheduling the execution of input jobs and job steps;
- solving the collisions when lack of resources and system deadlocks take places;

- managing "control points".

Jobs are prepared on front-end machines. A job consists of a set of data collections. The obligatory data collection in a job is a main program in the job-control language.

A job is stored into the OS file system. The scheduler of the initiator-terminator chooses a task for execution. The job consists of a sequence of steps. To execute each step a task is created. When job steps are executed, output data are stored in output files of the job. The initiator-terminator passes these files to the system task of output. After the job is terminated the system task of output passes them to a front-end machine.

When executing a job step the initiator-terminator may receive from it a message - demand to create a control point. The initiator-terminator will inform all the system tasks that a control point is created for a given task. In reply the initiator-terminator will get information according to which the task execution can recommence afterwards. The initiator-terminator stores this information in the "log" of control points where the information on the state of registers and memory of the task are also stored. After a crash the initiator-terminator executes the inverse operations and so the computation of the task can renew from its last control point.

When several tasks are working they can get into a deadlock because of sharing files or lack of mass memory. The system "Archive" task detects the tasks that have got into a deadlock and informs the initiator-terminator about that. The initiator-terminator taking into consideration the priorities of tasks and occupied by them resources excludes one of the tasks that are executed and so solves the deadlock. The excluded task rolls back to its last control point from which its execution will renew later.

The sequence of task steps is described in the job- control language.

The job-control language is a command programming language that connects the task with the OS.

The job-control language serves to describe the tasks where computations that use both several task steps and several operations allowed by one task step are performed. A task step is a multi-enter program specially prepared by the loader and executed as a task on a host machine. When several host machines share semi-conductor memory processes (steps) of one job can be executed on them.

The JCL contains tools for data exchange between user task steps

and the control program through input and output parameters of job steps. The availability of loop and conditional statements in the JCL allows users to create compact and flexible programs for control of computation. The JCL contains constructions allowing to process exceptions in the control program and to generate exceptions in it. The user can set the order of looking the file system directory when searching job steps. Catalogized procedures stored in the file system can be written in the JCL.

To ensure flexible construction and up-date of the operative environment of the executed program mechanism of abstraction is implemented in the programming system using abstract objects "clusters".

A cluster is determined by the specification level and by the representation level. On the specification level a cluster is represented by the operation set with a specified interface. On the representation level of a cluster local data structures, procedures implementing operations, other abstract objects are described. A cluster is accessible for an abstract program (a program that uses an abstract object), only on the specification level, the representation level is hidden from it.

The cluster is generated by a declaration construction where the name and the type of an instance are specified. The type of a cluster determines the information used when generating the cluster: the description of local memory; the procedures implementing operations over objects; the declaration of local clusters.

There are equivalent types of cluster. Equivalent types are types that generate clusters with the same specification level, i.e. with equal names of operations but different representation levels.

The totality of clusters determines the operative environment of a program execution.

The cluster is generated at the step of loading (assembling a program from object modules).

An abstract program may contain only constructions of calling clusters; and instances may be declared in other object modules (including extracted from the library of object modules) that will come to the input of the loader together with the given program that is also prepared in the form of an object module.

It is possible to declare an instance of a cluster in the job-control language too. Given declarations are transformed into object modules that contain constructions for generation of instances that are passed to the input of the loader together with the main program.

When assembling a program a component of the job-control language - the step interpreter is loaded into its space; it can access to all declared entry points of the program represented as a binary code.

After loading a job step into the main memory the initiator-terminator hands over the control to the step interpreter. The step interpreter will interpret the task control file in the inner representation that determines the order of calling to entry points of the given step. The common sequence of calls consists of the initiation of clusters (preparation of the operative

environment for execution) and of starting the program itself. When the step interpreter meets in the file a call to another step it calls the initiator-terminator and indicates to it the next step and offset in the file of the inner representation.

The file system objects are stored in the two-level external memory (semi-conductor mass memory and disk memory).

The files stored in archives of the host machines are accessed by interaction with the software of the host machines.

The OS file system consists of a system task "Archive", a pseudo-task "Communication" (the task "Communication" is implemented as a pseudo-task because the privileged mode of program work is required to arrange communications through channels), clusters of file access methods that are executed in address spaces of user tasks.

The part of the file system is located in the user task in order to decrease the overhead of access to a separate record of a file. If the cluster were outside the user task it would lead to the task switching every time when the file is accessed.

The well-known OS (RSX-11M, UNIX) are used in front-end machines added by drivers of communication channels with host machines and programs of transport and functional services. For front-end machines an object-oriented cluster operating system (CLOS) is designed as well. The CLOS consists of communicating clusters and subsystems (sets of logically interconnected clusters). User tasks can be build in the same way. Emulation of the OS UNIX user interface to the CLOS is also possible.

The cluster technology that ensures the protection of clusters one from another, facilitates the creation of a program system with asynchronously working branches and of distributed program systems is the common approach to building large program systems including operating system.

The basic programming system supports programming in the macro assembler language. The macro assembler language besides common means contains a mechanism of abstract data types that allows to make the most of modular programming when developing complicated software systems.

The programming systems of the Fortran 77, Pascal and C languages possess means for program optimization that take into account the features of vector architecture (the loop vectorization and the scheduling of the control flow).

To supply the user with tools of effective solving his problems that bring the values of performance close to the maximum, the supercomputer complex is equipped with the "Scientific Library" - an extensible set of libraries and routines that embrace solutions of the most widespread problems of computation and discrete mathematics.

The simulation complex of the host machine implemented on instrumental computers (including IBM PC) is designed for developing and debugging the software of the host machine processor, the cross-assembler, the cross loader, the cross debugger.

VERTICAL PROCESSING IN PARALLEL COMPUTING SYSTEMS

W. Händler

Institut für Mathematische Maschinen
und Datenverarbeitung (Informatik),
Universität Erlangen-Nürnberg, 8520,
Erlangen, FRGermany

and

Ya.I. Fet

Computing Center, Siberian Division
of the USSR Academy of Sciences,
630090, Novosibirsk, USSR

Abstract

This paper describes some investigations of finegrain SIMD-systems with bit-sequential processing, which we propose to call Vertical Processing Systems (VPS). Several hardware solutions for development of simple efficient VPS's are suggested. A method is described of designing VPS from standard von-Neumann computers. It is proposed to utilize specialized devices based on vertical and distributed processing as accelerators for VPS's. Examples of such coprocessors are given.

1. Introduction

Between the data structures subject to processing in computing systems the most widespread seems to be rectangular arrays. In the computers of traditional von-Neumann architecture the basic operational unit is a parallel adder, and the whole processing is made by means of this adder in a bit-parallel word-serial manner (i.e. sequential by the rows of the array). Such processing is called *horizontal processing*. The contemporary parallel computing systems use a large number of concurrently working computers, but each of them accomplishes the classical horizontal processing.

An important improvement of efficiency can be achieved by using a different processing mode: word-parallel bit-sequential (i.e. sequential by the columns of the array). This is the *Vertical Processing* (VP). There are several reasons making VP highly attractive. First of all, it permits to realize a very large degree of parallelism. The vertical approach is quite natural in many problems (for instance in solving information retrieval problems). In other cases it seems unusual and a search for a new view on the problem is required. But, as shown by practice, efficient "vertical" algorithms can be constructed for a broad range of applications.

It is usually considered that the fastest processing can be reached when the structure of the operational unit corresponds as much as possible to the structure of the input data and the applied algorithm. From this point of view, the best solution seems to be to superpose the memory with logic and to produce the result by means of some "global" process performed on the argument arrays, which is called *distributed processing*. This idea is just the basis of the well known content-addressed memories and associative processors with distributed logic proposed as early as in 50ties. Unfortunately, the task of implementing content-addressed memories of sizes sufficiently large for practical use remained too complex over a long period of time.¹ In view of these difficulties, vertical processing is recently of growing importance.

The correlation between distributed, horizontal and vertical modes of processing is made transparent by the conception of compression and extension of logic circuits. The schematic view of a distributed structure is shown in Fig. 1a. Imagine that the memory is separated from the distributed Operational Unit (OU), and then the latter is compressed in the vertical direction. Thus, we get the usual horizontal processing scheme with horizontal memory access (Fig. 1b). If the OU is compressed in horizontal direction, we get the vertical processing scheme with vertical memory access (Fig. 1c). Transformations of this kind were introduced by F.Hennie.²

Thus, the vertical processing represents a happy compromise. On the one hand, the vertical OU's are quite within the reach of the present technology, on the other hand, they ensure parallel processing of

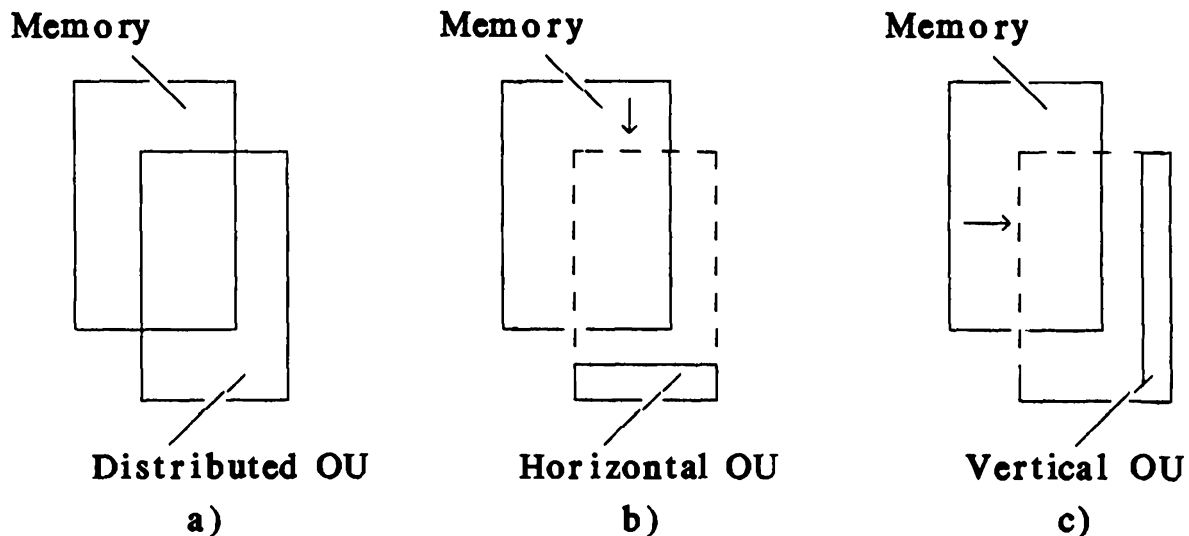


Fig. 1.

sufficiently large fragments of argument arrays. For example, an array of 2^{12} 32-bit words has to be processed and the parallelism of the vertical OU is equal to 2^{12} , the whole processing requires only 32 time steps, i.e. it can be accomplished 2^7 times faster compared to the horizontal approach.

2. A short history of vertical processing

Some ideas concerning VP are connected with the development of associative parallel processors. During the 60ies various methods for solving massive numeric and logical problems in associative memories were intensively investigated.^{3,4} In these methods a technique has been used of consecutive selection (by means of appropriate masks) of isolated bit-slices of the argument array and analysing the corresponding contents of the response register. The ideas contained in this algorithms are related to VP.

To implement associative processors, it was suggested to use bit-slice sequential "quasi-associative processing". To ensure the needed access mode, different types of memory devices have been considered: rotating memories with parallel read/write - so called "head-per-track systems",^{5,6} shift registers,⁷ "orthogonal" memories,⁸ etc. A comprehensive review of quasi-associative processors can be found in the book.⁹

The notion of VP seems to have been first explicitly defined by W.Shooman,¹⁰ who also introduced the terms *vertical processor* and *orthogonal processor*. The orthogonal processor developed by W.Shooman contains two OU's working upon a common orthogonal memory. This memory ensures two access modes: the usual horizontal processor exploits it as a storage with an usual random access to n -bit words, and the special vertical processor as a storage with random access to m -bit bit-slices. The vertical processor has several m -bit general purpose registers. The VP as such is carried out by a function generator consisting of m identical logical circuits, each of which is capable to implement any of the 16 boolean functions of two variables as well as the functions of a full adder.

In the middle of the 60ies a computer family *OMEN-60* based upon Shooman's project was designed by *Sanders Associates*.¹¹ This family may be considered as the first commercially available computers with vertical processing.

3. Orthogonal Computation by Conventional Equipment

The decisive problem in organizing VP is the memory construction. A simple solution of this problem has been suggested in¹² and described in detail in papers.^{13,14}

The central idea allowing both horizontal and vertical operations in one *Horizontal-Vertical Computer (H-V-Computer)* is the dual arrangement of data in the usual main store, Fig.2. The interface between the CPU and the main store itself is conventional, and the instructions are stored conventionally in horizontal words. Only the data written into the main store can be stored horizontally, in which case a memory register contains one data word (n bits), or vertically, in which case a memory register contains a bit-slice, i.e. one bit out of each of m different data words. Thus, orthogonal processing is achieved by two-dimensional arrangement of data in memory, and not by two-dimensional interface to memory. The conversion of incoming horizontal data to the vertical form and vice versa can be accomplished by means of a simple dedicated hardware device, so called *turn-table*, usually connected as an input/output device.

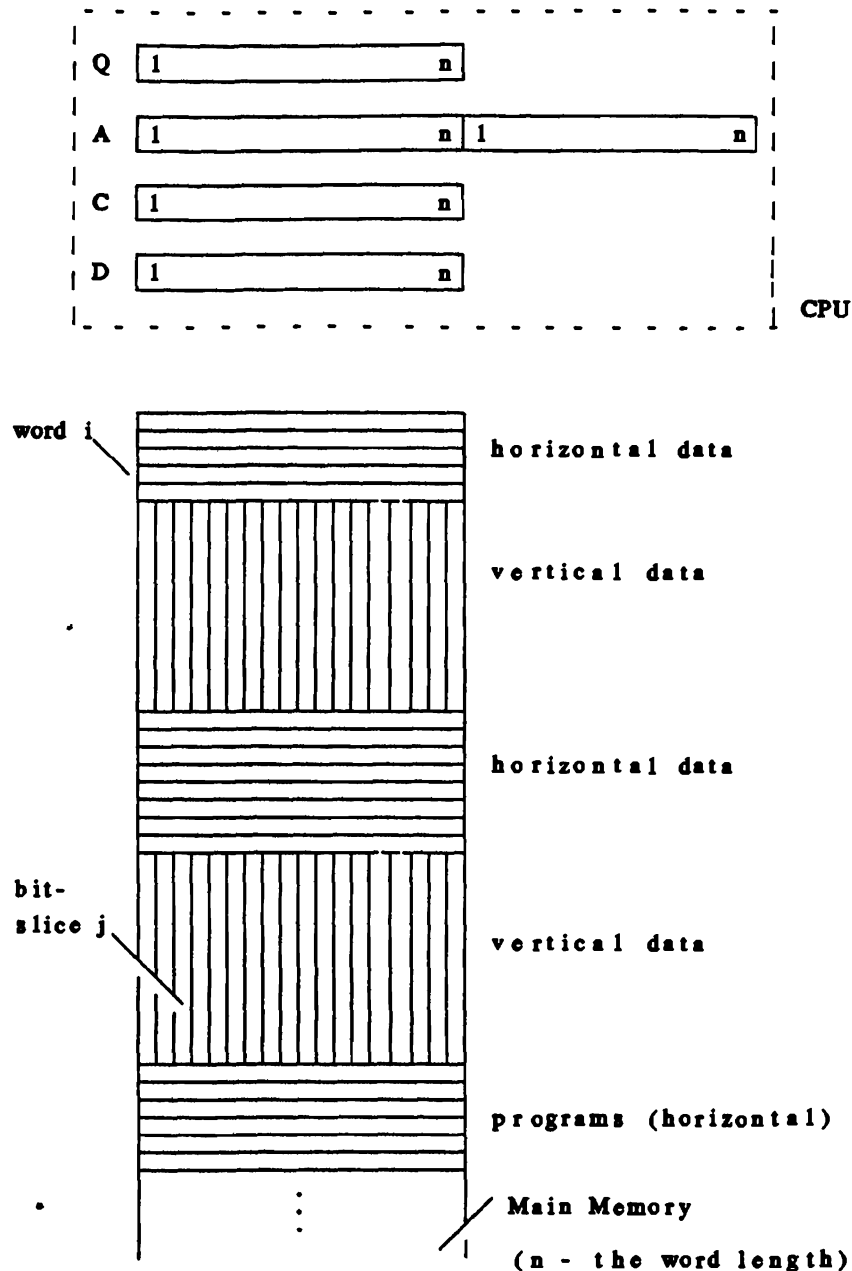


Fig.2.

The main advantages of the H-V concept are:

- A single H-V-Computer replaces a conventional General-Purpose Processor (GPP) and a vertical Associative Array Processor (AAP) coupled to it. The costs are only slightly higher than for a GPP.
- The risk involved in the investment in a special AAP, which will be usually not fully loaded, is avoided.
- The available main store can be partitioned arbitrary between areas used for: GPP programs (horizontal), GPP data (horizontal), AAP programs (horizontal), AAP data (vertical).

- The programming of the vertical mode does not differ too much from the programming of the GPP, which makes it considerable easier.
- In programs which contain parts in both horizontal and vertical modes, the mode can be changed when required without delegation of parts of the program to another processor.

4. Vertical Processing Systems

An important step in development and practical implementation of VP was made by K. Batcher who suggested the *MultiDimensional Access (MDA) memory*.¹⁵ This memory is built of standard RAM chips, but during the read/write operations a dedicated hardware unit generates particular chip numbers and corresponding addresses (so called *scewing scheme* of data storage). This technique ensures horizontal and vertical accesses as well as several other access modes ("mosaic" data formats).

It should be noted that the problem of conflictfree access in parallel memories have been intensively investigated in recent years. Interesting results in this field are contained in.^{16,17}

On the basis of Batcher's MDA, *Goodyear Aerospace Corp.* since 1972 successfully developed the line of *STARAN* processors.¹⁸ In early versions of *STARAN* each Associative Module (AM) consisted of a MDA memory matrix sized 256 x 256 (65K bit) and a vertical OU of 256 simple Processing Elements (PE). Subsequently the capacity of each AM was enlarged. The system permits the concurrent operation of at most 32 AM's, thus enabling the maximum parallelism of 8192 (256 x 32) PE's.

The main reason of VP is just the high degree of parallelism, which allows to get superhigh performance of the whole system, though each word of the input data is processed bit-sequential, i.e. comparatively slowly.

Traditionally *STARAN* is called AAP. However, it would be more exact to classify it as a special category of the SIMD-architecture, which is characterized by the bit-sequential operation mode and very large number of simple PE's, each of which is connected by means of

one-bit-wise bus with its own local memory. We call such finegrain SIMD-architectures *Vertical Processing Systems (VPS)*. Besides *STARAN*, to this category belong also computers *DAP*,¹⁹ *MPP*,²⁰ *CM*.²¹ Unlike *STARAN*'s MDA memory, these systems exploit local RAM's with simple linear addressing of bits, but with vertical arrangement of data. Table 1 shows the main characteristics of contemporary VPS's.

Table 1

System model	Manufacturer	Number of PEs	Intercon. Network	Local Mem., Kbits	Memory Size, Mbytes
DAP-510	Active Memory Techn.	1024 (32x32)	NEWS	32	4
DAP-610	---	4096 (64x64)	NEWS	64	32
MPP	Local Defence Systems	16384 (128x128)	NEWS	64	128
CM-1	Thinking Machines Corp.	65536	Hypercube	4	32
CM-2	---	65536	Hypercube	64	512
STARAN	Goodyear Aerospace Corp.	up to 8192	Flip	9	up to 70

The important features of VPS are the structure and the throughput of interconnection network used for communications between the PE's

because these features are often the decisive factor influencing the system performance. The *STARAN* system exploits a FLIP network, which is a dedicated multistage network implementing a broad set of "butterfly"-type permutations and batch shifts. In *DAP* and *MPP* only four nearest-neighbour connections are provided forming so called NEWS (North-East-West-South) network. In the new *BLITZEN* system²² which represents a development of *MPP* designed especially for VLSI implementation, the PE's form so called X-grid, a two-dimensional network with interconnections in eight compass directions: N, NE, E, SE, S, SW, W, NW.

The most sophisticated routing network is used in the *CM* system. Each chip, consisting of 16 PE's, has his own router. The routers of all chips are connected into a 12-dimensional hypercube. This structure enables communications between arbitrary PE's by means of message transfer through intermediate nodes. Besides the hypercube network, *CM-1* has a NEWS-type grid which accelerates local communications. *CM-2* realizes various grid structures by means of program reconfiguration of the hypercube.

In recent years the problem of interprocessor communications in parallel computing systems has grown a special important branch of computer science. A lot of books and papers is devoted to research in this field. A good survey of these can be found in.²³ It should be emphasized that in most interconnection networks the data are transferred in each channel in a bit-serial manner, that is the rectangular arrays are transferred columnwise. This is also to some degree a vertical processing, which garmonizes well with vertical storage and vertical computations adopted in VPS.

The interconnection network defines the topology of any parallel computing system. Thus, the PE's of *STARAN* make up a linear (one-dimensional) structure, PE's of *DAP* and *MPP* two-dimensional structures and *CM* a multi-dimensional one.

5. A Memory-Coupled Multiprocessor

It has been shown above how a linear VPS can be organized on the basis of a standard von-Neumann computer. Elaborating this idea, we can introduce the second dimension by linear connection of a set of standard processors made for instance via a multiport storage.

To simplify the description we assume an existing structure of Erlangen Multiprocessor, which was built up and investigated at the IMMD and basically corresponds to the projects *EGPA* and *DIRMU*.

An Erlangen Multiprocessor is composed of unit-elements consisting of a processor and a memory. The memory is accessible via a multiport switch. There is a total of 8 connections to the multiport, of which one is reserved to the "local" processor. The unit-element may be therefore considered as a Processor-Memory-element (PM) connectable to at most 7 other PM's. For example, two such PM's can be connected as shown in Fig.3. In other words, the possible topologies are identical to graphs of maximal degree 7.

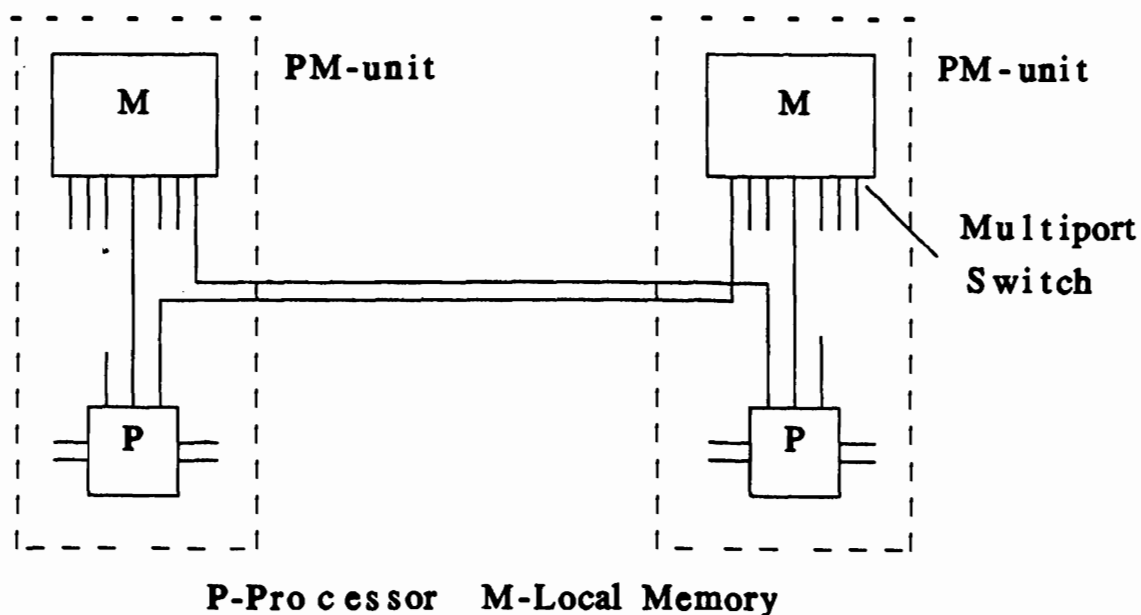


Fig.3.

Suppose that each processor is working on 32 bits and 32 such processors are combined to a multiprocessor in the described manner. Thus, a two-dimensional array (32 x 32) is composed of the 32-bit

machine words of the coupled processors, on the one hand, and of 32 processors in the nearest-neighbour topology, on the other hand.

An extended picture of the array is shown in Fig.4, where the North-South direction is stretched to make clear how the processors are composed. Each line of 32 bits represent one PM-unit. As can be seen, this structure is similar to *DAP*. Considering the nearest-neighbour (NN) relation, the two dimensions are asymmetrical. One dimension (say East-West) corresponds to the machine word of the PM. NN-values in this dimension are taken by a simple shift, left or right. The other dimension (then North-South) corresponds to the coupled PM's. NN-values in this direction are accessed from the neighbouring PM's via a multiport.

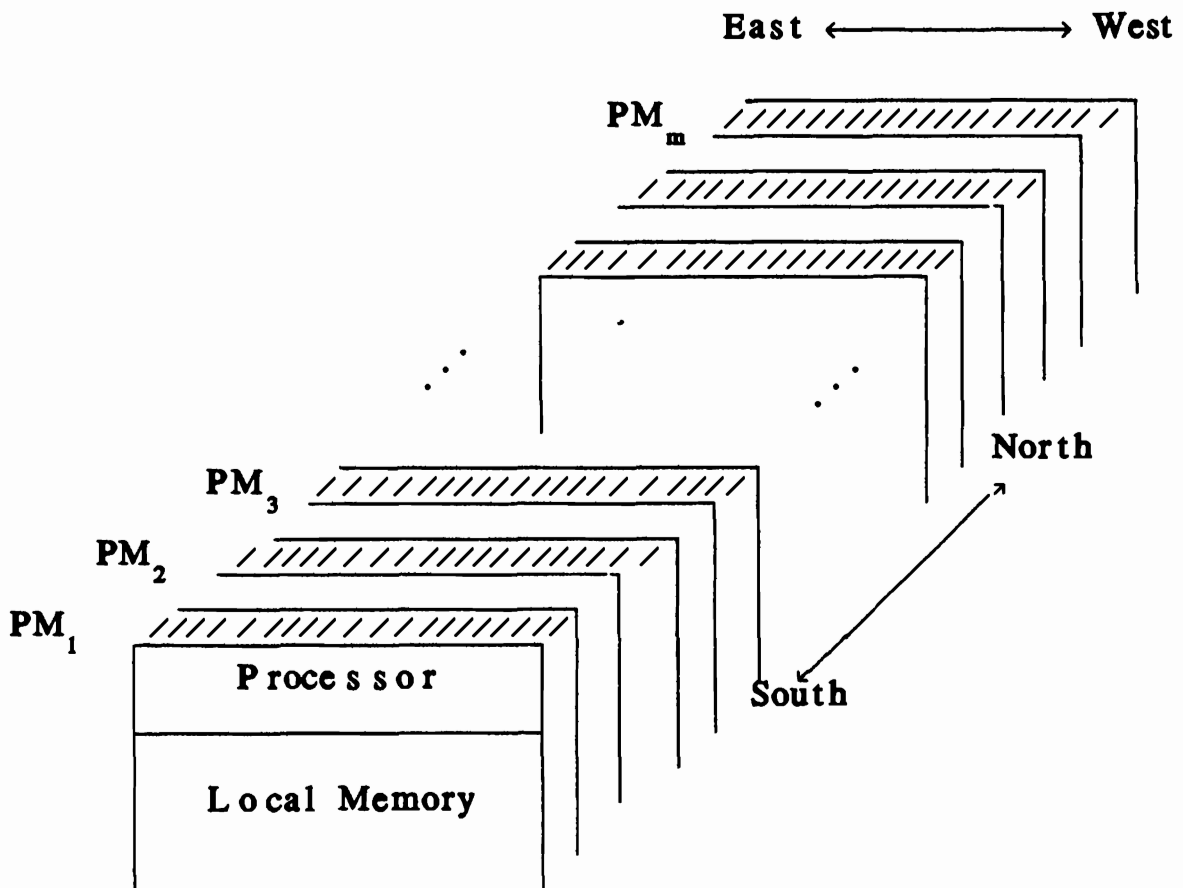


Fig.4

The suggested method of constructing VPS from PM-elements allows to organize fault-tolerant computing in the system. For this purpose, it

is necessary to add some redundant PM's and to make appropriate cross-connections via the multiport switches. Then, a definite fault-tolerant strategy of testing, isolation and replacement of PM's can be used.

The described two-dimensional VPS retains all properties of a linear 32-processor MIMD-structure, and each of the PM-units retains the properties of a standard horizontal processor.

It is to be noted, that such a situation always accompanies the evolution of computers: after each step of incorporation of a new feature, computers were able to perform what they performed before plus what came up as a specific additional feature. This property is also called *upward compatibility*. It offers a new opportunity to the traditional user or programmer without forcing him to renounce his successful habits, at least during a transition time. Such a kind of compatibility seems to be a general principle of innovative computer architecture.

6. Distributed Functional Structures

We have demonstrated above the correlation between horizontal, vertical and distributed processing. The distributed processing, as well as vertical, has a long history. It goes back to the J. von-Neumann's works on cellular automata.²⁴ In early 60ies E.Yevreinov has suggested an attractive conception of universal *Homogeneous Computing Medium*.²⁵ He investigated two-dimensional grids, consisting of uniform cells each of which can be configured as a functional, storage, or connection element. The communications with the outer world was envisioned only through the peripheral cells. The inner cells of the grid were connected only with the nearest neighbours. For this model various authors worked out a problem of mapping an arbitrary logical circuit into the medium.

When implementing in such universal distributed structures complicated data processing devices, significant time and area overheads arise. For this reason investigations of dedicated *Distributed Functional Structures (DF-structures)* are of considerable

interest, which allows to obtain a very high performance for corresponding basic functions or algorithms, and at the same time a good utilization of the chip area. As a matter of fact, the DF-structure performs physical simulation of an algorithm by spreading the signals through the uniform logical network.

A classical example of a specialized DF-structure is the content-addressed memory (CAM) with a basic function: equality search. Some authors examine the possibility of implementing in DF-structures other functions similar to CAM's. W.Kautz described several original DF-structures for data ordering,²⁶ intercommunications,²⁷ etc.

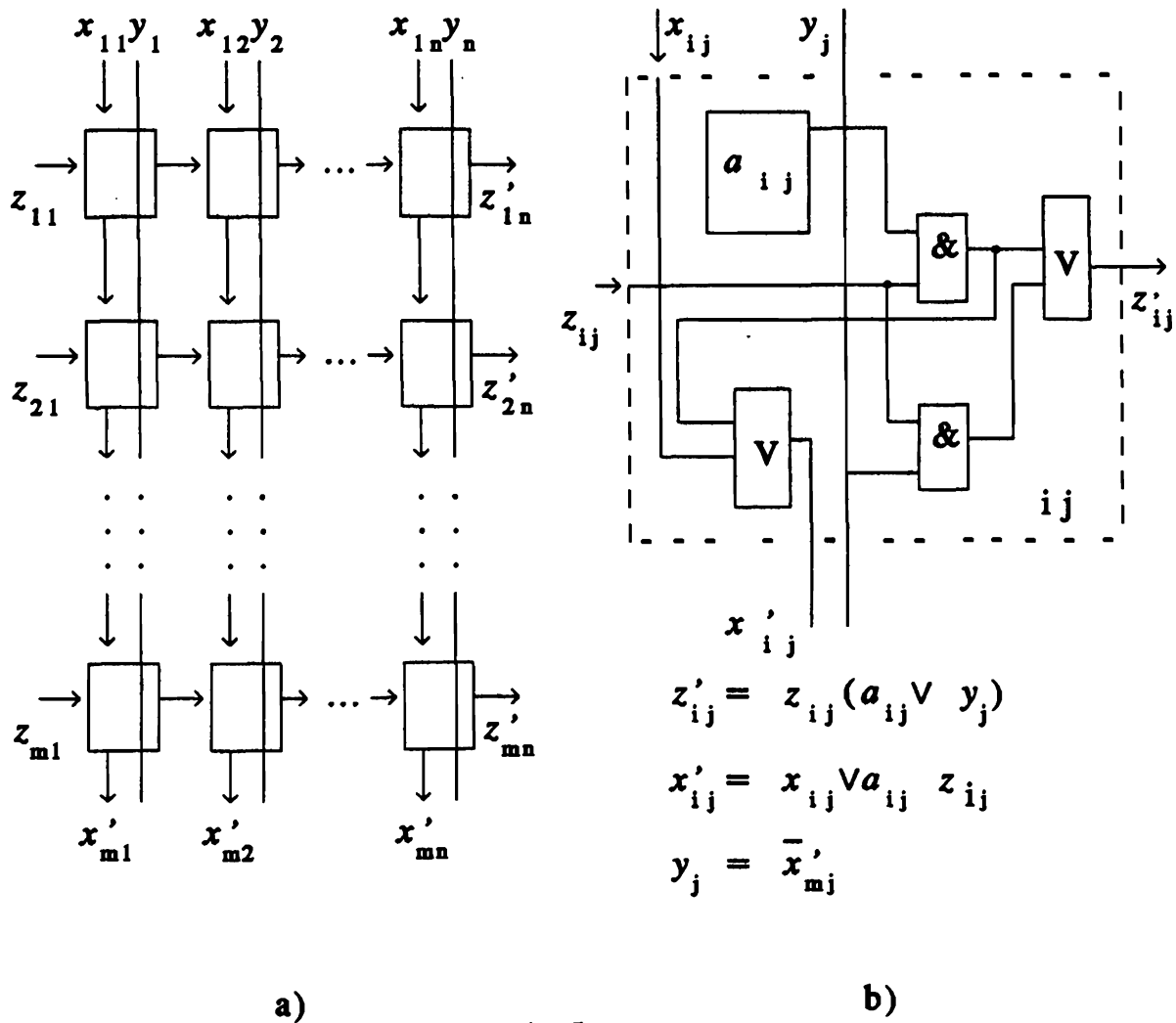


Fig.5

A DF-structure for direct implementation of extremum search algorithms has been suggested, which was called an α -structure.²⁸ A careful study of α -structure features was made in.^{29, 30} These

references contain also suggestions for other DF-structures for direct implementation of complex associative searches and other nonnumeric functions.

As an example, Fig.5 shows the flow-charts of the α -structure (Fig.5a) and its cell (Fig.5b). The elements of the array argument are written in the rows of the memory matrix. When applying the boundary constants $z_{i1}=1$ and $x_{1j}=0$, after the termination of the transition period, the signals $z'_{in}=1$ appear in those outputs of the right edge of the α -matrix, which corresponds to rows, containing the elements of greatest value.

Similar algorithms and structures were considered also by other authors.^{31,32,33}

An important feature for the practical realization in VLSI is the possibility of wide application of a standard module. From this point of view the specialization seems to be unfortunate. Nevertheless, in a right approach to designing, some DF-structures display a surprisingly flexibility and multifunctionality. So, for the α -structure, which is oriented to extremum searches, the following properties can be proved:

- an α -structure of size $2mn$ realizes a content-addressed memory with the capacity of m n -bit words;
- an α -structure of size $(2s+t)q$ realizes a reprogrammable PLA with s inputs, t outputs and q product terms;
- an α -structure can be used as a configurable homogeneous computing medium;
- an α -structure of size $m \times m$ represents a crossbar switch with m inputs and m outputs.

The correctness of the cited statements has been shown in.^{29,30} Furthermore, the logic circuits of different DF-structures, intended to implement their dedicated functions, often include some common fragments, which allows to design various combined structures. At the expense of a small complication of the initial cell, the properties of two or more DF-structures can be realized by the same module.

7. Specialized Processors for VPS

Due to the concurrent functioning of a very large number of PE's the contemporary VPS's reach extremely high overall performance. Another outstanding advantage of VPS's results from the fact that when using tens or hundreds of thousands of PE's, it is natural to reduce demands to each PE. In this way one can utilize a comparatively simple technology with corresponding reducing of cost. Consequently, the VPS's are distinguished by a large performance/cost ratio.

We have listed above the considerable advantages of VPS architecture. But at the same time the VPS has an important inherent shortcoming: the performance is restricted by the bit-serial processing principle itself.

One possible way to overcome this limitation is to supplement the system with specialized processors. The trend of using coprocessors can be seen in the progress of many novel architectures, including VPS. For example, *CM-2* employs coprocessors for floating-point arithmetic.

In our opinion, an important gain in VPS performance can be achieved by means of attaching some coprocessors, dedicated for increasing the speed of typical computational procedures, and based, in their turn, on vertical processing. At least two reasons can be cited in this context. First, the coprocessor intended to such supercomputers as VPS's, should, as well as the main system, possess a very high performance, which can be provided by the vertical principle. Second, a vertical coprocessor naturally harmonizes with the hardware structure of VPS, forming a common continuous pipeline.

As an example consider a Pipeline Vertical Adder.

The main purpose of this processor is the reductive summation of elements of an array. In the basic VPS architecture such a summation is usually accomplished by organizing the addends into a binary tree and performing parallel additions at each level of the tree. Assuming the sequential nature of each addition, this algorithm requires an order of $n \log_2 m$ time steps.

The vertical summation algorithm consists in sequential counting (beginning from the least significant bit) of the number of 1's contained in the corresponding bit-slices of the array. The obtained partial sums are added together with a systematic one-bit left shift, taking into account the position of subsequent slices. This procedure is of a pipeline nature, which defines the whole architecture of the coprocessor.

The *Pipeline Vertical Adder* (Fig.6) contains four units: the Digital Compressor DC, the Leading 1's Selector LS, the Code Converter CC and the Binary Accumulator BA. The first three units are intended to count the 1's in the slices (so called "weighting") and the last, - to add the partial sums. The weighting is organized as follows. At first, the DC transforms the next bit-slice (a binary vector with arbitrary arrangement of 1's) into an equivalent compressed code, i.e. a binary vector with the same number of 1's forming a dense block. Then, the LS generates a signal, marking the first 1 of the compressed code. Finally, the CC produces a binary number, corresponding to the weight of the given bit-slice.

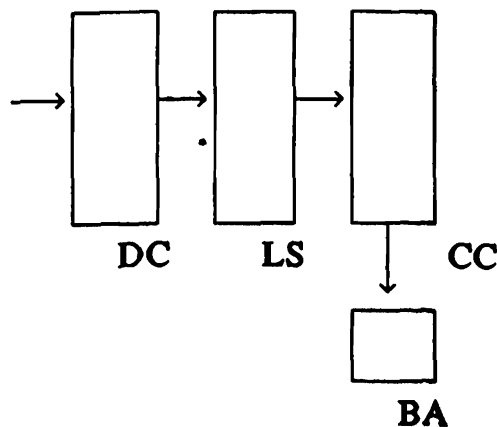


Fig.6.

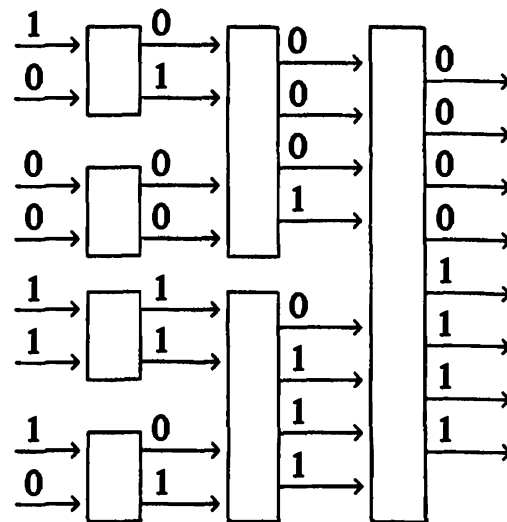


Fig.7.

Some issues of the design of digital compressors are considered in the book.³⁴ One of the known compressor structures is shown in Fig.7. This compressor has $\log_2 m$ stages (three stages for the example in which $m=8$). The first stage contains four ($m/2$) two-input compressors

The binary accumulator BA requires no discussion. However, it should be noted that the BA must be able to accomplish the shifting and accumulation of the next partial sum during one clock cycle. In this case, the whole time of reductive summation will depend only of the length n of the array element. Taking into account the time of the compressor pipeline filling (the depth of the pipeline is $\log_2 m$) the procedure of reductive summation in the vertical adder requires an order of $n + \log_2 m$ time steps, in contrast with $n \log_2 m$ in the basic VPS.

The application of pipeline vertical coprocessors is one possible way of enhancing the effectiveness of parallel computing systems. It represents a particular case of the general conception of *combined architectures*. The main idea of this conception is to join several highly efficient computing devices of different types and features into a united coordinate system. Another example of combined architecture is described in paper,³⁵ where a cooperation of VPS with a set of systolic arrays is suggested.

Conclusion

Vertical Processing becomes one of the most promising ways of supercomputing. The advantages of VPS's arise from the fact that, due to concurrent work of a very high number of PE's, the superhigh performance can be reached even at moderate speed of each PE. This allows to reduce the demands to the electronic components and so to design relatively cheap supercomputers on the basis of custom chips or standard processors.

Several hardware solutions for further improvement of VPS's are demonstrated. The distinguished features of these solutions are simplicity and multifunctionality. It is shown, in particular, how a standard n -bit processor can be used as a linear VPS of n PE's, and a set of n such processors as a two-dimensional VPS of the size $n \times n$.

The inherent restriction of the VPS due to bit-serial processing can be compensated in part by using coprocessors. The possibility of

implementing specialized processors with vertical and distributed processing as accelerators in VPS's is considered. Examples of a vertical adder and a flexible distributed functional structure are described, and advantages of this hardware are shown.

References

1. B.C. Cole. Content-addressable memories catch on. *Electronics*, vol.61, no.18, pp.82-84, Dec.1988.
2. F.C. Hennie. *Finite-state models for logical machines*. New York - London - Sydney : Wiley, 1968, 466 pp.
3. B. Parhami. Associative memories and processors: an overview and selected bibliography. *Proc. IEEE*, vol.61, no.6, pp.722-730, June 1973.
4. K.J. Thurber, L.D. Wald. Associative and parallel processors. *ACM Computing Surveys*, vol.7, no.4, pp.215-255, Dec.1975.
5. D.L. Slotnick. Logic-per-track devices. In *Advances in Computers*. New York : Academic Press, 1970, vol.10, pp.291-296.
6. J.L. Parker. A logic per track retrieval system. In *Proc. of IFIP Congress 71*. Amsterdam : North-Holland, 1971, vol.1, pp.711-716.
7. C.A. Finnilla, H.H. Love. The associative linear array processor. *IEEE Trans. Comput.*, vol.C-26, no.2, pp.112-125, Febr. 1977.
8. K.J.Thurber. *Large scale computer architectures*. Rochelle Park, NY: Hayden, 1976.
9. E. Ozkarahan. *Database machines and database management*. Englewood Cliffs, NJ : Prentice-Hall, 1986, 636 pp.
10. W. Shooman. Parallel computers with vertical data. In *AFIPS Confer. Proc.*, 1960 SJCC, vol.18, pp.111-115.
11. L.C Higbie. The OMEN computers: associative array processors. In *COMCON 72. 6th Annual IEEE Comp. Soc. Int. Confer.*, San Francisco, 1972, pp. 287-290.
12. W. Händler. *Rechenwerk einer digitalen Rechenmaschine*. Patentschrift 1157009, 1961.

13. W. Händler. Unconventional computation by conventional equipment. In *Proc. of the NATO Defence Research Group Seminar on Design and Evaluation of Inf. Systems*. Athens, 1974, pp.351-371.
14. W. Händler. A unified associative and von-Neumann processor EGPP and EGPP-array. In *Parallel Processing. Proc. of the Sagamore Computer Confer. 1974.* (Lecture Notes in Computer Science. Berlin : Springer. 1975, vol.24), pp.97-99.
15. K.E. Batchner. The multidimensional access memory in STARAN. *IEEE Trans. Comput.*, vol.C-26, no.2, pp.174-177, Febr.1977.
16. M. Gössel, B. Rebel, R. Creutzburg. *Speicherarchitektur und Parallelzugriff*. Berlin, 1989.
17. E.A. Metlitzky, V.V. Kaversnev. *Parallel memory systems*. Leningrad : University Press, 1989, 240 pp. (In Russian).
18. K.E. Batchner. STARAN parallel processor system hardware. In *AFIPS Confer. Proc.*, 1974 NCC, vol.43, pp.405-410.
19. D. Parkinson. The distributed array processor (DAP). *Computer Physics Communications*, vol.28, pp.325-336, 1983.
20. K.E. Batchner. Design of a massively parallel processor. *IEEE Trans. Comput.*, vol.C-29, no.9, pp.836-840, Sept.1980.
21. L.W. Tucker, G.G. Robertson. Architecture and applications of the Connection Machine. *IEEE Computer*, vol.21, no.8, pp.26-38, Aug.1988.
22. D.W. Blewins, E.W. Davis, R.A. Heaton, J.H. Reif. BLITZEN : a highly integrated massively parallel machine. *J.Parallel Distrib. Comput.*, vol.8, no.2, pp.150-160, Febr.1990.
23. H.J. Siegel. *Interconnection networks for large-scale parallel processing: theory and case studies*. Lexington, MA : Lexington Books, 1985.
24. J. von Neumann. *Theory of self-reproducing automata*. (Edited and completed by Arthur W.Burks). Urbana - London : University of Illinois Press, 1966.
25. E.V. Yevreinov, Yu.G. Kosarev. *High-performance universal homogeneous computing systems*. Novosibirsk : Nauka, 1966, 308 pp. (In Russian).

26. W.H. Kautz. Cellular logic-in-memory arrays. *IEEE Trans. Comput.*, vol.C-18, no.8, pp.719-727, Aug.1969.
27. W.H. Kautz, K.N. Levitt, A. Waksman. Cellular interconnection arrays. *IEEE Trans. Comput.*, vol.C-17, no.5, pp.443-451, May 1968.
28. Ya.I. Fet. *Data sorting device*. USSR Inventor's Certificate 424141, Cl. G06 F 7/00, 1971. (In Russian).
29. Ya.I. Fet. *Bulk information processing in specialized cellular processors*. Novosibirsk : Nauka, 1976, 200 pp. (In Russian).
30. Ya.I. Fet. *Parallel processors in control systems*. Moscow : Energoizdat, 1981, 156 pp. (In Russian).
31. C.V. Ramamoorthy, J.L. Turner, B.W. Wah. A design of a fast cellular associative memory for ordered retrieval. *IEEE Trans. Comput.*, vol.C-27, no.10, pp.800-815, Sept.1978.
32. W.A. Dawis, D.-L. Lee. Fast search algorithms for associative memories. *IEEE Trans. Comput.*, vol.C-35, no.5, pp.456-461, May 1986.
33. T.-Y. Feng. Search algorithms for bis-sequential machines. *J.Parallel Distrib. Comput.*, vol.8, no.1, pp.1-9, Jan.1990.
34. *Specialized architectures for high-speed data processing*. (V.E. Kotov, and N.N. Mirenkov, Eds.). Novosibirsk : Nauka, 1988, 208 pp. (In Russian).
35. A.P. Vazhenin, S.G. Sedukhin, Ya.I. Fet. High- performance computing systems of combined architecture. (*This Proceedings*).

METHODS OF PARALLEL INFORMATION PROCESSING IN THE ARCHITECTURE OF SOVIET HIGH-PERFORMANCE COMPUTERS

Boris A. Babajan

Institute of Precision Mechanics and Computer
Technology of the USSR Academy of Science,
Moscow, 117333, USSR

ABSTRACT

The present work has been fulfilled by the research team of the Institute of Precision Mechanics and Computer Technology of the USSR Academy of Sciences. From the very origin of the Soviet computer science this team has been dealing with the development of the most high-performance Soviet computers. The work of this team has been mainly connected with the perfection of computer architecture, which allowed to create sufficiently high-speed computers despite the considerable lag of the Soviet high-tech.

This paper contains the description of the main architectural achievements, which in combination with the more advanced foreign high-tech would allow to create computers with broad range of performance by their logic speed and cost/ performance ratio a few times surpassing the best world architectures. On the basis on these achievements the 16- processor "Elbrus-3" supercomputer has been designed, which has the 10 GFLOPS performance in spite of the comparatively low clock frequency (100 MHz).

On the basis of the developed architecture it is possible to create quite a high-performance and cost-effective microprocessor chip set, the marketing of which should be undoubtedly accompanied by considerable commercial success. This development included not only the creation of the architecture, but also the creation of highly optimized translator supporting the computer functioning.

The main results obtained for the "Elbrus-3" computer complex and the characteristics of the possible micriprocessor, which can be

created on the basis of this architecture in the course of the mutual design collaboration, will be given below. Then the main technical features of the architecture being discussed will be briefly stated. The proposed architecture allows to ensure the best possible logic speed of a computer for a given hardware volume. This result is attained thanks to the use of the inner fine-grain parallelism inherent to both vector and scalar algorithms. With the reasonably substantial hardware volume realized in the "Elbrus-3" processor it is possible to use practically all the parallelism existing in scalar algorithms. As a result, each processor of the "Elbrus-3" supercomputer features the following characteristics: the logic speed characterized by the mean number of machine cycles necessary for the execution of a single instruction (CPI) is equal to 0.1-0.15 cycles per instruction, i.e. the computer executes about 6-10 instructions within one cycle at an average.

At the clock cycle of 10 ns each processor of the "Elbrus -3" supercomputer while executing the standard LINPACK test package programs works approximately 2 times faster than the CRAY Y-MP supercomputer, which is one of the most high-performance machines in the world with the clock cycle of 6 ns. The results of the "Elbrus-3" supercomputer performance for the Livermore loops is given in Table.

LIVERMOR

#	1	235	#	14	170
#	2	62	#	15	225
#	3	325	#	16	203
#	4	49	#	17	68
#	5	128	#	18	413
#	6	142	#	19	99
#	7	517	#	20	52
#	8	322	#	21	164
#	9	319	#	22	264
#	10	169	#	23	325
#	11	93	#	24	88
#	12	184	harmonic mean		129
#	13	95	mean		193

Presently the work is being carried out on creating a microprocessor on the basis of this architecture. The possible

performance results, which can be obtained provided the advanced western electronic high-tech is used for its realization, are given below.

In contrast to the "Elbrus-3" supercomputer, where the reasonable degree of parallelism is used (7 parallel pipeline microchannels or up to 16 instructions initiated in parallel), it is proposed to use in the microprocessor only 2 parallel pipeline microchannels; this will make possible to execute simultaneously up to 7 instructions within each clock cycle. Thus, quite a favourable CPI value equal to 0.25 is ensured. As a result, at various clock frequencies the following performance values of the microprocessor for the standard LINPACK test package are obtained:

33 MHz - 110 MIPS

50 MHz - 180 MIPS

100 MHz - 350 MIPS.

The first two clock frequency values can be obtained while using the CMOS or Bi-CMOS technologies; the third value of the clock frequency is attainable only for those microprocessors, which are built on the basis of the ECL-technology. The volume of the necessary hardware is equal to approximately 5-7 mln. transistors.

For the sake of comparison it should be noted, that the recently announced new IBM RS/6000 RISC-microprocessor, which features the same hardware volume (7 mln. transistors) and the same clock frequency (33 MHz), has the performance of 13 MFLOPS or about 40 MIPS for the standard LINPACK test package; i.e. it works 3 times slower than our proposed architecture.

In case of mutual design cooperation it is necessary to develop a UNIX-oriented microprocessor, in which the bit-level emulational compatibility with one of the existing wide-spread RISC-microprocessors will be ensured. This will make its marketing much easier, because in this case the extensive libraries of existing application software will be readily available to the users. In our Institute the work has been conducted, which makes the possibility of such emulation effective enough. Our development substantially differs from other existing architectures supporting the execution of a number

of instructions within a single clock cycle, such as VLIW and superscalar architectures; in comparison with them it makes possible to attain almost three times better characteristics.

The basis of the architecture is the wide instruction, which allows to ensure the control of all the process resources within each clock cycle. This makes possible to realize the arbitrary schedule of units functioning. As a result, it is obvious, that this architecture can't be inferior in performance to any other architecture, because it is always possible to introduce into a program the schedule of functioning of units of any other computer ensuring strict repetition. The basis of the translation method for this architecture is the transformation of the initial sequence of instructions into the flow graph of the informational dependence of the algorithm. This graph clearly shows in full the parallelism present in the given program. There have been developed the methods of compactification of this graph into wide instructions which require linear time for their work and make possible to ensure for scalar algorithms the execution of a program within the time practically equal to the critical path of the information graph of this given program. This also proves, that in our architecture the theoretically top and best possible speed of execution of a given scalar task has been attained.

To significantly reduce the critical path of conditional clauses of a language all the operations have versions of conditional execution. In our parallel structure this allows to program the execution of conditional branches before the result of the calculation of the conditions of the choice of branches is obtained. As a result, after clearing up the conditions what is left is only to choose one of the alternative values calculated previously together with the calculation of conditions. This substantially increases the speed of scalar programs execution. Besides, for the operations having by-effects the conditional operations, into which the condition is fed explicitly as one of the initial arguments, have been also introduced. This allows to program some of the conditional branches situated outside the critical path without the use of real jumps and in parallel with the calculation of the main critical branch.

All this substantially (not less than by an order of magnitude) reduces the number of real physical jumps, because the semantics of conditional branches is realized with the help of conditional operations. In spite of this some special measures have been taken to accelerate real conditional jumps - the instruction of jump preparation has been introduced, which is executed long before the real jump takes place and in parallel with the main body of calculations.

For the fast execution of loop algorithms a unit is provided in the architecture, which calculates the regularly changing addresses of array elements and initiates a number of parallel array elements calls within each clock cycle. As a result, the main arithmetic units are completely relieved of the bulky calculations of address expressions and of loop end conditions.

The vectorization of programs is not necessary for the programming of loop expressions. This makes the creation of the translator considerably easier. The translator transforms the loop body into an information graph, which is the same as in the case of a scalar program. Then, if the iterations of the loop are informationally independent of each other, the translator programs the execution of the loop with the considerable overlay of successive iterations.

The loading of arithmetic units is the factor limiting the maximum overlay. Thus, the 100-percent loading of the unit having the biggest load for the given loop is ensured. As a result, even in the case of loop programs the best possible theoretical speed of control is obtained. In the presence of recurrent dependence between the iterations of the loop the translator uses the well-known method of recursive doubling. The cases of many other "heterogeneities" inside the loop such as conditional jumps, emergency exits, etc. are translated quite conveniently and effectively.

The brief summary of the main characteristics of the developed architecture is as follows:

a) the architecture makes possible to ensure high logic performance, which is 3 times higher than the performance of any other

modern architecture, including superscalar (IBM RS/6000, INTEL 80860 and 80960, etc.) and vector pipeline (CRAY Y-MP, NEC SX-3, etc.) architectures;

b) this architecture allows to obtain the best possible theoretical speed for a given set of arithmetic units. The CPI value for the 7-channel "Elbrus-3" supercomputer is equal to 0.1-0.15 cycles per instruction. The CPI value for the will-to-be 2-channel microprocessor is equal to 0.25 cycles per instruction.

The 7-channel version seemingly exhausts the inherent parallelism of scalar algorithms.

c) this architecture makes possible to develop the most cost-effective computers having the best cost/performance ratio;

d) the architecture is equally suitable for the use in workstations, minisupercomputers, mainframes and supercomputers. With the growth of integration it will be seemingly possible to effectively use this architecture in personal computers and built-in controllers.

It can be realized within any technology (CMOS, Bi-CMOS, ECL).

e) it is possible to develop a UNIX-oriented computer, which will be emulationally compatible at the binary level with one of the existing program bases for a chosen RISC architecture (SPARC, MIPS, RS/6000, MOTOROLA 88000, INTEL 80860, 80960, etc.).

INFLUENCE OF COMMUNICATIONS ON THE COMPLEXITY OF GIVENS FACTORIZATION ON A NETWORK OF PROCESSORS

M. COSNARD (1), E.M. DAOUDI (2) and B. TOURANCHEAU (1)

*(1) Ecole Normale Supérieure de Lyon, CNRS¹
46 Allée d'Italie
69364 Lyon Cedex 07, France*

*(2) Faculté Polytechnique de Mons
Rue de Houdain, 9
7000 Mons, Belgique*

ABSTRACT

We compare the complexity of the orthogonal decomposition using Givens method on three models of parallel architectures: a complete network, a linear array and a ring of processors. We show the influence of the architecture on the complexity of algorithms implemented on networks of processors working either by shared data or by message passing. Our main results show that the communications role between the host and the network is more important than communications between processors.

1. Introduction

Basic architecture of parallel models in the last decade was formed by a large common memory. Processors are identical and work using a shared memory: each processor reads the data from the common memory, performs its local work and writes the results in the common memory. Hence, parallel algorithms are based on scheduling technics and architectural problems mainly consist in memory access difficulties. To increase the flow of data from memory, several solutions are proposed in [1].

Memory access difficulties limit the number of processors: beyond a few dozen, the communication performances of the network decrease. To obtain a machine with a high degree of parallelism, one can use architectures with a decentralized memory and a limited connections network: each processor has a local memory and is connected to several neighbouring processors. Processors work by message passing: each processor receives data from processors, performs its local work and sends results to other processors. In this case, the execution time does not only depend on the computation time

¹This work has been supported by the Programme de Recherche Coordonnées C3 and the Direction des Etudes et Recherches Techniques

but also on the place of the processor in the network.

In this work we compare the complexity of orthogonal decomposition by Givens method [2] on several models of parallel architectures which work using a shared memory or by message passing. We call step the time necessary to read data, to perform the local work and to write results independently of the size of the vectors. We want to show the influence of the architecture on the complexity of parallel Givens algorithms.

Sameh [3, 4] has proposed several algorithms for solving linear systems on a linear array and on a ring of processors. Various methods to solve linear systems are presented in [5] on a ring of processors, with a bus for communication. In this model the number of processors is small compared to the problem size. The influence of communications in the design of parallel algorithms is studied in [6] for shared memory architectures and in [6][7] for networks of processors with local communications. Saad [8] takes Gauss method as a test algorithm to examine the possibilities of communication of several architectures composed with a great number of processors compared to the problem size. His main result shows that, when we take into account the communications, we do not obtain a linear execution time, since the communication time is of order $O(n^2)$ independently of the number of processors. In [9], Saad studies the implementation of Gauss elimination on a hypercube. He shows that the ring and the grid of processors suffice to solve the problem efficiently by duplication of matrix rows (grid) or using pipelining (ring). We use, in this work, a pipeline strategy method to obtain optimal algorithms.

In this paper, first of all, we recall complexity results obtained on shared memory. Then, we present complexity results on a linear array and on a ring of processors when the number of processors is unlimited ($p = \frac{n}{2}$). The communication between processors is local and the two networks communicate with the host by one processor. We show that $T_{opt} = 2n - 1$ and $p_{opt} = 5\frac{n}{2}$ and the communication between the host and the network is more important than communication between processors.

Afterwards we consider the same architecture to deduce a lower bound on the time complexity when the number p of processors is limited ($p < \frac{n}{2}$), we show that $T_{opt}(p) \geq n^2/2p + 2p + o(n)$. In this case the construction and the execution time of an optimal algorithm are still open. Then we show the influence of the architecture on the complexity results. We increase the connections between the host and the network. We assume that the host communicates with the network by two processors. We show that the optimal time $T_{opt} = 2n + o(n)$ is reached only with $n/3$ processors and when the number p of processors is limited we construct an optimal algorithm in $T_{opt}(p) = n^2/2p + 3p/2 + o(n)$. In section 6, we give a lower bound of the optimal time when the host communicates with an arbitrary number of processors.

2. Previous results for shared memory architectures

Throughout the paper, A is a dense square matrix of size n . We consider parallel algorithms to compute orthogonal decomposition via Givens method. We compute an orthogonal n by n matrix Q such that $QA=R$ where R is n by n upper triangular. Q is formed as the product of plane rotations. The sequential algorithm is well-known [2]. We let $R(i,j,k)$, $i \neq j$, $1 \leq i,j \leq n$ and $1 \leq k \leq n$, denote the rotation in plane (i,j) which annihilates the element $A(i,k)$. The elements are annihilated in the following way: in general from bottom to top and from left to right and all zeros introduced are preserved.

To parallelize Givens method we generate and apply independent rotations simultaneously. Since a plane rotation combines two rows to annihilate one element, then at most $\binom{n}{2}$ rotations can be performed simultaneously. So that, it is not necessary to have more than $\binom{n}{2}$ processors. For implementations of this algorithm on systolic networks and shared memory architectures (see e.g. [10], [11], [12], [13], [14], [15], [16]). The main results on the last type of network is recalled in this section. To describe informally the parallel algorithms, let us fix $n=8$.

The first table of figure 1 illustrates Sameh and Kuck's annihilation scheme [10]. It annihilates the elements in each column from bottom to top and in each row from left to right. An integer r is entered when zeros are created at the r -th step. We do not specify completely each rotation. A zero can be created in a row using any other row with the same number of annihilated elements. Sameh and Kuck's scheme is easy to program and to analyze: a possible choice is to use $R(i,i-1,k)$ in order to annihilate the element in position (i,k) . Clearly, the total number of steps is $2n-3$.

Cosnard and Robert [17] and independently Modi and Clarke [12] have introduced a greedy algorithm which performs at each step as many rotations as possible, annihilating the elements in each column from bottom to top and in each row from left to right. This scheme is depicted in the second table of figure 1.

*								*							
7	*							3	*						
6	8	*						2	5	*					
5	7	9	*					2	4	7	*				
4	6	8	10	*				1	3	6	8	*			
3	5	7	9	11	*			1	3	5	7	9	*		
2	4	6	8	10	12	*		1	2	4	6	8	10	*	
1	3	5	7	9	11	13	*	1	2	3	5	7	9	11	*
Sameh and Kuck's								Greedy							

Figure 1: Annihilation scheme

Theorem 1: On a shared memory architecture:

- The Greedy algorithm is optimal. Its execution time is equal to $2n-o(n)$ [17]
- Sameh and Kuck's algorithm is asymptotically optimal. Its execution time is equal to $2n-3$ [10].

In the remaining of this section, we give the complexity results (theorem 2) obtained by Cosnard, Daoudi and Robert [18] when the number of Givens rotations which can be performed simultaneously is limited to p ($p < (\frac{n}{2})$).

We call $T_{opt}(p)$ the optimal time for computing the Givens factorization with p processors and T_{opt} the optimal time when an unlimited number of processors is available (in fact we deduce from the preceding results that $T_{opt} = T_{opt}(p)$ for $p \geq (\frac{n}{2})$). The following relations are straightforward:

$$(i) T_{opt} = T_{opt}((\frac{n}{2})) \leq T_{opt}(p) \quad \text{for all } p \quad (ii) T_{opt} = 2n-o(n)$$

Theorem 2:

- For $1 \leq p \leq \frac{n}{2+\sqrt{2}}$, $T_{opt}(p) = \frac{n^2}{2p} + p + o(n)$.
- The minimum number of processors in order to obtain the Givens factorization of a square matrix of size n in optimal time, $T_{opt} = 2n-o(n)$, is asymptotically equal to:

$$P_{opt} = \frac{n}{2+\sqrt{2}}$$

3. Algorithms on a linear array

The shared memory architecture considered in the previous section can be viewed as a complete network of processors with $O(p^2)$ communication channels. So it possesses the maximum possibilities of communication. Therefore, an algorithm on a particular network can be executed on a shared memory architecture. Clearly the contrary is not possible, so the optimal time $2n-o(n)$ obtained for shared memory is a lower bound of the optimal time on another architecture. The network with the minimal capabilities of communication is the linear array. In this section we study the complexity of Givens factorization on a such network.

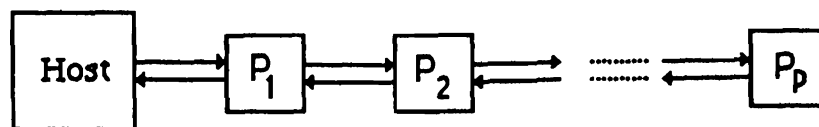


Figure 2: Linear array of processors

The linear array is a multiprocessor network composed with p processors,

numbered from P_1 to P_p , connected as indicated on figure 2. Each processor has its own local memory large enough to store at least two rows of the matrix A . Processor P_i ($2 \leq i \leq p-1$) communicates with P_{i-1} and P_{i+1} by one channel for read and one channel for write. The processor P_p communicates only with processor P_{p-1} . The host computer communicates with processor P_1 . We assume that the unit, called "step", is the time necessary to read one row, perform one Givens rotation and send one row irrespectively of the position k where the new zero is introduced. At the beginning, the matrix A is stored in the host and at the end, the host must contain the resulting matrix R . Each processor P_i operates as indicated on figure 3.

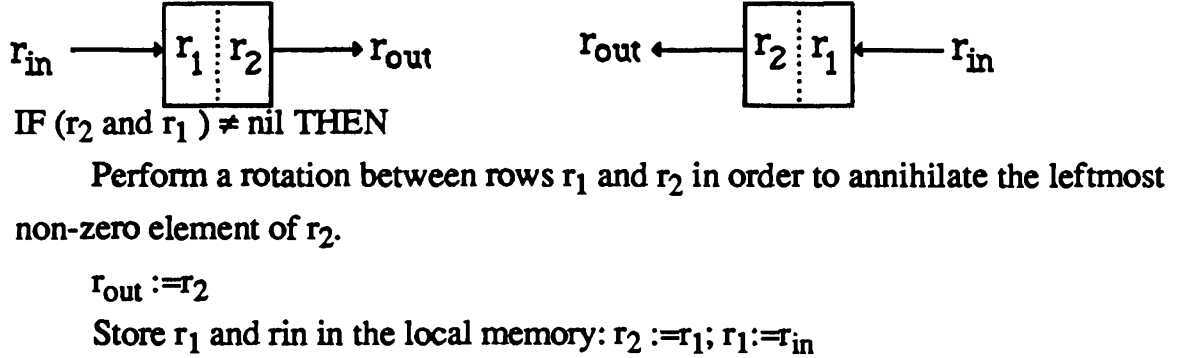


Figure 3: Architecture and program of the processors

First of all, we give some notations and definitions. Let :

- $ac(t)$ be the number of active processors (which are annihilating) at step t
- $id(t)$ be the number of idle processors at step t
- $T(p)$ be the execution time with p processors

The following relation holds: $id(t) = p - ac(t)$ (1)

Define $P_{ac} = \sum_{t=1}^{T(p)} ac(t)$ and $P_{id} = \sum_{t=1}^{T(p)} id(t)$

$\sum_{t=1}^{T(p)} ac(t)$ is the number of annihilated elements equal to $n(n-1)/2$

Using (1) Lord et al. [11] derive the following:

Property 1: $pT(p) = P_{id} + P_{ac}$

3.1. A lower bound

In this section, we shall derive a lower bound for the complexity of parallel Givens factorization on a linear array.

Theorem 3: Let $T_{opt}(p)$ be the optimal time to compute Givens factorization of a square matrix of size n on a linear array composed with p processors ($p \leq (\frac{n}{2})$). We have :

$$T_{opt}(p) \geq \frac{n(n-1)}{2p} + 2p$$

Proof:

(i) Since the rows of A can only be introduced sequentially from processor P_1 , we deduce that $ac(t) \leq (t/2)$ for $1 \leq t \leq 2p$. Therefore $\sum_{t=1}^{2p} id(t) \geq p^2$.

(ii) Let t_i be the step at which the element in position $(i+1, i)$ is annihilated. Since two rows are used for annihilating one element then $ac(t_i) \leq \min(p, (\frac{n-i+1}{2}))$. So, for $i \geq n-2p+2$,

we have: $ac(t_i) \leq (\frac{n-i+1}{2})$ and $id(t_i) \geq p - (\frac{n-i+1}{2})$. Therefore $\sum_{i=n-2p+2}^{n-1} id(t_i) \geq p(p-1)$.

From (i), (ii) and since we write the row n in one step we deduce that $P_{id} \geq p(2p-1) + p = 2p^2$

According to property 1 we conclude that: $T_{opt}(p) \geq (\frac{n(n-1)}{2} + 2p^2)/p$

3.2. An optimal algorithm

We assume that n is even. In this section we propose an implementation of Givens factorization which proves that the previous lower bound is reached for $p = \frac{n}{2}$. Clearly we cannot execute the greedy algorithm since it performs several Givens rotations at the first step. On the other hand, Sameh and Kuck's algorithm can be implemented directly.

The optimal GivensRL algorithm consists in two phases. The rows are pipelined in the network from row n to row 1 in the following way:

First phase: From step $t=1$ up to $t=n$, the rows are pipelined in the network, each row staying two units of time in P_i before being transmitted to P_{i+1} . At each time unit processor P_i operates as indicated on the first scheme of figure 3: it reads one row from P_{i-1} and performs the corresponding Givens rotation.

Second phase: From step $t=n+1$ up to $t=2n-1$, the data flow is reversed: P_i reads one row from P_{i+1} and performs the corresponding Givens rotation. During this phase, processor P_1 delivers at each step a new row of the resulting matrix R . Processors operate as indicated on the second scheme of figure 3.

This behavior does not need a global control because the data flow is reversed when processor P_i reads row $2i-1$. The execution time of this algorithm, described on figure 4, is equal to $(2n-1)$. The rotations are accomplished in the same way as in Sameh and Kuck's annihilation scheme.

Theorem 4: On a linear array:

- $T_{opt} = 2n-1$

- The minimum number of processors p_{opt} to obtain Givens factorization in optimal time is equal to $5\frac{n}{2}$.

Proof:

- To prove that $T_{opt}=2n-1$ it suffices to remark that $\frac{n(n-1)}{2p} + 2p \geq 2n - 1$ for all integer p .

- p_{opt} is obtained by the value of p for which $\frac{n(n-1)}{2p} + 2p = 2n - 1$.

	P_1		P_2		P_3		P_4	
$t=1$	8							
$t=2$	78	$R(8,7,1)$						
$t=3$	67	$R(7,6,1)$	8					
$t=4$	56	$R(6,5,1)$	78	$R(8,7,2)$				
$t=5$	45	$R(5,4,1)$	67	$R(7,6,2)$	8			
$t=6$	34	$R(4,3,1)$	56	$R(6,5,2)$	78	$R(8,7,3)$		
$t=7$	23	$R(3,2,1)$	45	$R(5,4,2)$	67	$R(7,6,3)$	8	
$t=8$	12	$R(2,1,1)$	34	$R(4,3,2)$	56	$R(6,5,3)$	78	$R(8,7,4)$
$t=9$	23	$R(3,2,2)$	45	$R(5,4,3)$	67	$R(7,6,4)$	8	
$t=10$	34	$R(4,3,3)$	56	$R(6,5,4)$	78	$R(8,7,5)$		
$t=11$	45	$R(5,4,4)$	67	$R(7,6,5)$	8			
$t=12$	56	$R(6,5,5)$	78	$R(8,7,6)$				
$t=13$	67	$R(7,6,6)$	8					
$t=14$	78	$R(8,7,7)$						
$t=15$	8							

Figure 4: GivensRL algorithm for matrix of size (8,8)

Remarks:

- (i) When n is odd, processor $P_{5n/2^0}$ is only used to store one row.
- (ii) The channels for communication between P_i and P_{i+1} are never used simultaneously. Therefore theorem 3 is also verified when two processors communicate by one bi-directional channel.
- (iii) Evaluating the complexity (construction and execution time of the optimal algorithm) for $p < (\frac{n}{2})$ is still open. In both cases, it is clear that the lower bound is not reached.

4. Algorithms on a ring

A ring of processors as depicted on the figure 5, is a network with more communication capabilities than a linear array. Each processor P_i ($2 \leq i \leq p$) communicates with processors P_{i-1} and P_{i+1} (modulo p) by a bi-directional channel. In addition

processor P_1 communicates with the host by a bi-directional channel.

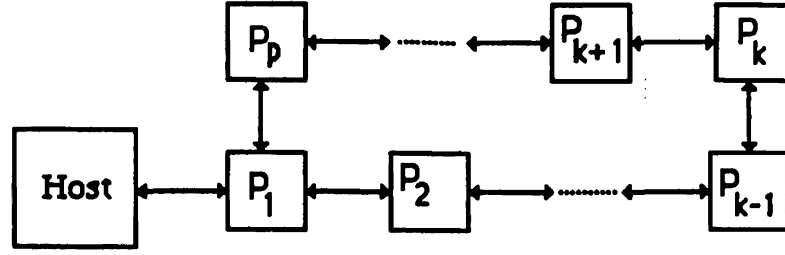


Figure 5: Ring which communicates with the host by one processor (processor P_1)

The linear array is a sub-network of the ring. Therefore, the algorithms of the previous section can be executed on the ring. By the same previous proofs we show that theorems 3 and 4 are verified for the ring.

5. Algorithms on a linear array which communicates with the host by two processors

This section is devoted to the influence of the architecture on the complexity of algorithms presented for the linear array. Now, we assume that the host communicates simultaneously with processors P_1 and P_p by a bi-directional channel (figure 6).

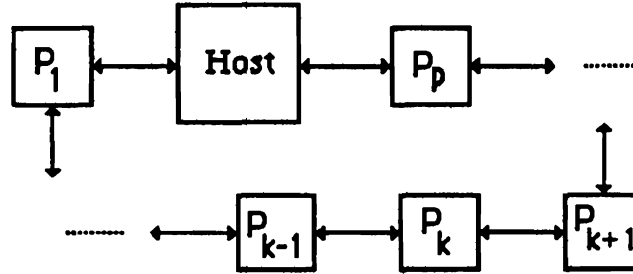


Figure 6: Linear array which communicates with the host by P_1 and P_p

5.1. A lower bound

In this section, we shall derive a lower bound for the complexity of parallel Givens factorization on such a network.

Theorem 5: Let $T_{\text{opt}}(p)$ be the optimal time to compute Givens factorization on a linear array composed with p processors which communicates with the host by processors P_1 and P_p . We have:

$$T_{\text{opt}}(p) \geq \frac{n(n-1)}{2p} + \frac{3p}{2}$$

Proof:

(i) Since the rows are simultaneously introduced in the network by processors P_1 and P_p ,

we deduce that $ac(t) \leq 2(t/2)$ for $1 \leq t \leq p$. Therefore $\sum_{t=1}^p id(t) \geq p^2/2$

(ii) Let t_i be the step at which the element in position $(i+1, i)$ is annihilated. So for $i \geq n-$

$2p+2$ we have $ac(t_i) \leq (\frac{n-i+1}{2})$ and $id(t_i) \geq p - (\frac{n-i+1}{2})$. Therefore $\sum_{i=n-2p+2}^{n-1} id(t_i) \geq p(p-1)$.

From (i), (ii) and since we write the row n in one step we deduce that $P_{id} \geq \frac{3p^2}{2}$

According to the property 1 we conclude that $T_{opt}(p) \geq \frac{n(n-1)}{2p} + \frac{3p}{2}$

The following result is a straightforward consequence of the theorem 5.

Corollary 2: Let p_{opt} be the minimum number of processors for computing Givens factorization of a dense square matrix of size n in $T_{opt} = 2n - o(n)$. Asymptotically, we have

$$\frac{n}{3} \leq P_{opt} \leq \frac{n}{2}$$

5.2. An asymptotically optimal algorithm

In this section we propose an implementation of Givens factorization which proves that the previous lower bound can be reached with $p = \frac{n}{3}$. The algorithm which we construct is described on figure 7 where $n=12$ and $p=4$. It proceeds in four sequential phases:

First phase: From step $t=1$ up to $t=p$, the rows are pipelined in the network in the following way:

(i) From row n to row $n-p+1$ by the processor P_1 . Each row stays during two steps in P_{i-1} before being transmitted to P_i . Processor P_i , $2 \leq i \leq p/2$, operates as indicated on the first scheme of figure 3.

(ii) From row p to row 1 by the processor P_p . Each row stays during two steps in P_{i-1} before being transmitted to P_i . Processor P_i , $p/2 \leq i \leq p-1$, operates as indicated on the second scheme of figure 3.

Second phase: From step $t=p+1$ up to $t=2p$, the rows are pipelined in the network by P_1 from row $n-p$ to row $n-2p+2$. On the other hand the data flow in the last $p/2$ processors is reversed: P_i reads one row from P_{i-1} and performs the corresponding rotation. During this phase, P_p writes at each step one row of the matrix A in the host.

Third phase: From step $t=2p+1$ up to $t=2n-2p-1$, the data flow changes at each step.

- At step $t=2p+2j+1$, for $0 \leq j \leq n-2p-1$, P_1 reads from the host the row $j+1$ and P_p writes the row n in the host. For $2 \leq i \leq p-1$, P_i operates as indicated on the first scheme of figure 3.

- At step $t=2p+2j+2$, P_1 writes one new row (row $j+1$) of the resulting matrix R in the host and P_p reads the row n from the host. For $2 \leq i \leq p-1$, P_{i-1} operates as indicated on the second scheme of figure 3.

Fourth phase: From step $t=2n-2p$ up to $t=2n-1$, P_i reads one row from P_{i+1}

and performs the corresponding rotation. During this phase, P_1 delivers at each time step one new row of the resulting matrix R .

	P_1	P_2	P_3	P_4
$t=1$	12			4
$t=2$	11.12 R(12,11,1)			R(4,3,1) 4.3
$t=3$	10.11 R(11,10,1)	12	4	R(3,2,1) 3.2
$t=4$	9.10 R(10,9,1)	11.12 R(12,11,2)	R(4,3,2) 4.3	R(2,1,1) 2.1
$t=5$	8.9 R(9,8,1)	10.11 R(11,10,2)	12 4	R(3,2,2) 3.2
$t=6$	7.8 R(8,7,1)	9.10 R(10,9,2)	11.12 R(12,11,3)	R(4,3,3) 4.3
$t=7$	6.7 R(7,6,1)	8.9 R(9,8,2)	10.11 R(11,10,3)	12 4
$t=8$	5.6 R(6,5,1)	7.8 R(8,7,2)	9.10 R(10,9,3)	11.12 R(12,11,4)
$t=9$	1.5 R(5,1,1)	6.7 R(7,6,2)	8.9 R(9,8,3)	10.11 R(11,10,4)
$t=10$	5.6 R(6,5,2)	7.8 R(8,7,3)	9.10 R(10,9,4)	11.12 R(12,11,5)
$t=11$	2.5 R(5,2,2)	6.7 R(7,6,3)	8.9 R(9,8,4)	10.11 R(11,10,5)
$t=12$	5.6 R(6,5,3)	7.8 R(8,7,4)	9.10 R(10,9,5)	11.12 R(12,11,6)
$t=13$	3.5 R(5,3,3)	6.7 R(7,6,4)	8.9 R(9,8,5)	10.11 R(11,10,6)
$t=14$	5.6 R(6,5,4)	7.8 R(8,7,5)	9.10 R(10,9,6)	11.12 R(12,11,7)
$t=15$	4.5 R(5,4,4)	6.7 R(7,6,5)	8.9 R(9,8,6)	10.11 R(11,10,7)
$t=16$	5.6 R(6,5,5)	7.8 R(8,7,6)	9.10 R(10,9,7)	11.12 R(12,11,8)
$t=17$	6.7 R(7,6,6)	8.9 R(9,8,7)	10.11 R(11,10,8)	12
$t=18$	7.8 R(8,7,7)	9.10 R(10,9,8)	11.12 R(12,11,9)	
$t=19$	8.9 R(9,8,8)	10.11 R(11,10,9)	12	
$t=20$	9.10 R(10,9,9)	11.12 R(12,11,10)		
$t=21$	10.11 R(11,10,10)	12		
$t=22$	11.12 R(12,11,11)			
$t=23$	12			

Figure 7: Algorithm with $p=n/3$ on the linear array which communicates with the host by 2 processors (rows are entering the network by both sides, the direction is given by the position of the R() rotation)

Theorem 6: On a ring, composed with p processors, which communicates with the host by two processors (P_1 and P_p):

- (i) The optimal time to compute the Givens factorization is asymptotically equal to $2n-1$.
- (ii) The minimum number of processors is asymptotically equal to $\frac{n}{3}$.

Proof: - To prove (i) it suffices to remark that $\frac{n^2}{2p} + \frac{3p}{2} \geq 2n$ for all $p \leq \frac{n}{3}$.

- The asymptotic value of p_{opt} is obtained by the value of p for which $\frac{n^2}{2p} + \frac{3p}{2} = 2n$.

The algorithm is based on the following ideas:

- (i) Since the first block is composed by $3p$ rows, we apply the optimal algorithm proposed for $p=n/3$.
- (ii) While one of the two extremities (P_1 or P_p) of the linear array delivers one row of the block $(j-1)$ ($j>1$), by the other extremity we introduce the rows of the block j , from the row $(2j+1)p$.

The algorithm is constructed in k sequential phases and is described on figure 9 where $n=20$ and $p=4$.

Description of the phases of the algorithm :

First phase: From step $t_1=1$ up to $f_1=t_1+4p-1$, it consists in applying the first three phases of the algorithm 5.2. in order to annihilate the elements in the first block.

Second phase: From step $t_2=f_1+1$ up to $f_2=t_2+8p-1$. We proceed in 2 stages.

- **First stage:** consists in annihilating simultaneously in the first and the second blocks in the following way. While P_1 delivers, at each step, one row of the first block, P_p reads one row of the second block from the host starting by the row $5p$. So we execute: the first phase of GivensRL algorithm for the second block and the second phase of GivensRL algorithm for the first block.
- **Second stage:** starts at time step $t_2+2(p-1)$. It consists in annihilating in the second block in the same way as the third phase of algorithm 5.2.

Third phase: From step $t_3=f_2+1$ up to $f_3=t_3+12p-1$. It annihilates in the same way as the second phase. The only difference consists in the fact that the role of P_1 and P_p is reversed. In general:

j-th phase: From step $t_j=f_{j-1}+1$ up to $f_j=t_j+4jp-2$. It annihilates in the same way as the second phase if j is even and as the third phase if j is odd.

k-th phase: From step $t_k=f_k-1+1$ up to $f_k=t_k+2n-3$. We proceed in 3 stages:

- **First stage:** we annihilate in the same way as the first stage of the j -th phase
- **Second stage:** we annihilate in the same way as the second stage of the j -th phase but we deliver at each 2 time steps one new row of the resulting matrix R .
- **Third stage:** during this stage, the processor P_1 , if k is even (otherwise processor P_p), delivers at each time step one new row of the resulting matrix R .

Execution time: The execution time is equal to $T(p)=f_k=t_k+2n-3$. From relations $f_j=t_j+4jp-2$ and $t_j=f_{j-1}+1$ for $1 \leq j \leq k$ we deduce that $t_k=1+2k(k-1)p$. Therefore we have $T(p)=2k(k-1)p+2n-2$. Since $n=(2k+1)p$ then we replace k in the last formula by $(n-p)/2p$.

Asymptotically we obtain: $T_{\text{opt}}(p)=\frac{n^2}{2p} + \frac{3p}{2}$

	P ₁		P ₂		P ₃		P ₄	
...(see figure 7)								
t=16	R(6,5,5)	6.5	R(8,7,6)	8.7	R(10,9,7)	9.10	R(12,11,8)	11.12
t=17	R(7,6,6)	6.7	R(9,8,7)	9.8	R(11,10,8)	10.11	12	20
t=18	R(8,7,7)	7.8	R(10,9,8)	9.10	R(12,11,9)	11.12	R(20,19,1)	20.19
t=19	R(9,8,8)	8.9	R(11,10,9)	10.11	12	20	R(19,18,1)	19.18
t=20	R(10,9,9)	9.10	R(12,11,10)	11.12	R(20,19,2)	20.19	R(18,17,1)	18.17
t=21	R(11,10,10)	10.11	12	20	R(19,18,2)	19.18	R(17,16,1)	17.16
t=22	R(12,11,11)	11.12	R(20,19,3)	20.19	R(18,17,2)	18.17	R(16,15,1)	16.15
t=23		12.20	R(19,18,3)	19.18	R(17,16,2)	17.16	R(15,14,1)	15.14
t=24	R(20,19,4)	20.19	R(18,17,3)	18.17	R(16,15,2)	16.15	R(14,13,1)	14.13
t=25	R(19,18,4)	19.18	R(17,16,3)	17.16	R(15,14,2)	15.14	R(13,1,1)	13.1
t=26	R(20,19,5)	20.19	R(18,17,4)	18.17	R(16,15,3)	16.15	R(14,13,2)	14.13
t=27	R(19,18,5)	19.18	R(17,16,4)	17.16	R(15,14,3)	15.14	R(13,2,2)	13.2
t=28	R(20,19,6)	20.19	R(18,17,5)	18.17	R(16,15,4)	16.15	R(14,13,3)	14.13
t=29	R(19,18,6)	19.18	R(17,16,5)	17.16	R(15,14,4)	15.14	R(13,3,3)	13.3
t=30	R(20,19,7)	20.19	R(18,17,6)	18.17	R(16,15,5)	16.15	R(14,13,4)	14.13
...								
t=46	R(20,19,15)	20.19	R(18,17,14)	18.17	R(16,15,13)	16.15	R(14,13,12)	14.13
t=47	R(19,18,15)	19.18	R(17,16,14)	17.16	R(15,14,13)	15.14	R(13,9,12)	13.12
t=48	R(20,19,16)	20.19	R(18,17,15)	18.17	R(16,15,14)	16.15	R(14,13,13)	14.13
t=49		20	R(19,18,16)	19.18	R(17,16,15)	17.16	R(15,14,14)	15.14
t=50			R(20,19,17)	20.19	R(18,17,16)	18.17	R(16,15,15)	16.15
t=51				20	R(19,18,17)	19.18	R(17,16,16)	17.16
t=52					R(20,19,18)	20.19	R(18,17,17)	18.17
t=53						20	R(19,18,18)	19.18
t=54							R(20,19,19)	20.19
t=55								20

Figure 9: Givens algorithm for $n=20$ and $p=4$ on the ring which communicates with the host by processors P_1 and P_4

6. A lower bound of the optimal time when the host communicates with an arbitrary number of processors of the ring

In this section we give a lower bound of the optimal time when the host communicates with an arbitrary number of processors.

Theorem 8: Let x be the number of processors which communicate with the host and $y=1/x$. We have:

$$T_{\text{opt}}(p) \geq \frac{n^2}{2p} + (1 + \frac{1}{x})p = \frac{n^2}{2p} + (1+y)p$$

$$\text{and } p_{\text{opt}} \geq \frac{xn}{x+1} (1 - \sqrt{\frac{x-1}{2x}}) = \frac{n}{1+y} (1 - \sqrt{\frac{1-y}{2}})$$

Proof:

(i) Since the rows are simultaneously introduced in the network by x processors, we deduce

that, we deduce that $ac(t) \leq \frac{xt}{2}$ for $1 \leq t \leq \frac{2p}{x}$. Therefore $\sum_{t=1}^{\frac{2p}{x}} id(t) \geq \frac{p^2}{x}$

(ii) Let t_i be the step at which the element in position $(i+1, i)$ is annihilated. Since we use two rows for annihilating one element then: $ac(t_i) \leq \min(p, (\frac{n-i+1}{2}))$. So, for $i \geq n-$

$2p+2$, we have: $ac(t_i) \leq (\frac{n-i+1}{2})$ and $id(t_i) \geq p - (\frac{n-i+1}{2})$. Therefore $\sum_{i=n-2p+2}^{n-1} id(t_i) \geq p(p-1)$.

From (i) and (ii) we deduce that $P_{id} \geq p^2 + \frac{p^2}{x} = p^2(1+y)$

According to the property 1 we conclude that: $T_{\text{opt}}(p) \geq \frac{n^2}{2p} + (1+y)p$

The lower bound of p_{opt} is obtained by the value of p for which $\frac{n^2}{2p} + (1+y)p = 2n$.

7. Conclusion

We have studied the orthogonal decomposition via Givens method for square matrix of size n on two networks composed with p processors: the linear array and the ring. The results obtained show the influence, of both the network and the host - multicomputer communication architecture, on the algorithms proposed. Following the number of processors we distinguish two cases:

Case of unlimited number of processors: For the complete network, the processors have parallel access to the host (common memory). On the other hand the linear array and the ring of processors have sequential access. This has a small influence on the complexity because the performances obtained on each network are of the same order: $2n-o(n)$ for the complete network and $2n-1$ for the linear array and for the ring. On the other hand the minimum number of processors to compute Givens factorization in asymptotically optimal time is equal to $n/(2+\sqrt{2})$ for the complete network, $\frac{n}{2}$ for the linear

array and for the ring which communicates with the host by one processor (Ring 1) and only $\frac{n}{3}$ when we increase the communication means (ring communicates with the host by two processors). We deduce that the communication part between the network and the host is more important than communication part between the processors. For example an hypercube which communicates with the host by one processor is not better than a linear array.

Case of limited number of processors: In this case the difference of performances obtained on the three models of architectures cannot be neglected. For the complete network, processors have parallel accesses to the host (common memory), therefore we annihilate with full efficiency all time except during the last $2p$ steps [18]. When the processors have a sequential access to the network, we cannot annihilate with full efficiency during the first $2p$ steps for the linear array and for the ring which communicates with the host by one processor, therefore the performances decrease. The augmentation of communication channels between the host and the network has a great influence on the complexity results: we have shown that, in the case where the host communicates with the network by two processors (Ring 2), we are not able to annihilate with full efficiency only during the first p steps. The last result gives a general lower bound when the host can communicate with an arbitrary number of processors (Ring x).

On table 1 we show the complete results obtained on the three models of architectures.

Givens Algorithm	T_{opt}	P_{opt}	$T_{opt}(p)$
Shared memory	$(2n-o(n))$	$\frac{n}{2+\sqrt{2}}$	$\frac{n^2}{2p}+p+o(n)$
Linear array	$2n-1$	$\frac{n}{2}$	$\geq \frac{n^2}{2p}+2p+o(n)$
Ring 1	$2n-1$	$\frac{n}{2}$	$\geq \frac{n^2}{2p}+2p+o(n)$
Ring 2	$2n-1$	$\frac{n}{3}$	$\frac{n^2}{2p}+\frac{3p}{2}+o(n)$
Ring x	$(2n-o(n))$	$\geq \left(\frac{xn}{x+1}\right)\left(1-\sqrt{\frac{x-1}{2x}}\right)$	$\geq \frac{n^2}{2p}+\left(1+\frac{1}{x}\right)p$

Table 1: Complexity results of Givens factorization on several multiprocessor architectures.

References

1. K. HWANG, F. BRIGGS, *Parallel processing and computer architecture*, (Mc Graw-Hill 1984)
2. G.H. GOLUB, C.F. VAN LOAN, *Matrix Computations*, (The John Hopkins Univ. Press, 1983)
3. A. SAMEH, "Solving the linear least squares problem on a linear array of processors", *Proc. Purdue Workshop on algorithmically-specialized computer organizations*, W. Lafayette, (Indiana, 1982)
4. A SAMEH, "On some parallel algorithms on a ring processors", *Computer Phys. Com.* 37 (1985) 159-166
5. I.C.F. IPSEN, Y. SAAD, M.H. SCHULTZ, "Complexity of dense linear system solution on a multiprocessor ring", *Computer Science Dpt, yale University* RR 349 (1985)
6. D. GANNON, J. VAN ROSENDALE, " On the impact of communication in the design of parallel algorithms ", *IEEE T.C.* 33, 12, (1984), 1180-1194
7. G.W. STEWART, "Communication and matrix computation on large message passing sytems", *Parallel Computing* 16 (1990), 27-40
8. Y.SAAD, "Communication complexity of Gaussian elimination algorithm on multiprocessors", *Computer Science Dpt., Yale University* R R 348 (1985)
9. Y. SAAD, "Gaussian elimination on hypercube, in *Parallel Algorithms and Architecture*, eds. M. COSNARD et al., North-Holland (1986), 5-18
10. A. SAMEH, D. KUCK, "On stable parallel linear system solvers", *J. ACM* 25 (1), (1978), 81-91
11. R.E. LORD, J.S. KOWALIK, S.P. KUMAR, "Solving linear algebraic equations on an MIMD computer", *J. ACM* 30 (1), (1983), 103-117
12. J.J. MODI, M.R.B. CLARKE, "An alternative Givens ordering", *Numerische Mathematik* 43, (1984), 83-90
13. E. CHU, A. GEORGE, "QR factorization of a dense matrix on a shared-memory multiprocessor", *Parallel computing* 11 (1989), 55-71
14. M. COSNARD, J.M. MULLER, Y. ROBERT, "Parallel QR decomposition of a rectangular matrix", *Num. Math.* 48, (1986), 239-249
15. M. COSNARD, Y. ROBERT, "Systolic Givens factorization of dense rectangular matrices", *Intert. Journal computer Mathematics*, 25, (1988), 287-298
16. A. BOJANCZYK, R.P. BRENT, H.T. KUNG, "Numerically stable solution of dense systems of linear equations using mesh-connected processors", *Tech. Rep.*, Carnegie Mellon University (1981)
17. M. COSNARD, Y. ROBERT, "Complexity of parallel QR decomposition", *J. ACM* 33 (4), (1986), 712-723
18. M. COSNARD, E.M. DAOUDI, Y. ROBERT, "Complexity of parallel Givens factorization on shared memory architectures", *Symposium on optimal algorithms*, H. Djidjev (Ed.) Springer-Verlag (1989)

MASSIVELY PARALLEL SYSTOLIC PROCESSORS FOR HIGH-SPEED RECOGNITION OF SIMPLE PATTERNS

R. MÄNNER

*Chair for Computer Science V, University of Mannheim,
A5, D-6800 Mannheim, Germany,
Interdisciplinary Center for Scientific Computing, University of Heidelberg,
Im Neuenheimer Feld 368, D-6900 Heidelberg, Germany*

J. GLÄß

*Chair for Computer Science V, University of Mannheim,
A5, D-6800 Mannheim, Germany,*

F. KLEFENZ

*Chair for Computer Science V, University of Mannheim,
A5, D-6800 Mannheim, Germany,*

ABSTRACT

This paper describes two systolic array processors, which have been designed for high-speed pattern recognition. Both operate on pixel images and execute the recognition process within 5 μ s to 20 μ s. The patterns which are recognized are simple geometrical figures, namely ill-defined circles and lines of a certain curvature. Both processors consists of one processing element (PE) per pixel. In the first one 28,160 PE's operate on a 176x160 image. It is assembled of 22x20 custom VLSI chips containing 8x8 PE's each. The processor is operation at CERN, Geneva, since spring 1991. The second processor finds and counts in an image all circle segments with a common vertex point within 10 μ s. It is based on a Hough transform that maps the images polar coordinates onto the coordinates radius of curvature and starting angle of a track, and on the Euler relation. The systolic array executing the Hough transform is composed of sixteen X3090 chips (Xilinx). The Euler processor has been mapped into a single X3042 chip.

1. Introduction

Systolic array processors^{1,2} belong to a new class of pipelined array architectures where many identical processing elements (PE's) are interconnected locally so that data can be passed from all PE's to their respective neighbors synchronously and in parallel. In principle, all of them perform the same basic operation on their current operands in one clock cycle. Due to their regular structure and the locality of their interconnection topology, systolic processors are particularly well suited for VLSI implementation. Since Kung and Leiserson presented the basic systolic architecture in 1978¹, systolic array processors have been applied extensively for signal processing and pattern recognition tasks.

Many pattern recognition tasks can be solved by mapping the input data to a feature space in which a feature vector indicates a set of feature attributes. These attributes span a n-dimensional coordinate system for a number n of attributes. The coordinate transform can often be executed by a systolic array of identical PE's which operate in parallel on n data streams that are synchronized by some sequential ordering parameter. The coordinate transform can therefore be expressed by a kind of a space-time equation and can be formu-

lated as a lock-step process, i.e., a data flow scheme on a clock cycle basis. Here the data arrive just-in-time at the right place and where the right operation is executed. The input data are assumed to be asserted in the right sequential order. This has to be assured by appropriate preprocessing where necessary.

In this paper we describe two systolic array processors which have been designed for high-speed recognition of simple patterns. Both operate on pixel images and execute the recognition process within 5 μ s to 20 μ s. The patterns which are recognized are simple geometrical figures, in the first case ill-defined circles, in the second case curved lines. Both processors are trigger processors for high energy physics experiments. The first one detects rings in Ring Image Cherenkov (RICH) detectors, the second one tracks in wire chambers. Below both systems are described in some detail.

2. Recognition of circles

The first systolic processor described here is used as a programmable processor for the recognition of Cherenkov rings in a RICH detector in the CERES/NA45 experiment³ at CERN. The detector outputs continuously images of 176 $\times$ 160 pixels which have to be analyzed in respect to the conditions described below. Every image has to be analyzed completely within 20 μ s.

Due to the short time available massive parallelism is exploited by assigning one PE to every pixel. The CERES trigger processor therefore consists of a systolic array of 176 $\times$ 160 PE's. It is controlled by a sequencer that can be programmed freely via VME bus. It operates at 45 MHz during data read-in and at 30 MHz during data processing. The CERES trigger processor has been build up and tested completely. It is in use at CERN since May 1991.

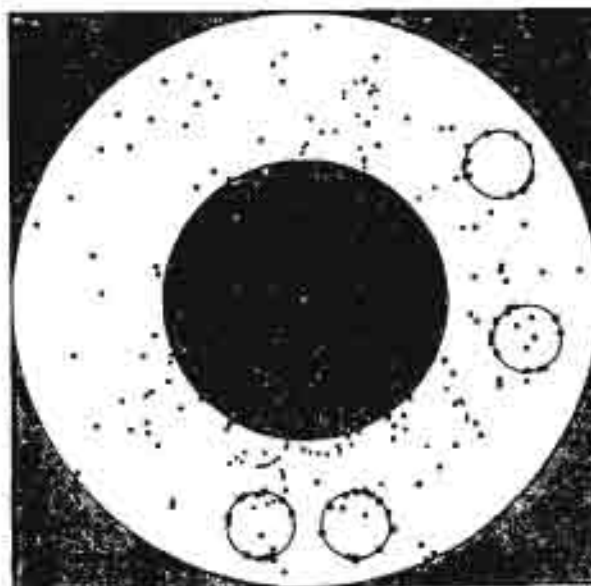


Fig. 1: Typical image generated by a RICH detector. The circles searched for are indicated.

2.1. The trigger problem for the CERES experiment

The CERES/NA45 experiment investigates the production of e^+e^- pairs. These and an enormous amount of secondary particles are detected via ill-defined circles that are generated by the small number (10 to 20) of photons in the Cherenkov cone around the direction of the particles. A typical detector image is shown in Fig. 1. The radius of these Cherenkov rings depends on the particle type. Unfortunately most of the ring pairs are (for physical reasons) uninterestingly "close" and have to be suppressed. Interesting images contain at least one "open" electron pair, i.e., two rings with a certain minimum distance. For each image it is therefore required

- to identify all relatively well defined rings of a given diameter,
- to drop all rings which do not have a certain minimal distance, and
- to count all remaining rings.

2.2. Principal operation of the CERES trigger processor

The identification of rings is achieved by a two-dimensional correlation of the input image with a mask image (a ring of the nominal diameter). The correlation result is computed in parallel by shifting the input image over a counter plane along such a ring and by incrementing in each step all counters that are covered by pixels in the input image. The result of this correlation is a two-dimensional distribution whose analog values $f(x,y)$ represent the probability to find the ring center at the position (x,y) of the input image. In the correlation result, high peaks correspond to well defined rings.

The next step consists of finding sufficiently high peaks, i.e., the position of sufficiently well defined rings. Due to binning effects, proper thresholding followed by a thinning operation has to be used to locate their positions, i.e., to compute a single pixel per peak. The new intermediate image generated in this way consists of individual pixels representing the remaining centers of well defined rings.

To drop all rings that are too close together, a second correlation step is done. Here all ring centers with a distance smaller than the required minimum have to be deleted. To check all mutual distances in parallel, this intermediate image is correlated with a circular disk with a radius equal to the minimum distance. In this two-dimensional correlation result, all pixels (x,y) with a function value $f(x,y) = 1$ correspond to "isolated" ring centers which do not have neighbor pixels within the minimum distance. By deleting all pixels with a function value >1 , a result image is formed that contains pixels only at positions where an isolated ring with the radius searched for was found in the detector image.

The generation of the trigger signal requires to determine whether at least two ring centers remained. This is done in the last processing step by shifting the result image out of the systolic array into an adder.

For the whole trigger process 543 sequencer steps, i.e. 543 clock cycles, are necessary. With a frequency of 30 MHz the trigger decision is computed in 18,1 μ s. A detailed description of the pattern recognition process is given elsewhere^{4,5}.

2.3. Overview of the CERES trigger processor

The complete CERES trigger processor consists of four parts, a discriminator that preprocesses the detector data, a systolic processor array that filters out rings with a minimum distance, a sequencer that stores the micro-program and controls the processor array, and a pseudo adder array that determines whether there were at least two rings with a minimum distance. In the following we restrict on the systolic processor array.

It consists of one PE per pixel in the 176×160 image, i.e., of 28,160 PE's. This processor array is assembled of 22×20 custom VLSI chips where each one contains 8×8 PE's (Fig. 2). These semi-custom chips have been designed on a VENUS system⁶ in $2 \mu\text{m}$ CMOS standard cell technology. The VLSI implementation of these chips is described elsewhere⁴. The PE's are arranged as a grid with connections from/to the next four neighbors.

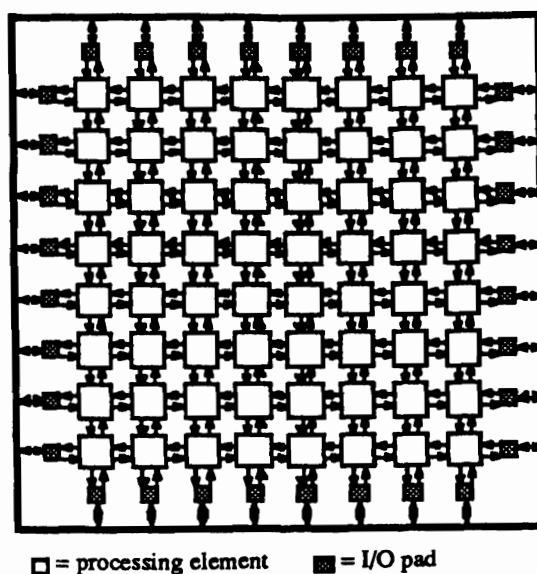


Fig. 2: Internal structure of the VLSI chip.

These chips are assembled on four boards of 11×10 chips each. Topologically, the PE's are interconnected as a cylinder to allow shifting of the stored image along one axis without boundary problems. Thus it is possible to read out the processor array without losing data by shifting the image once around the cylinder. The 160×176 identical PE's are controlled by a sequencer that asserts identical control signals onto all VLSI chips.

2.4. The processing element

Each PE (Fig. 3) contains a pixel flip-flop that stores one of the pixels of the image to be processed. These flip-flops are first loaded with the detector image. During processing the contents of the pixel flip-flops are modified to represent intermediate images or the result image. In addition each PE contains a mask flip-flop to store a mask image and a 6-

bit counter to determine the correlation results. The operation of the counter is controlled by four "counter enable select" signals and by the contents of the pixel flip-flop.

To allow two-dimensional image shifting, each PE has I/O lines from/to the next four neighbors. At each clock cycle, the current state of each data input or internal information can be selected via a data multiplexer and loaded into the pixel flip-flop or the mask flip-flop. If the output of the pixel flip-flop is selected as the internal input, the current image is held without processing. By selecting the logical AND or OR of the output of the pixel flip-flop and the counter comparison output, simple image processing operations can be executed. The 8 possible inputs are selected by a 3-bit function code that switches the input data multiplexer. Both flip-flops and the counter are clocked continuously by the system clock.

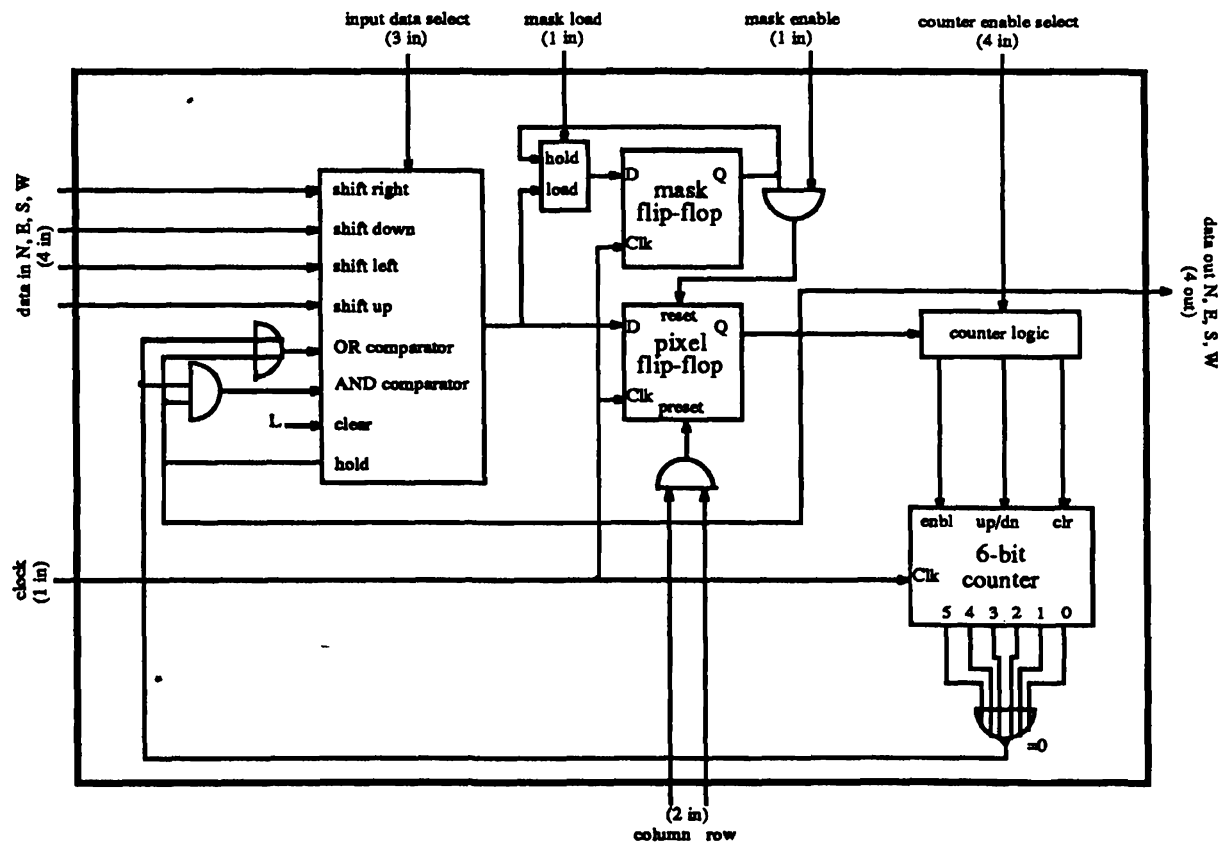


Fig. 3: Processing element.

2.5. System features

The systolic processor described here is a very powerful device. In our application 28,160 PE's are used in parallel. At its operation frequency of 30 MHz this setup represents a computing power of 844,800 million operations/s where one operation is typically a parallel shift-and-increment or a low-level image operation (AND, OR, etc.).

All processing steps can be programmed freely. It is therefore possible to search, e.g., for rings with other diameters. Even the exact form of the object searched for can be

chosen arbitrarily. Also the neighborhood for the peak detection can be programmed as needed as well as the minimum distance between valid objects and the minimum and maximum peak heights. The size of the processor array can be adjusted by adding or removing individual boards or chips.

3. Recognition of curved lines

The second systolic processor described here finds and counts in an image all circle segments with a common vertex point. It is designed to identify particle tracks in the OPAL wire chamber at CERN⁷ which originate from the interaction zone and are given as circle segments of an arbitrary starting angle and a minimal radius of curvature. The complete operation has to be finished within 10 μ s.

As discussed below, the pattern recognition process consists of four main steps,

- preprocessing of the detector data for adaption to the systolic architecture,
- a Hough transform that maps the detector coordinates in the (r, ϕ) -plane onto new coordinates in a $(1/r_c, \phi_s)$ -plane where r_c is the radius of curvature and ϕ_s is the starting angle of a track,
- thresholding the transform result, and
- applying the Euler relation to compute the number of peaks in the resulting image.

The system therefore consists of a preprocessor which is not described here, a Hough transform processor that determines well defined tracks, and an Euler processor that counts their number. For the latter two processors a systolic architecture is presented that is implemented in field programmable gate arrays.

We first give a description of the operation principle. This is followed by a description of the architecture of the processor.

3.1. Principle of Operation

The OPAL detector consists of 24 sectors which operate independently of each other. Each track i is defined by a set of track points $(r, \phi)_i$. Because interesting tracks come from the origin, each one is determined by its starting angle ϕ_s and its radius of curvature r_c . Together with the origin every pixel (r, ϕ) in the detector image specifies a class of tracks with different ϕ_s and r_c ⁸. From Fig. 4 one can derive the relation between (ϕ_s, r_c) and (r, ϕ) as

$$\frac{r/2}{r_c} = \sin(\phi - \phi_s) \quad (1)$$

The histogramming method for identifying particle tracks used is based on accumulating the number of pixels that vote for every possible track (ϕ_s, r_c) . As can be seen from Eq. (1), each pixel (r, ϕ) is mapped onto a sine curve with amplitude $2/r$ and a phase shift of ϕ . This mapping is a special type of a Hough transform. Applying this transform to every

detector pixel yields one curve per pixel in the $(1/r_c, \phi_s)$ -plane (Fig. 5). Pixels belonging possibly to a single track with parameters $(1/r_c, \phi_s)$ intersect at this point. The result of this Hough transform therefore gives a probability measure for the existence of all possible tracks.

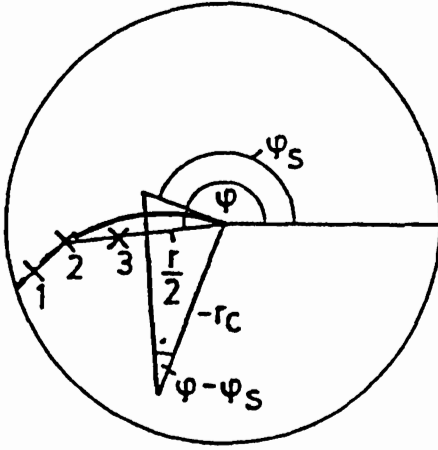


Fig. 4: Relationship between (r, ϕ) and $(1/r_c, \phi_s)$.

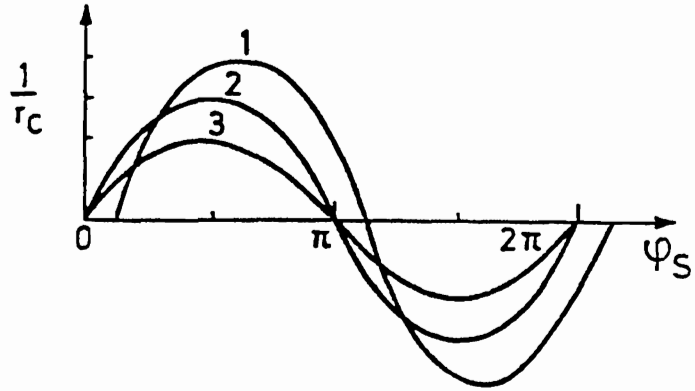


Fig. 5: Hough space.

3.2. The Hough transform processor

The Hough transform corresponds to the mapping described by Eq. (1). For every input pixel it requires to increment all bins of an accumulator plane which are covered by a certain sine curve. If all detector pixels had to be handled in parallel, all possible sine curves, i.e., sine curves with all possible combinations of amplitudes and phase shifts, had to be used in parallel. Obvious this is not feasible. Instead processing in the ϕ coordinate is serialized and only the r coordinate is handled in parallel. By doing so only one class of sine curves - called mask below - with different amplitudes but fixed phase shift has to be handled at every instant t . This kind of serialization is suggested by the time-of-flight determination of ϕ in the OPAL detector (pixels with a larger distance to the center plane are read out later due to the detector set-up). By appropriate preprocessing all detector pixels with a certain ϕ , i.e., up to 160 r coordinates, are supplied in parallel and handled by the Hough transform processor in one cycle.

The essential operation is to generate for each r_i the corresponding sine at the right place in the $(1/r_c, \phi_s)$ -plane at activation time t . Now instead of shifting the mask synchronously with ϕ over the accumulator plane, it is more appropriate to shift the contents of the accumulators synchronously with ϕ column by column in the opposite direction. In this case the mask can be held fixed and every line of it will be activated individually. After all ϕ are handled sequentially, the Hough transform has been computed.

3.3. VLSI realization

One design goal was to achieve a highly regular PE structure, because this normally implies an uniform floor-plan and a scalable processor architecture.

Because the detector is assembled of 48 half-sectors that are handled individually, only ϕ values from 0 to 7.5 degrees have to be processed so that the sine curves can be approximated by straight lines. On the accumulator plane, these lines are generated by incremental moves in upward and sideward direction. Even if sideward moves are restricted to one step only because the contents of the accumulator array are shifted one column a time, all required lines can be realized by an appropriate choice of upward moves. For each r_i value there exists one column which is responsible for producing this digitized representation of the sine with amplitude $2/r_i$.

The singular point of this mask at $(1/r_C, \phi_S) = (0,0)$ has been avoided by shifting successive lines one pixel each on the $1/r_C$ axis. To ensure proper updating in the accumulator plane the introduced spatial offset has to be compensated through the appropriate temporal offset. This temporal offset, given by r_i , is known in advance and is handled correctly in the preprocessing step.

The Hough processor is subdivided into two building blocks. First, it consists of a 36×32 array of counter elements. The array tiling is oriented in rows. The counter elements in each row are interconnected horizontally to their two nearest neighbors and can transfer the actual counter value to their right neighbors so that each slice forms a unidirectional 36-stage counter pipeline. The second part is a 32×36 activation matrix of the counters. This time the activation matrix tiling is oriented in columns. The 32 parallel countenable lines of each column are bundled and connected to an activation unit for each column. Each activation unit creates the right sequence of upward moves for one column. It is realized by a serial shift register. The serial bit stream is internally transformed into the activation of PEs with a fixed mapping of each stage in the serial shift register to the corresponding activation lines.

3.4. The Euler Processor

The Hough transform processor outputs a 2D histogram which contains peaks at those positions in the $(1/r_C, \phi_S)$ -plane that correspond to well defined tracks. By setting a global threshold in the 2D histogram a binary image is created that contains clusters of adjacent pixels at these positions. For a trigger decision these clusters have to be detected and counted. Counting of the clusters can easily be done with the help of the Euler relation:

$$\text{connectivity number} = n \cdot \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} - m \cdot \begin{bmatrix} x & 1 \\ 1 & 0 \end{bmatrix}.$$

The connectivity number is computed as the difference of how often the two 2×2 patterns appear in the image. Since the operation is fully local this updating can again easily be

done in parallel by a systolic processor. The binary image that has been created by thresholding is shifted into the pipeline one row at every time step in parallel. Each PE sets the registration line "n" to high if the left pattern is detected or line "m" to high if the right pattern is detected. The pattern registration is accomplished by simple logic elements. Subsequently an accumulator sums up the number of occurrences of n and m.

3.5. Implementation with field programmable gate arrays

The systolic array is completely composed of Xilinx chips. The mapping of the PEs to these chips is dictated by the row-oriented data flow and the vertical control flow. Two counter pipelines of 36 stages can be mapped efficiently into one X3090 chip. The counter stages are cascaded in one 1-bit, two 2-bit, four 3-bit, eight 4-bit, sixteen 5-bit, and five 6-bit counters without causing overflow because the needed bit depth is given by the column number. To speed up the counters only one-level logic was adopted since delays between logic blocks slow down the operation significantly. Up to the fourth stage a one-level logic for the binary counter is achievable after logic optimization. The 5-bit counter is composed of a 3-bit counter with carry and a 2-bit counter with carry-in. The 2-bit counter is delayed one time step and lags behind as to await the possibly occurring carry from the lower stage. The output is then resynchronized in adding a second delay at the end of the counter pipe in the lower section. To cover the whole array for one detector sector sixteen X3090 chips are needed.

The counter array is activated by 1152 count enables lines. These are decomposed into 18 X2018 chips with 2×32 output enable lines. Each X2018 chip incorporates two serial shift registers with 2 serial-in's and 2×32 output-enable lines.

The whole program is stored in the flow pattern of the serial shift registers. Since all other parts are uniform and static components which never have to be redefined this programming step has been automated. The control flow pattern defined in the simulation is directly converted to logic equations, the Xilinx programming cycle is initiated, and the resulting bit stream downloaded to the Xilinx chips. So the array can be adapted to a new task in several minutes.

The Euler processor has been mapped into one X3042 chip whose I/O pins are fully used. It consists of 32 pattern matching units analyzing one column at a time in parallel. A second implementation of the Euler chip is cascadable in both directions, so that larger arrays of size 2^n ($n \geq 5$) can be handled in parallel.

3.6. System features

Also this second systolic processor is very powerful and flexible. It analyzes within 10 μ s an image and delivers the number of all well defined tracks, their corresponding radius of curvature and their starting angle. The first prototype (for one detector sector) consists of 36×32 PE's that are implemented in field programmable gate arrays (1.08 μ m technology) with clock rates of 30 MHz. Using these chips the system can be tailored and

customized to a broad class of special configurations. The architecture is flexible, scalable, reconfigurable and reprogrammable. The adaption process can be automated by a program which directly accepts the program data extracted from simulation, converts them to logic equations, and starts the conversion process. The output bit stream will then be downloaded to the Xilinx chips and the array is reconfigured and run in the new operation mode.

4. Conclusions

The two systolic arrays described here are special-purpose processors for low-level pattern recognition tasks. Due to the extreme speed requirements given, massive parallelism has been exploited by assigning one PE to every pixel of the images to be processed. Due to the high regularity of systolic arrays and the relative simple structure of the PE's a VLSI implementation could be done easily.

In the first case we developed a chip in standard cell technology. From the detailed circuit design up to the test of the first samples only one man-year had to be invested. Most of the time has been spent for optimizing the PE for high-speed operation and dense layout. The first samples fulfilled their specifications completely. It took several months more to set up the whole processor and to adjust all timing requirements on the 1 m^2 area which is completely filled with pin grid arrays operating synchronously at 45 MHz. This task would have been considerably easier with a full-custom chip in $\leq 1\text{ }\mu\text{m}$ technology. In this case, it would probably have been possible to pack 16×16 PE's into one chip, reducing the processor size to about $50 \times 50\text{ cm}^2$ and allowing an even higher operating frequency.

In the second case we avoided the chip design process by applying programmable gate arrays. From the installation of the design software to the simulation of the first chips only two man-months had to be invested. Again, most of the time has been spent for optimizing the PE's for high-speed operation and dense layout. Since no chips have been programmed yet we are not sure about the final operation frequency. It should be comparable, however, to the custom design chips of our first processor. However, the much smaller layout density makes this technology less attractive for massive parallel processors in case enough time and funding is available for a full custom design.

5. References

1. H.T. Kung, C.E. Leiserson, Systolic arrays (for VLSI); Proc. SIAM Sparse Matrix Symposium, 1978, p. 256.
2. H.T. Kung, *IEEE Computer*, Vol. 15, pp. 37-46, 1982
3. U. Faschingbauer, M.G. Trauth, J.P. Wurm, A. Drees, P. Fischer, P. Glässel, M. Guckes, D. Irmscher, L.H. Olsen, A. Pfeiffer, H. Ries, A. Schön, H. Sickmüller, H.J. Specht, T.S. Ullrich, E.F. Barasch, A. Breskin, R. Chechik, Z. Fraenkel, D. Sauvage, V. Steiner, I. Tserruya, Proposal SPSC/P237; CERN/SPS 88-25 (1988).
4. J. Gläß, R. Baur, R. Männer, *IEEE Trans. Nucl. Sci.* NS-37 (1990) 241.
5. R. Baur, J. Gläß, R. Männer, Proc. Comp. in High Energy Physics '91, Tsukuba

City, Japan (1991), in print.

6. E. Hörbst, M. Nett, H. Schwärtzel, *VENUS - Entwurf von VLSI-Schaltungen*; (Springer, Berlin, 1986).
7. R.D. Heuer, A. Wagner, *Nucl. Instr. Meth. A265* (1988) 11-19.
8. F. Klefenz, R. Männer, *Informatikfachberichte 219* (Springer, Heidelberg, 1989) p. 401.

**ON THE DESIGN OF SYSTOLIC ARRAY ARCHITECTURES FOR
THE ITERATIVE SOLUTION OF LINEAR SYSTEMS**

Roman WYRZYKOWSKI

Computer Science Center, Technical University of Czestochowa
Deglera 35, 42-200 Czestochowa, POLAND

Abstract: The goal of this paper is unfolding the inherent properties of some iterative algorithms to use these properties for the design of optimized parallel architectures implementing these algorithms in a systolic manner. This has been done on the basis of purposive transformations of dependency graphs. The application of such transformations allows us to extend the capability of the existing methods of architectural designing the VLSI arrays.

Systolic arrays are examples of VLSI special-purpose networks that directly implement computationally expensive algorithms in hardware. According to [1], a systolic array (SA) is a synchronous computing network possessing the following features: modularity and regularity, spatial and temporal locality, linear rate pipelinability, and balancing computation with I/O. Such arrays are also used broadly for the high-speed implementation of many computations in linear algebra, an important class of operations that arises [2] in a wide variety of application areas, including modelling and simulation, real-time control, signal and image processing etc. The iterative solution of linear systems is one of these computations.

At present, there exist [1,3-6] many systolic designs, which can be used or have been designed specifically for solving linear systems by the iterative methods. The principle advantage of these methods [7] is a possibility of reducing all the computational process to a repetitive sequence of rather simple steps represented by matrix-vector or matrix-matrix multiplications. This possibility, which may be extended e.g. to solution of eigenvalue problems, creates also a base for the unification of computations in linear algebra. However, the critical analysis of the existing designs have allowed us to conclude that they do not use all the potentiality of algorithms to optimize the hardware-time cost of SA's.

Thus, the classical two-directional array [3] for the matrix-vector multiplication consists of $2N-1$ processing elements (PE's) performing multiply-adds. When the use of the Jacobi or Gauss-Seidel algorithm is assumed, this SA does not permit to solve a system

$$A \bar{x} = \bar{b} \quad (1)$$

faster than in $T_c = 2KN$ cycles, where N is the order of matrix A , and K is the number of iterative steps. This leads to the low processor utilization $g_c \approx 1/4$. Another linear SA [5], which consists of only N PE's, executes K iterative steps in $T_f = K(N+1)$ time, so that $g_f \approx 1$. However, in this array, any element of the vector $\bar{x}^{k-1}$ of unknowns is preloaded, and stays in the corresponding PE during the k -th iterative step, where $k = 1, 2, \dots, K$. This results in an infringement of the temporal locality, since a global communication channel between the array output and inputs of all the PE's is needed. To minimize the execution time without using any global link, an array with fully localized communications was designed recently [6] on the basis of a cooperative execution of any two neighbour steps. The proposed solution, which contains $3N-1$ PE's, enables to solve system (1) in $T_l = N(K+5)$ cycles, but this is reached at the cost of increasing the amount of hardware, so that $g_l \approx 1/3$.

The goal of this paper is unfolding the inherent properties of some iterative algorithms in order to use these properties for the design of optimized parallel architectures implementing these algorithms in a systolic manner. The procedure of architectural designing takes as a basis our matrix-graph approach described e.g. in [8,9]. This approach relies on the space-time mapping of an algorithm presented in the form of a dependence graph (DG), and allows us to construct the optimized structural scheme $C = \langle S, T, U \rangle$ of a SA, where S - directed graph called the array structure, T - synchronization function, U - set of operation algorithms of PE's. For this aim, the DG, which is determined in an integer lattice, is subjected to an injective and monotonous mapping F containing a processor assignment F_s and a schedule mapping F_t . The procedures for both the DG derivation from a high-level language description, and the determination of the space-time mapping F were detailed in [8,9]. In this paper, we will concentrate on considering results of these procedures, i.e. structural schemes implementing the iterative algorithms.

Also, we will pay a particular attention to the problem of DG transformations, whose application is supposed to be a logical development of the approach described above. The importance of these transformations stems e.g. from the fact that any value of the

mapping F must lie within the area of such parallel implementations of a certain algorithm that are allowed by its DG. Hence to obtain systolic architectures with new combinations of characteristics, it is desirable to extend this area by purposive generating more than one DG for an initial description of the algorithm. For this aim, we can use existing a wide choice in the order of transferring the input data between nodes of the DG. Moreover, even for those data which are generated during computations, we might change the order of their generation, using the commutative and associative properties of mathematical operations. However, this must not jeopardize the correctness of results of the algorithm, taking also into account properties of these operations in computer systems (e.g. roundoff errors).

Thus, many algorithms of the iterative solution of system (1) with nonsingular matrix A can be reduced [7] to the following iterative process:

$$\bar{x}^k = P \bar{x}^{k-1} + \bar{f}, \quad k = 1, 2, \dots, K, \quad \bar{x}^0 = \bar{x}. \quad (2)$$

This applies e.g. to such well known algorithms as the Jacobi and Gauss-Seidel ones. For instance, for the former algorithm, we have

$$P = -D^{-1}(L + R), \quad \bar{f} = D^{-1}\bar{b},$$

where L , D and R are subdiagonal, diagonal and superdiagonal components of the composition $A = L + D + R$.

As a matter of fact, each iterative step of process (2) is a matrix-vector multiplication. After localizing all the variables, the DG of this matrix algorithm is known [1] to be the simplest planar lattice graph with $N \times N$ nodes and the orthonormal arcs $\bar{d}_1 = (d_1^1, d_1^1) = (1, 0)$, $\bar{d}_2 = (0, 1)$. Each of these nodes corresponds to a multiply - add. This graph is shown in Fig.1 ($N = 4$), where the horizontal arcs correspond to transferring elements of the input vector $\bar{x}^{k-1}$, while the vertical ones describe the generation of elements of the output vector $\bar{x}^k$.

The properties of this basic graph G_b are just the reason of deficiencies of all the SA's based on it. Not considering superfluous details, we note only that if the iterative steps, each with the graph G_b , are implemented in the same way on a SA without any global interconnection, than process (2) can not be executed faster than in $T_{B,K}^{\min} = L_{B,K}^{\min} = K(2N-1)$ cycles. The obtained time bound is determined by the length $L_{B,K}$ of the critical paths in the resulting

DG of process (2). This length takes into account not only the computational nodes, but also those nodes that are necessary to organize I/O of data from one source or/and to one receiver. The execution of any two neighbour steps in different but cooperative manners [6] also does not permit to avoid some deficiencies. In fact, this allows the authors to minimize the execution time $\bar{T}_{B,K}^{\min} = \bar{L}_{B,K}^{\min} = N(K+5)$ cycles, but leads to increasing the width W of the resulting DG, since $W = 3(N-1)$. As a result, they have to use W PE's, so that the processor utilization is still low.

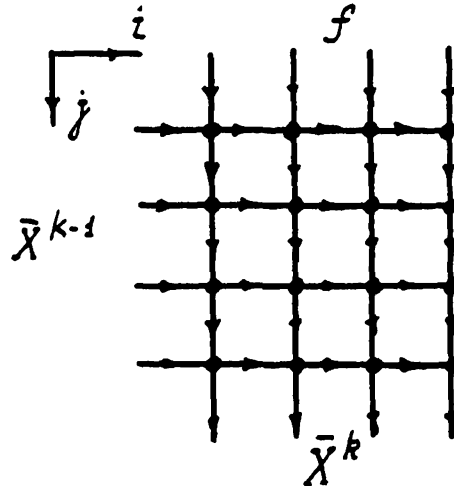


Fig.1. Basic graph G_B for one iterative step

At the same time, the great potentiality to overpass this embarrassing situation lie in some purposive transformations of the basic graph. The simplest variant of them is joining the nodes in such a way that any two neighbour nodes $\bar{K}_1 = (i, j-1)$ and $\bar{K}_2 = (i, j)$ of the graph G_B are replaced with the node $\bar{K}^* = (i, j^*) = (i, j/2)$ of a new graph G_B^* , where $j = 2, 4, 6, \dots, N$.

The resulting graph G_B^* with $N * N/2$ nodes is depicted in Fig.2a ($N=6$) as a processing subgraph. This subgraph is then supplemented by an input one G_{IN} to provide the pipelining input of all the elements of the vector $\bar{x}^{k-1}$ from one external source. The procedure of constructing the G_{IN} subgraph takes into account, on the one hand, the full overlapping (Fig.2b) of two following DG's. The first one is that subgraph G_B^* which describes computing the vector $\bar{x}^k$ on the k -th step. The second DG corresponds to input operations arising when $\bar{x}^k$ is used as an input vector on the $(k+1)$ -th step. On the other hand, we start with the space $F_S^1(\bar{K}^*) = j^*$ and time $F_I^1(\bar{K}^*) = i^* + j^*$ mappings, which are optimum when the number of PE's and execution time are considered.

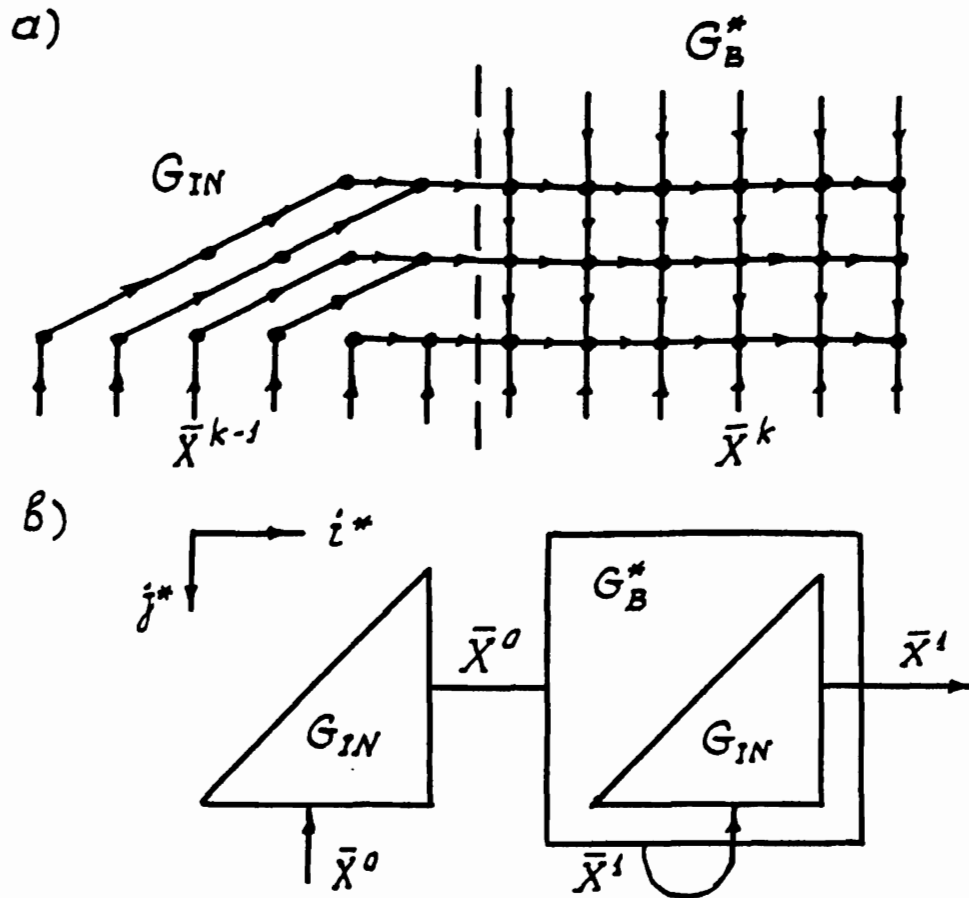


Fig.2. Graphs G_B^* and G_{IN} (a); resulting graph for process (2) (b)

The resulting structural scheme C_1 , which corresponds to the DG of Fig.2b, is a linear SA with a data feedback. This array is shown in Fig.3 ($N=4$), where the comparer CM handles vectors $\bar{x}^{k-1}$ and $\bar{x}^k$ to determine when process (2) should be finished. The actual vector $\bar{x}^k$ is derived from the previous $\bar{x}^{k-1}$ one in the following way.

On computing by the last $R_{N/2}$ adder of the upper layer, an element of the vector $\bar{x}^{k-1}$ is injected into a distributed shift register composed by $N/2$ registers R . The DSR together with N registers R_i^* provide transferring the i -th element x_i^{k-1} of the input vector $\bar{x}^{k-1}$ to the first input of a multiplier M_i , whose the second one accepts elements of the i -th row of the matrix P . This transferring is implemented in such a way that during the first $N/2$ cycles after computing x_1^{k-1} on cycle s_1^{k-1} , only the shift of data between registers R takes place, without fixing these data in registers R_i^* . But during the subsequent $N/2$ cycles such fixing is also provided. This is organized in a sequential manner, starting from a couple of the registers R_1^*, R_2^* on cycle $s_{s,1}^k = s_1^{k-1} + N/2 + 1$, and finishing by a couple of R_{N-1}^*, R_N^* on cycle $s_{f,1}^k = s_1^{k-1} + N$. As a result, each register couple holds its data during N cycles.

Hence the computation of the first element of the vector $\bar{x}^k$ starts on cycle $s_{s,1}^k$, involving the multipliers M_1, M_2 and adders R_1^*, R_1 . The computation lasts $N/2$ cycles, and finishes on cycle $s_{f,1}^k$ when the multipliers M_{N-1}, M_N and adders $R_{N/2}^*, R_{N/2}$ are involved. The rest of elements of the vector $\bar{x}^k$ are generated in the same but shifted way.

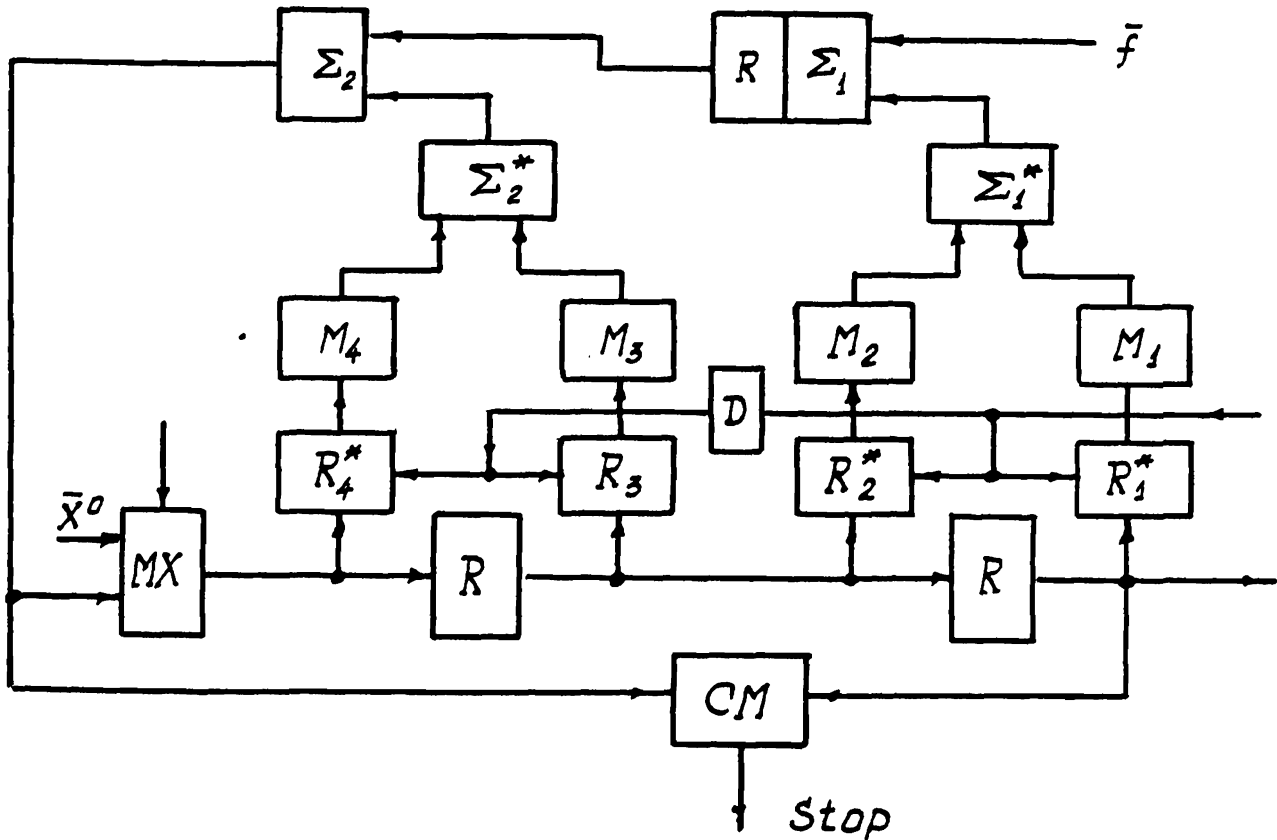


Fig.3. Structural scheme C_1

Summarizing, we conclude that our linear SA, which consists of N multipliers, N adders, and $2N$ delays only, allows us to execute process (2) with almost full processor utilization in $T_G = N(K+1)$ time, without using any global interconnection. Any of the known arrays does not permit to implement process (2) so effectively.

Of course, the transformation described above does not exhaust the variety of purposeful transformations of the basic graph. The use of the commutative and associative properties of addition is especially promising, since by this way we can change not only the order of transferring the input variables x_i^{k-1} , but also the order of generating the results x_i^k . A certain graph G_c , which has been obtained from the graph G_b by such a transformation, is depicted in Fig.4a ($N=4$), where indices i, j of nodes of the graph G_b are shown near nodes of the new graph. As before, the vertical arcs $\bar{d}_1^{-1} = (0,1)$

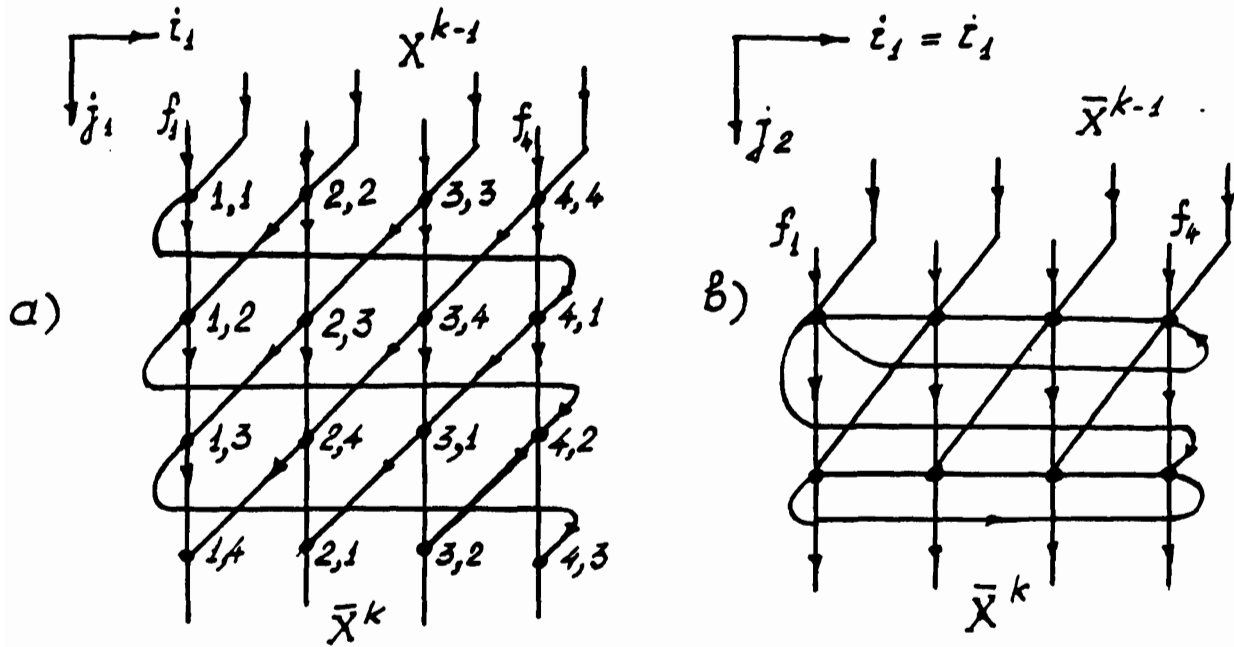


Fig.4. Graphs G_C (a) and G_C^* (b)

of the graph G_C correspond to computing the results x_i^k , while transferring the input data x_i^{k-1} is described by the arcs $\bar{d}_2^1 = (-1, 1)$ and $\bar{d}_3^1 = (N-1, 1)$. The important feature of the graph G_C is decreasing the length $L_C^{\min} = N$ nodes of the critical paths, as compared with $L_B^{\min} = 2N-1$ nodes for the graph G_B .

Basing on the graph G_C , we can design a ring-connected array, which corresponds to projecting the graph along the direction $\bar{r} = (0, 1)$, and consists of N PE's performing multiply-adds. This array also allows us to reach almost full processor utilization. But manifesting almost the same same performance as the scheme C_1 , it needs more hardware. In connection with this, we move on to the procedure of joining the nodes of the graph G_C . As before, we will join those neighbour nodes which lie along the direction of generating the results x_i^k , although the direction of transferring the input variables x_i^{k-1} may be also used. But it can be proved that these two approaches are equivalent. It should be also noted that when the latter variant of joining is applied to the basic graph, this results in another linear SA which, however, required more hardware than the scheme C_1 .

Thus, using the first variant of joining the nodes of the graph G_C , we construct a new graph G_C^* (Fig.4b) with $N * N/2$ nodes. In the graph G_C^* , the generation of any result x_i^k corresponds to the arc $\bar{d}_1^2 = (1, 0)$, while transferring each x_i^{k-1} is described by a set of $N-1$

elements; any of them is either the arc $\bar{d}_2^2$ or arc $\bar{d}_3^2$, where

$$\bar{d}_2^2 = \begin{cases} (-1,0), & \text{if } i_2^s \geq 1 \\ (N-1,0), & \text{if } i_2^s = 1 \end{cases}, \quad \bar{d}_3^2 = \begin{cases} (-1,1), & \text{if } i_2^s \geq 1 \\ (N-1,1), & \text{if } i_2^s = 1 \end{cases}.$$

Here i_2^s denotes the value of index i_2 corresponding to a node, in which the arc $\bar{d}_2^2$ or $\bar{d}_3^2$ originates.

The nature of the information dependencies described above allows us to construct the resulting DG of process (2) by linking K graphs G_c^* immediately. Then, we subject the resulting DG to the space $F_s^2(\bar{K}) = i_2$ and time $F_1^2(\bar{K}) = j^*$ mappings that are optimum when the number of PE's and execution time are considered, where $\bar{K}^* = (i_2^*, j_2^*)$; $i_2^* = i_1^* = 1, 2, \dots, N$; $j_2^* = 1, 2, \dots, KN/2$. This results in the structural scheme C_2 of a ring-connected semisystolic array, which (without comparer) is shown in Fig.5a, where each delay on data paths is depicted by a circle. The designed array consists of N PE's (Fig.5b), each of them performs two multiplications and two additions. Besides of PE_1 and PE_2 , all these PE's function according to the following algorithm:

$$\begin{aligned} s &:= \text{if } c_1 = 0 \text{ then } x_{in} \text{ else} \\ &\quad \text{if } c_1 = 1 \text{ then } x_{in}^k; \\ u &:= \text{if } c_1 = 0 \text{ then } x_{in}^k \text{ else} \\ &\quad \text{if } c_1 = 1 \text{ then } f; \\ x_{out}^k &:= u + s * p_1 + \hat{x}_{in}^{k-1} * p_2; \\ \hat{x}_{out}^{k-1} &:= s; \quad \hat{x}_{out}^{k-1} := \hat{x}_{in}^{k-1} \end{aligned} \quad (3)$$

Here c_1 is a control signal for choosing modes of operation of all the PE's simultaneously. This is organized in such a way that for $c_1 = 0$ these PE's carry out a certain iterative step, and for $c_1 = 1$ they realize the transition from one step to another. The algorithm of functioning PE_1 or PE_2 , which are responsible for input of odd or even elements of the vector $\bar{x}^0$ respectively, differs from algorithm (3) only by setting $s := d_{in}$ when this input is carried out (at the same time, the rest of PE's accept $c_1 = 0$).

It can be proved that taking also into account I/O operations, the ring-connected semisystolic array, corresponding to the scheme C_2 , executes process (2) in $T_R = N(K+2)/2$ time, that means increasing the performance about twice as compared with the scheme C_2 . This advantage has been reached at the cost of increasing twice the

amount of processing hardware (i.e. multipliers and adders), without changing the amount of registers.

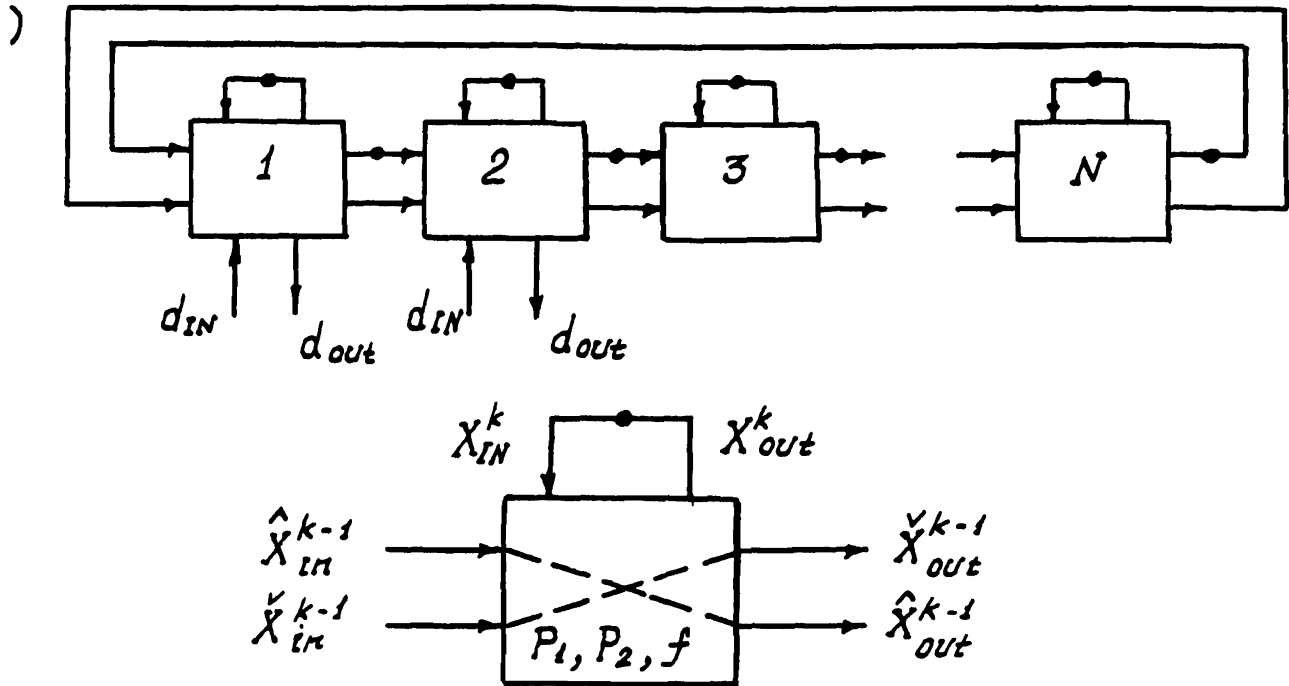


Fig.5. Structural scheme C_2 (a), and its PE (b)

Unlike the graph G_c^* , we can join more than two neighbour nodes of the graph G_c . However, as the number m of these nodes is growing, the proportional decrease of the execution time is counterbalanced, on the one hand, by proportional increasing not only the amount of processing hardware, but also the number of communication channels. On the other hand, the duration of the clock period is increasing because of necessity for summing $m+1$ operands during one clock cycle. Moreover, the undesirable infringement of the locality takes place, since these operands are computed with participation of some data distributed within m neighbour PE's. Therefore, our paper does not deal with $m > 2$, although in this case the design of structural schemes has nothing special as compared with the scheme C_2 .

All the results described above can be immediately extended to the iterative solution of linear systems with the following form:

$$A X = B. \quad (4)$$

Being more general than system (1), this form corresponds to the case of one matrix A of coefficients and different right-hand side vectors $\bar{b}_j$, each of them generates a vector $\bar{x}_j$ of unknowns, where $X = \&\bar{x}_j^*$, $B = \&\bar{b}_j^*$, $j = 1, 2, \dots, M$. System (4), in its turn, generates the following iterative algorithm:

$$\mathbf{x}^k = \mathbf{P} \mathbf{x}^{k-1} + \mathbf{F}, \quad \mathbf{F} = \mathbf{A} \bar{\mathbf{f}}_j^*, \quad (5)$$

which is a natural generalization of process (2). Process (5) can be splitted into M matrix-vector multiplications, each of them involves the same matrix $\mathbf{A}$ and the j -th row $\bar{\mathbf{x}}_j^{k-1}$ of the matrix $\mathbf{x}^{k-1}$ of unknowns. Taking this possibility into account, there is no necessity to accomplish all the procedure of architectural designing. Instead of this, we can link M one-dimensional schemes C_1 or C_2 to obtain two-dimensional schemes C_1^T or C_2^T , respectively. For executing process (5), any of these schemes requires only $M-1$ cycles more than corresponding one-dimensional scheme (prototype) needs to execute algorithm (2). This enable to preserve the high processor utilization so characteristic of the prototypes.

Another important extension of the application range of the proposed systolic solutions stems from the desire to provide the ability of large matrix handling by fixed-size arrays. In this case, the partitioning is an effective technique [1]. For processes (2) and (5), the use of this technique will be considered in the next paper. Here we note only that for the scheme C_1 both the locally sequential globally parallel and locally parallel globally sequential methods of partitioning can be applied succesfully, while for the scheme C_2 the former method may be used only.

REFERENCES

1. Kung S.Y. VLSI Array Processors. - Englewood Cliffs: Prientice-Hall, 1988.
2. Nash J.G., Hansen S. Modified Faddeeva algorithm for concurrent execution of linear algebraic operations // IEEE Trans. Comput. - 1988. - No.2. - P.129-137.
3. Kung H.T., Leiserson C.E. Systolic arrays (for VLSI)// Symp. on Sparse Matrix Processing, Philadelphia, 1978.- 1979.- P.256-282.
4. Berzins M., Dew P.M. Systolic matrix iterative algorithms // Int. Conf. Parallel Computing., Berlin, Sept.1983. - 1984.- P.483-488.
5. Johnsson L., Cohen D. Computational arrays for DFT//IEEE COMPCON Spring'81, San Francisco, Dec.1981. - 1982. - P.236-244.
6. Lihoded N.A., Sobolevski P.I. Systolic implementation of some iterative algorithms // Conf. on Homogeneous Computing Media and Systolic Structures, Lvov (USSR), 1990. - P.95-100 (in Russian).
7. Faddeev D.K., Faddeeva V.N. Computational Methods of Linear Algebra. - Moskva: Fizmatgiz, 1963 (in Russian).
8. Wyrzykowski R. Method of designing the specialized parallel digital systems and its application for synthesis of FIR-filter schemes // 5th Symp. Microcomput., Microproces. Appl., Budapest (Hungary), Oct.1987. - 1987. - P.621-631.
9. Kanevsky Ju.S. ,Korchev D.V. A systematic approach to the design of processor arrays (see present volume).

A SYSTEMATIC APPROACH TO THE DESIGN OF PROCESSOR ARRAYS

JU. S. KANEVSKY

Kiev Polytechnical Institute, Pr. Pobedy, 37
2003, Kiev, 252056, USSR.

D. V. KORCHEV

Department of the Technical Diagnostics of the
Ukrainian Branch of the World Laboratory
Department in the USSR,
fl. 1, 26, Moscovsky st., Zhitomir, 262029, USSR

ABSTRACT

A method for synthesis of processor arrays is developed and a system of rules and formulas is suggested which permits the acquisition of a set of alternative structural circuits of processors with arbitrary dimensions. The essence of the technique lies in the search for linear, monotonous and injective S-T-mapping P in time and space, which transforms the graph of the initial algorithm into the structural scheme. The proposed technique consists of constructive procedures which allow us to define full description of the processors' structural scheme, namely: graph structure, exchange synchronization function, input-output sequence and distribution of algorithm variables, as well as, additional information needed to coordinate the processor operations with the outside world. Processor array design does not depend on dimensions of space in which the graphs of the algorithm and the structure are represented.

1. Introduction

For the last years we can see intensive mutual influence between VLSI technology and modern methods of signal processing, where more and more computing resources are desired for growing demands of higher performance, and modern VLSI technology gives a possibility to create cheap parallel computing media with high-levels of homogeneity. Systolic processors, semisystolic processors and processor arrays are representatives of that class of computer networks, which have a kind of an array of simple processing elements (PEs) which work in parallel and which are

mostly of the same type. They have a regular network of local links among them. Such type of array reflects adequate topological and functional features of the algorithm and interaction between data streams in it. Array performance is adjusted due to its input-output channels' capacity. Input data which pass through border PEs are used intensively for pipelined transfers in the PE network ^{1,2} .

On the one hand, architectural peculiarities of the processor arrays imply certain restrictions on the algorithms, for like the demand for local regularity in the initial algorithm. On the other hand, the processor array design procedure is a rather complicated task which cannot be solved on the heuristic level in most of the real situations either due to multibase possibilities of structural-and-algorithmic outcomes, or because it does not provide desired closeness of solution to the hypothetical model.

Therefore, it is necessary to develop methods of formalized structural synthesis of processor arrays simultaneously with the improvement of hardware technology ^{3,4} . The ultimate goal of such methods usually is the design of analytical procedures, which permit us to represent the mathematical and structural features of the algorithm by on adequate structural scheme for the processor. In other words, it is necessary to create an analytical apparatus, which enables us to transfer the mathematical (formal) description of the original algorithm to a structural-and-algorithmic description of the processor performing this algorithm.

2. A Method of Processor Arrays Design

Due to the proposed matrix-graph method (which is the development of methods ^{3,4}) of obtaining a structural scheme, the following major steps ^{5,6,7} are necessary: 1) generation of compact forms of dependence graph G for the initial algorithm A which is based on recurrent equations or a high-level language program; 2) synthesis of linear, monotone and injective space-and-time mapping P of the algorithm graph G which is needed to obtain the structural graph S and PE exchange synchronization function DT for processor array; 3) determination of space

distribution, input-output order for algorithm variables, and desired control signals; 4) synthesis of auxiliary input-output vector for those input and output variables, whose elements are not transferred among PEs during the calculations; 5) designing of PE structural scheme.

According to these procedures, initial algorithm A is represented in the form of a Cartesian product K^n or dependence graph G placed in the linear integer-valued space Z^n , where n is the space dimension. To perform the next steps of the design it is necessary to use not the graph G, but its compact forms such as: set V of all vertices I of the graph G; characteristic matrix $D=[\bar{d}_1]$, which contains vector-columns $\bar{d}_1$ on information dependence between functional operators (FOs) along with every l-th variable of the initial algorithm A; the l-th variable index increment matrix H_1 , which is described in the form of vector-columns $\bar{h}_1$ for increment of the l-th variable's index i; border vector-columns $\bar{d}_{1,b}$ for each l-th variable. According to our method, the synthesis of compact forms of graph G is done on the basis of writing algorithms in a high-level language.

One of the most complicated tasks for mapping n-dimensional algorithms into m-dimensional structures, where $r=n-m \wedge 2$, is the synthesis of a schedule, and getting the PE exchange synchronization function DT from it⁵.

In this case, the m-dimensional graph S of the processor structure is obtained by the mapping graph G onto the space Z^m . This mapping has the form of a matrix expression

$$D_S = P_S D, \quad (1)$$

where a vector of the sub matrix $D_S = [\bar{d}_1^S]$ determines direction and length of interprocessor links in the graph S along with l-th algorithm variable (it is necessary for the conservation of the locality of the exchanges between the PEs to provide that the elements of the vectors $\bar{d}_1^S = [d_{j1}^S]$ satisfy the condition $d_{j1}^S \in \{0, \pm 1\}, j=\overline{1, m}$), P_S is the space component of the mapping P, which can be obtained by solving not more than $2^n - 1$ systems of n equations.

In connection with this, the function DT is determined according to the following expression:

$$DT = R_T P_T D = R_T D_T, \quad (2)$$

where DT is the PE exchange synchronization function, $DT = [r_1^T]$ determines the quantity of the delay elements for the 1-th variable, R_T is row-matrix of the weight coefficients (this matrix serves to transform the r-dimensional time vector P_T to the scalar time); P_T is the time component of the mapping P; $D_T = [d_1^T]$ is submatrix with elements d_1^T , determined by multidimensional schedule, $d_1^T = [d_{i1}^T]$, $i=1, r$.

The multidimensional schedule is determined by the time component $P_T = [P_{Ti}]$, where P_{Ti} is the i-th constituent of the time component P_T in the mapping P. In this connection the constituent P_{Ti} determines the direction of some hyperplane P_{Ti}^* ($\bar{y} \bar{d} \bar{n} P_{Ti}^* \Rightarrow P_{Ti} \bar{d} = 0$) transfer. The process of the schedule generation for the graph G can be represented in the following way: if $r=1$, then sequence of FO computation is determined by sequential transfer of hyperplane P_{T1}^* ; if $r=2$, then we introduce the second hyperplane P_{T2}^* , whose transfer determines sequence of the execution of FOs lying in one of its position; if $r=3$, then we introduce P_{T3}^* , which determines the sequence of the execution of FOs, belonging to $P_{T1}^* \cap P_{T2}^* \cap G$. We can proceed the same way for $r>3$.

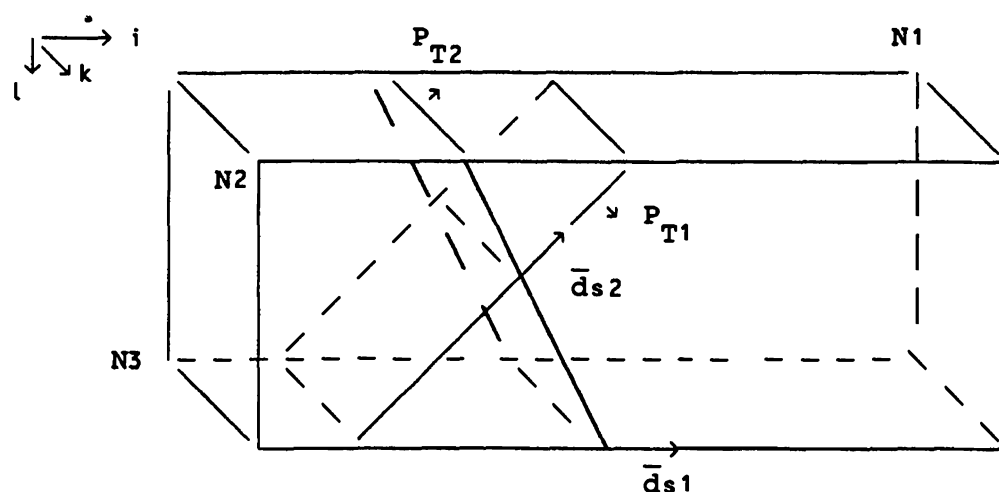


Fig.1.

The schedule of the graph G by two constituents P_{T1} and P_{T2} of the time component of P is shown in figure 1.

Let us consider necessary and sufficient conditions for the determination of all of the constituents P_{Ti} of the time component P_T which do not violate the preceding relation between the FOs from the original algorithm, which optimizes the desired capacity of processor array memory and the algorithm execution time. So, P_{T1} can be determined from the system of inequality $P_{T1} \bar{d}_1 \wedge 0$.

The $(k+1)$ -th constituent synthesis is done on the basis of the following type of inequalities:

$$P_{T(k+1)} \bar{d}_1 \wedge 0 \quad (3)$$

for all of the $\bar{d}_1$ vectors satisfying conditions

$$P_{T1} \bar{d}_1 = 0, \quad P_{T2} \bar{d}_1 = 0, \quad \dots, \quad P_{Tk} \bar{d}_1 = 0. \quad (4)$$

The r -th constituent synthesis for processor array is done by solving such inequalities as $P_{Tr} \bar{d}_1 \wedge 1$. In this connection, to save the injectiveness of the mapping P it is necessary to observe the relation

$$\det P \neq 0. \quad (5)$$

The determination of the elements of the matrix R_T is done on the basis of V_{Ti} sets of vertices I , satisfying the condition

$$V_{Ti} = (I \mid I \bar{\wedge} P_{T1}^* \wedge I \bar{\wedge} P_{T2}^* \wedge \dots \wedge I \bar{\wedge} P_{Ti}^* \wedge I \bar{\wedge} G). \quad (6)$$

Then, elements of the matrix R_T are determined in the following way:

$$R_T = [\hat{r}_i], \quad \hat{r} = \frac{|V_{Ti}|}{M_S}, \quad (7)$$

where $|V|$ is the cardinality of the set V , M_S is the PEs number in the processor structure. $|V_{Ti}|$ and $\hat{r}_i$ values are called weight and reduced weight of the i -th constituent of time component P_T , respectively. They can be variables.

To make further facilitation of the elements d_1^T of the function DT determination, it can be showed that the constituents P_{Ti} of the time component P_T should be chosen by the condition

$$P_{Ti} \bar{d}_{mk} = 0, \quad k = \overline{1, i-1}, \quad (8)$$

where $\bar{d}_{mk}$ is a vector of computational fronts displacements, stipulated by hyperplane P_{Tk}^* transfer, and also $\bar{d}_{mk} \bar{\wedge} \text{Ker } P_S$,

where $\text{Ker } P_S$ is a kernel of the space component P_S .

The examples of multidimensional schedule synthesis see in the works ^{5,7}.

3. Index set mapping

Before considering the placement, input-output and using prefixed variables, it is necessary to classify them ⁶. All of the l -th variables, for which the condition $P_S^j \bar{d}_1 = 0$ ($j = \overline{1, m}$, where j is the number of matrix P_S row) is fulfilled, will be considered as the first type variables and variables for which the condition $P_S^k \bar{d} \neq 0$ ($k = \overline{1, m}$) hold, as the second type variables. All other variables satisfying the conditions $P_S^j \bar{d} = 0$ and $P_S^k \bar{d} \neq 0$ when $j \neq k$ and $j, k = \overline{1, m}$ will be considered as third-type.

Let us briefly review a method of mapping of indexes set of the algorithm variables. For this it is necessary to consider a subsidiary task. A part of some two-dimensional ($n=2$) graph G is shown in figure 2. Let us consider that it is necessary to obtain a value, a sign and an exchange direction of index i of the variable X between vertices I and J , $J - I = \bar{d}\Lambda$.

This task may be solved due to the auxiliary vector $\bar{h}_1^i$ for increment of the l -th variable index i . Analytical notation of this vector has the following form

$$(\bar{d}_1)^t \bar{h}_1^i = 0, \quad (9)$$

$$\frac{a_i}{\|\bar{h}_1^i\|} = \max \quad \text{with} \quad \min \|\bar{h}_1^i\| \geq 0, \quad (10)$$

where $(\bar{d})^t$ is the transpose symbol of the vector $\bar{d}$, $\|h\|$ is the norm of vector $\bar{h}$, a_i is the number of increment steps for index i between FOs J and I , $J - I = \bar{h}_1^i$. Let us assign $D_i = 1$.

Then the sought value, sign and exchange direction of index i may be obtained from the expression

$$q^i = (\bar{d}\Lambda)^t \bar{h}_x^i. \quad (11)$$

Consider a space distribution of the first type variable a in structure ($n=2, m=1$). On the base of Eq.11 the space distribution

of the first type variable a is obtained as

$$Q_{s,a}^i = \frac{a_i}{\|h_a\|^i} * \frac{1}{\|P_S\|} * P_S \bar{h}_a^i, \quad (12)$$

where $\|h\|^*$ is an integer-valued norm of $\bar{h}$ ($\|h\|^* = \sum_{k=1}^n |h_k|$, $\bar{h} = [h_k]$)⁸.

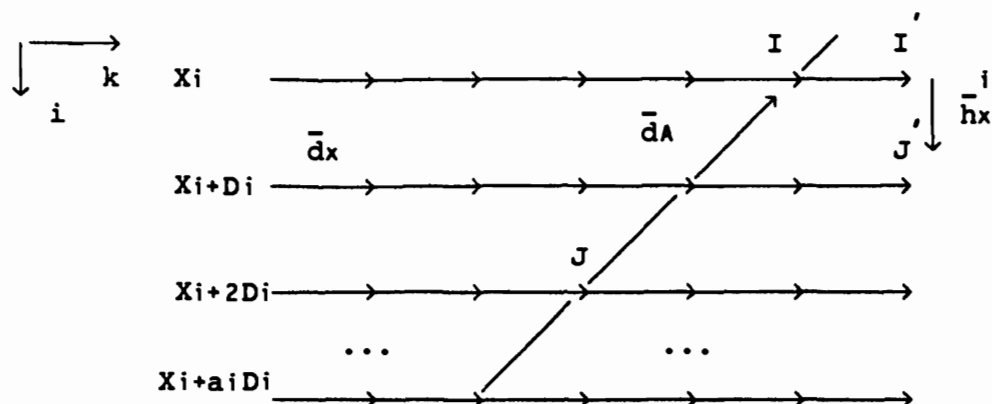


Fig. 2.

For most algorithms $\frac{a_i}{\|h_a\|^i} = D_i = 1$.

Two different cases of mapping the first type variable a is shown in figure 3.

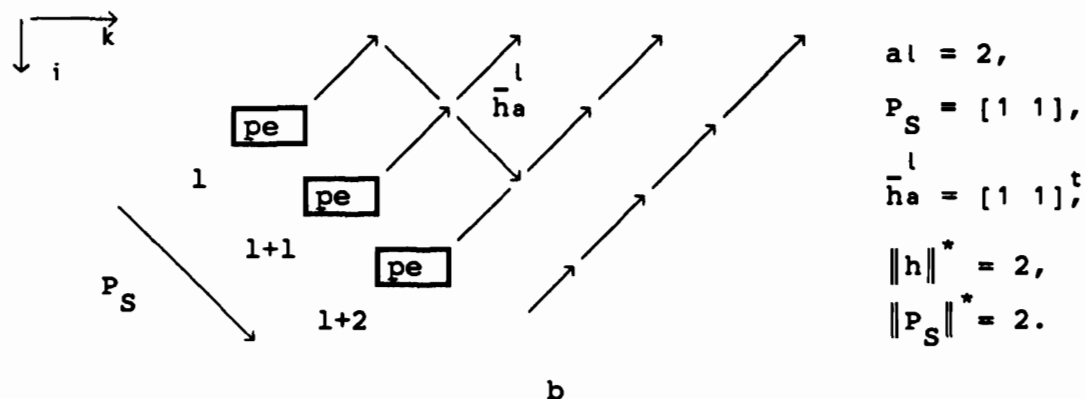
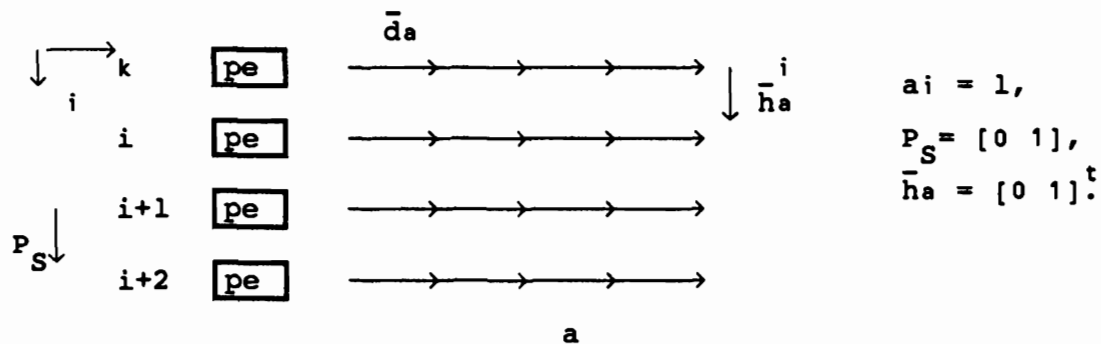


Fig. 3.

In the first case (fig. 3, a) the vector $\bar{h}_a^i$ and any of

coordinate vectors are parallel. Then $(\|P_S\|^*)^{-1} = 1$. In the second case (fig.3,b) the pointed vectors are not parallel.

When x is a multidimensional variable, the Ex.12 will be true for each dimension. The last may be presented in the following form

$$Q_{s,a} = D_* P_S H_a, \quad (13)$$

where D is a diagonal matrix,

$$D_* = \text{diag}[(\|p_s^1\|^*)^{-1}, (\|p_s^2\|^*)^{-1}, \dots, (\|p_s^m\|^*)^{-1}],$$

p_s^k is the k -th row of the matrix P_S ; H_a is a matrix consisting of increment indexes vectors $\bar{h}_a$, $H_a = [\bar{h}_a]$.

After obtaining the space distribution of the first type variables it is necessary to determine input (output) order and delay time between neighbouring elements for each input (output) variable. The time delay between input values of the neighbouring elements X_i and X_{i+a_i} of the variable X is defined such expression

$$d_{x,br}^T = |P_T \bar{d}_{x,br}|, \quad (14)$$

where $\bar{d}_{x,br}$ is the border vector-column of the variable X . An example of this vector is shown in figure 4. The index exchange direction of index i of variable X is defined from the inequality

$$P_T \bar{d}_{x,br} = \begin{cases} > 0 \text{ for decrement of index } i, \\ < 0 \text{ for increment of index } i, \\ = 0 \text{ for parallel input } X \text{ for all } i. \end{cases} \quad (15)$$

The example of a schedule and an entering into computing process the first type variable X are shown in figure 4. These conditions will be true for all input and output variables ($r=1$).

Let us consider the second type variables. It is necessary to obtain the time delay between loading (unloading) neighbouring values and index exchange direction for this type of variables.

The time delay between performing two neighbouring successive FOs in single PE is determined by the following expression ($r=1$)

$$d_{ds}^T = P_T \bar{d}_s, \quad (16)$$

where $\bar{d}_s$ is vector-column satisfying the conditions $P_S \bar{d}_s = 0$ ($\bar{d}_s \notin \text{Ker} P_S$), $P_T \bar{d}_s > 0$. Following this, a number of steps and a sign of exchange of the index i of the first type variable x ($r=1$) may be obtained from the next equation:

$$q_{T,x}^i = (\bar{d}_s) \bar{h}_x^i = \begin{cases} > 0 \text{ for decrement of index } i, \\ < 0 \text{ for increment of index } i, \\ = 0 \text{ for parallel input } X \text{ for all } i. \end{cases} \quad (17)$$

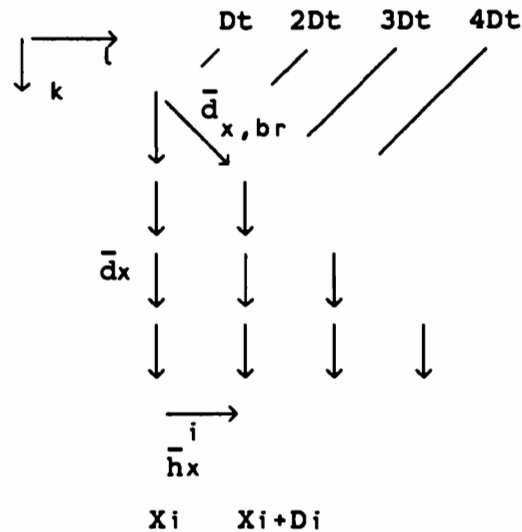


Fig.4.

The proposed method may be easily extended for cases when $r > 1$. The example of mapping of indexes set into structure see in the works ^{6,7,9}.

4. Conclusion

A method for synthesis of processor arrays is developed and a system of rules and formulas is suggested which permits to obtain a set of alternative structural circuits (see also ^{10,11}) of processors of the preset dimension for one step of designing.

Determination of space distribution and sequence of input-output of variables is done on the basis of the proposed method for the mapping of algorithm variables indexes set onto processor array structure. The synthesis of some auxiliary vector for input and output variables of the first type input-output is done on its basis also.

5. References

- 1 H. T. Kung, Why systolic architectures?, Computer, January 1982, pp. 37-46.
- 2 S. Y. Kung et al., VLSI and Modern Signal Processing, Prentice-Hall, Englewood Cliffs, NJ, 1985.
- 3 D. I. Moldovan, On the design of algorithms for VLSI systolic arrays, Proc. IEEE, Vol. 71, No. 1, January 1983, pp. 113-120.
- 4 P. Quinton, The systematic design of systolic arrays, Technical Report, INRIA Report, Paris, 1983.
- 5 Ju. S. Kanevsky and D. V. Korchev, Method for synthesis of multidimensional time-base of computations in systolic processors, Electronnoe modelirovanie, 1990, 4, pp. 43-49. (in Russian).
- 6 Ju. S. Kanevsky and D. V. Korchev, Method for imaging of a set of indexes of algorithm variables on systolic structures, Kibernetika, 1991, 1, pp. 51-57. (in Russian).
- 7 Ju. S. Kanevsky, Systolic arrays, Technika, Kiev, 1991. (in Russian).
- 8 R. A. Horn and C. R. Johnson, Matrix analysis, Cambridge University Press, 1986.
- 9 Ju. S. Kanevsky, and D.V. Korchev, A systematic design of DFT systolic processors', Radioelectronics, 1990, 9, pp. 61-66. (in Russian, translated into English "Radioelectronics and Communications Systems").
- 10 Ju. S. Kanevsky and D. V. Korchev, Systematic design of systolic FFT-processors, Latvian Signal Processing Int. Conf., LSPIC-90, - Riga, April 24-26, 1990, Vol. 2, pp. 193-197.
- 11 Ju. S. Kanevsky and D.V. Korchev, Two-dimensional systolic array for fast Fourier transform, First Int. Conf. On Information Technologies For Image Analysis And Pattern Recognition, ITIAPR'90, - Lviv, October 22 - 28, 1990, Vol. 2, pp. 172-176.

DESIGN OF SYSTOLIC ARRAYS FOR ITERATIVE ALGORITHMS: EIGENVALUE COMPUTATIONS

N.A.LIKHODED, P.I.SOBOLEVSKII and A.A.TIOUNTCHIK

Laboratory of Mathematical Problems of
Design Automation, Institute of Mathematics,
Byelorussian Acad. Sci., Surganov str. 11,
Minsk, 220604, USSR

ABSTRACT

A new method of designing systolic arrays for iterative algorithms is presented. Such arrays implement any number of iterative algorithm steps in a coordinated way. The quantity of their processor elements is independent of this number. Efficiency of this method is illustrated by the examples of systolic arrays (planar, linear and fixed size) for the solving eigenvalue problem by the triangular power method.

1. Introduction

Many problems of numerical analysis may be solved by iterative methods. Iterative steps of some algorithms include matrix operations. Such operations are efficiently implemented on systolic arrays which are high-performance special-purpose multiprocessors.

At present many systolic arrays for macro operations of iterative algorithms (matrix multiplication, QR and LU decomposition¹⁻⁴, etc.) are known. Unfortunately, an ordinary combination of the predesigned systolic arrays for macro operations does not allow to use all advantages of systolic architecture for iterative algorithms. The fact is that the results of a macro operation can not be immediately used for the next macro operation. It is necessary to add the global interconnections and interprocessor buffers to coordinate the successive implementations of macro operations.

In this paper a method of designing systolic arrays for

iterative algorithms is briefly presented. Such arrays implement any number of iterative algorithm steps in a coordinated way. The quantity of their processor elements (PEs) is independent of this number. Efficiency of this method is illustrated by the examples of systolic arrays (planar, linear and fixed size) for the solving eigenvalue problem by the triangular power method. Other examples of using this method are presented in⁵⁻⁷, where systolic arrays for the multiplication of a large number of matrices, for finding the absolutely largest eigenvalue of a matrix by the power method, and for the solution of linear systems of equations by the Seidel method and the relaxation method have been designed.

2. Main Stages of Design

1. Designing base systolic arrays (BSAs) for macro operations of the m -th iterative step. BSA is such a systolic array that its graph $G(V,E)$ coincides with the graph of the algorithm implemented by this systolic array: PEs of BSA correspond to the vertices $v \in V$ of graph $G(V,E)$, interconnections between PEs of BSA correspond to the arcs $e \in E$ of graph $G(V,E)$, directions of data propagation are defined by the system of directing vectors $\hat{E}(\varphi_1, \varphi_2, \dots, \varphi_r)$. The temporary conditions of calculation and data exchange are defined by the timing function $t(v)$, $v \in V$. This function defines the moment when the PE placed in point $v \in V$ starts running. Timing function for strictly oriented graphs can be defined by the equality

$$t(v) = \langle n, v \rangle + t_0 - \min_{v \in V} \langle n, v \rangle, \quad v \in V, \quad n \in K(\hat{E}),$$

where $K(\hat{E})$ is the cone of permissible directions (such an integer-valued vector set, that each vector forms acute angles with directing vectors of the set $\hat{E}$), $\langle n, v \rangle$ is a scalar product of the vector n and the position-vector of the vertex $v \in V$, t_0 is a constant characterizing the moment when BSA starts running. Values $\langle n, \varphi_\alpha \rangle = h_\alpha \geq 1$ define delays along interconnections with directing vectors $\varphi_\alpha \in \hat{E}$.

2. Designing BSA_m , $m=1,2,\dots$, which implements the m -th iterative step of the algorithm. The design of such BSA is carried out by the sticking method⁵⁻⁸. Sticking of BSA_m and BSA_{m+1} is denoted as $BSA_m \cup BSA_{m+1}$.

3. Designing infinite $BSA_m \cup BSA_m$ which implements any number of the steps of the iterative algorithm. This BSA must be limited by a cylinder with the diameter independent of the number of iterations. Designing such BSA is carried out by the sticking method which is applied to the base systolic arrays BSA_m .

4. Mapping $BSA_m \cup BSA_m$ by a linear operator. The operator must preserve interconnection locality and nullify the direction vector of the cylinder. Such mapping allows to obtain systolic arrays with the number of their PEs independent of the number of iterative steps. Mapping operator adds some restrictions for the timing function since all PEs being mapped onto one must have different running time. Note that some PEs of different types may be mapped onto one. Correct running of systolic arrays is provided by the system of tags (control signals). Tags are propagated by the special net which is created in BSA . Moreover, each datum is to be propagated from the boundary PEs. This is provided with the sticking of additional PEs which move data to the input area of BSA .

5. Choosing optimal timing for the systolic array. The solution of the running time minimization problem is carried out in terms of restrictions added at the stage of sticking and mapping.

6. Designing systolic arrays of fixed size. This procedure is based on the partitioning of the systolic array $BSA_m \cup BSA_m$ by hyperplanes parallel to the cylinder generator.

3. Example: Designing Systolic Array for the Eigenvalue Problem

The eigenvalue problem is often solved by the iterative algorithms (QR algorithm, triangular power method, etc.). Systolic arrays for the symmetric eigenvalue problem are

known to exist⁹⁻¹¹. For the general eigenvalue problem, however, only systolic arrays for separate iterative steps (QR decomposition, matrix multiplication, etc.) are known.

Let matrix $A=(a_{ij})$, $1 \leq i, j \leq N$, have N different eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_N$. Assume further that the eigenvalues are ordered by magnitude, $|\lambda_1| > |\lambda_2| > \dots > |\lambda_N|$. Calculation of the eigenvalues by the triangular power method is as follows:

$$\begin{aligned} 1) & AC_0 = B_1, \quad B_1 = C_1 R_1, \\ 2) & AC_1 = B_2, \quad B_2 = C_2 R_2, \\ & \dots \dots \dots \\ m) & AC_{m-1} = B_m, \quad B_m = C_m R_m, \\ & \dots \dots \dots \end{aligned}$$

where $C_m = (c_{ij}^m)$, $m=0, 1, 2, \dots$, $1 \leq i, j \leq N$, $c_{ij}^m = 0$, $i < j$, $c_{ii}^m = 1$, $R_m = (r_{ij}^m)$, $m=1, 2, \dots$, $1 \leq i, j \leq N$, $r_{ij}^m = 0$, $i > j$, (at the m -th step, we multiply two matrices and then calculate the LU decomposition). Since $\lim_{m \rightarrow \infty} r_{ii}^m = \lambda_i$, $1 \leq i \leq N$, then for the sufficiently large number m we may assume $\lambda_i \approx r_{ii}^m$.

Two BSA ($BSA_{m,1}$ and $BSA_{m,2}$) implementing macro operations for the m -th iterative step were designed: matrix multiplication:

$$\begin{aligned} & 1 \leq i \leq N, \quad 1 \leq j \leq N, \quad j \leq k \leq N: \\ a(i, j, k) &= \begin{cases} a_{ik}, & j=1, \\ a(i, j-1, k), & j \neq 1, \end{cases} \\ c^{m-1}(i, j, k) &= \begin{cases} c_{kj}^{m-1}, & i=1, \quad k \neq j, \\ c^{m-1}(i-1, j, k), & i \neq 1, \quad k \neq j, \end{cases} \\ b^m(i, j, k) &= \begin{cases} a(i, j-1, k), & k=j, \\ b^m(i, j, k-1) + a(i, j, k) \cdot c^{m-1}(i, j, k), & k \neq j, \end{cases} \\ b_{ij}^m &= b^m(i, j, N); \end{aligned} \quad (1)$$

and triangular decomposition:

$$\begin{aligned}
 & k=1, 2, \dots, N: \\
 r^m(i, j, k) = & \begin{cases} 1/b_{ij}^m, & i=j=k=1, \\ -b_{ij}^m, & i=k=1, \quad 2 \leq j \leq N, \\ 1/b^m(i, j, k-1), & k \neq 1, \quad i=j=k, \\ -b^m(i, j, k-1), & k \neq 1, \quad i=k, \quad k+1 \leq j \leq N, \\ r^m(i-1, j, k), & k+1 \leq i \leq N, \quad k \leq j \leq N, \end{cases} \\
 c^m(i, j, k) = & \begin{cases} b_{ij}^m \cdot r^m(i-1, j, k), & 2 \leq i \leq N, \quad j=k=1, \\ b^m(i, j, k-1) \cdot r^m(i-1, j, k), & k \neq 1, \quad k+1 \leq i \leq N, \quad j=k, \\ c^m(i, j-1, k), & k+1 \leq i, j \leq N, \\ c^m(i, j, k-1), & k \leq i \leq N, \quad 1 \leq j \leq k-1, \end{cases} \\
 \tilde{c}^m(i, j, k) = & \begin{cases} \tilde{c}^m(i-1, j, k), & k+1 \leq i \leq N, \quad 1 \leq j \leq k-1, \\ c^m(i, j, k-1), & i=k, \quad 1 \leq j \leq k-1, \end{cases} \\
 b^m(i, j, k) = & \begin{cases} b_{ij}^m + r^m(i-1, j, k) \cdot c^m(i, j-1, k), & k=1, \quad 2 \leq i, j \leq N, \\ b^m(i, j, k-1) + r^m(i-1, j, k) \cdot c^m(i, j-1, k), & k \neq 1, \quad k+1 \leq i, j \leq N; \end{cases} \\
 r_{ij}^m = & \begin{cases} b_{ij}^m, & i=1, \quad 1 \leq j \leq N, \\ b^m(i, j, i-1), & 2 \leq i \leq j \leq N, \end{cases} \\
 c_{ij}^m = c^m(i, j, i) = \tilde{c}^m(N, j, i), & 2 \leq i \leq N, \quad 1 \leq j \leq i-1. \quad (2)
 \end{aligned}$$

Graphs $G^1(V^1, E^1)$ and $G^2(V^2, E^2)$ of algorithms (1) and (2) are defined by sets

$$\begin{aligned}
 V^1 = & \{v(i, j, k) \in \mathbb{Z}^3 \mid 1 \leq i, j \leq N, \quad j \leq k \leq N\}, \quad \hat{E}^1\{e_1, e_2, e_3\}, \\
 V^2 = & \{v(i, j, k) \in \mathbb{Z}^3 \mid 1 \leq j, k \leq N, \quad k \leq i \leq N\}, \quad \hat{E}^2\{e_1, e_2, e_3\},
 \end{aligned}$$

where $\mathbb{Z}^3$ is the index space, $e_1 = (1, 0, 0)$, $e_2 = (0, 1, 0)$,

$e_3 = (0, 0, 1)$. Timing function is defined by the equality

$$t_o^{m, \alpha}(v) = \langle n_o^{m, \alpha}, v \rangle + t_o^{m, \alpha} - \min_{v \in V^\alpha} \langle n_o^{m, \alpha}, v \rangle, \quad v \in V^\alpha, \quad n_o^{m, \alpha} \in K_m(\hat{E}^\alpha),$$
 $\alpha = 1, 2$.

$BSA_{m,1}$ contains PEs of two types, and $BSA_{m,2}$ contains PEs of six types.

Each BSA_m , $m = 1, 2, \dots$, for the m -th iterative step of algorithm was designed by sticking $BSA_{m,1}$ and $BSA_{m,2}$ for macro operations. Sticking includes spatial match and temporal coordination. Spatial match of $BSA_{m,1}$ and $BSA_{m,2}$ is ensured by the translation of $BSA_{m,2}$ by the vector $N \cdot e_3$. Temporal coordination is achieved by making the following conditions to hold: $t_o^{m,2} = t_o^{m,1} + N h_3^{m,1}$, $h_1^{m,2} = h_1^{m,1}$, $h_2^{m,2} = h_2^{m,1}$, where $h_\gamma^{m, \alpha} = \langle n_\gamma^{m, \alpha}, e_\gamma \rangle \geq 1$.

Base systolic array for the triangular power method (Fig.1) was designed by sticking all $BSA_m = BSA_{m,1} \cup BSA_{m,2}$, $m = 1, 2, \dots$, implementing the steps of the iterations. Spatial match of BSA_m is ensured by the translation of BSA_m by the vector $N(m-1)(e_1 + e_3)$. The designed $\cup BSA_m$ is limited by the cylinder with direction vector $s = (1, 0, 1)$. The diameter of the cylinder is independent of the number of iterations. Temporal coordination of BSA_m is achieved by making of the following conditions to hold:

$$t_o^{m+1,1} = t_o^{m,2} + N h_1^{m,2}, \quad h_2^{m+1,1} = h_2^{m,2}, \quad h_3^{m+1,1} = h_3^{m,2}.$$

Timing function of the structure $\cup BSA_m$ (in terms of all restrictions) is

$$t(v) = t^{1,1}(v) +$$

$$+ \sum_{q=1}^{[(k-1)/N]} (h_3^{q,2} - h_3^{q,1})(k - qN - 1) + \sum_{q=1}^{[(i-1)/N]} (h_1^{q+1,1} - h_1^{q,2})(i - qN - 1),$$

$$v(i, j, k) \in V = \bigcup_{m \geq 1} (V^{m,1} \cup V^{m,2}), \quad V^{m,1} = \{v(i, j, k) \in \mathbb{Z}^3 \mid (m-1)N + 1 \leq i \leq$$

$\leq mN, 1 \leq j \leq N, j+(m-1)N \leq k \leq mN\}$, $V^{m,2} = \{v(i,j,k) \in \mathbb{Z}^3 \mid k+(m-1)N \leq i \leq mN, 1 \leq j \leq N, mN+1 \leq k \leq (m+1)N\}$, where $[\cdot]$ is floor function.

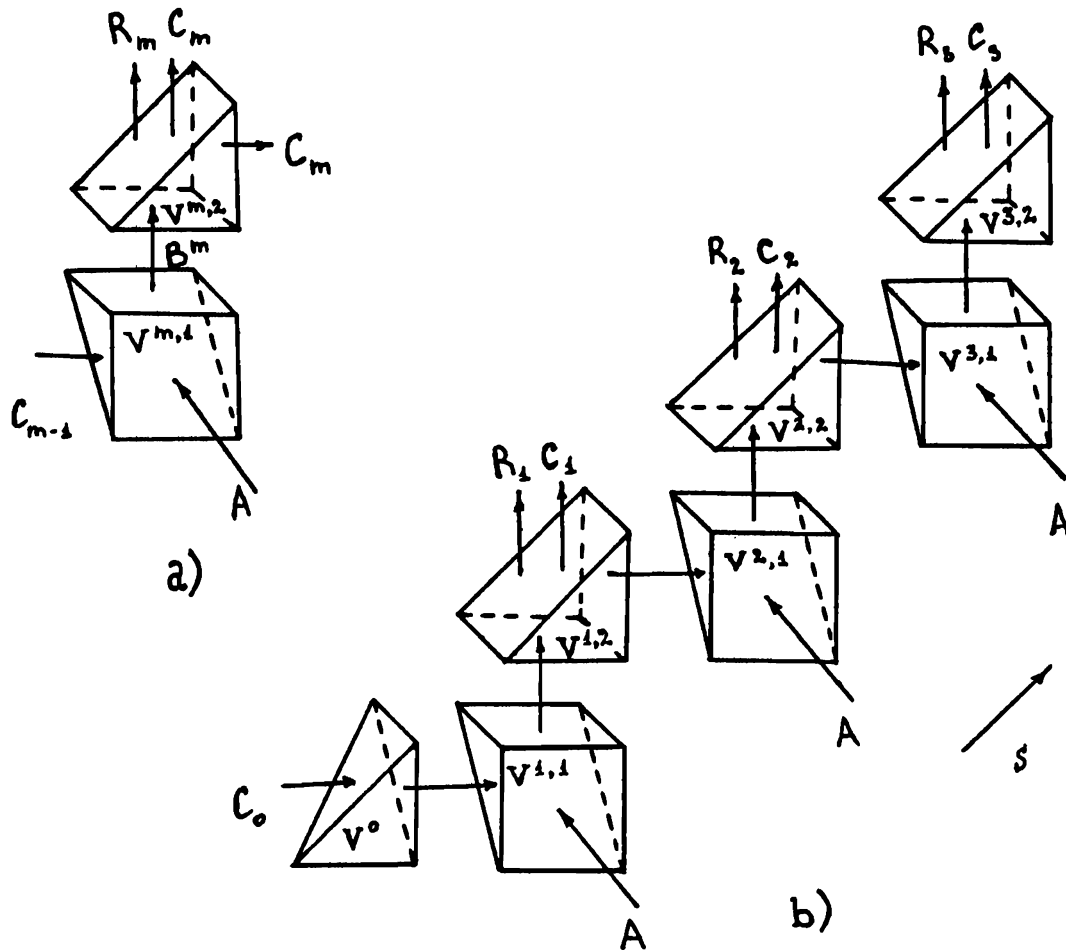


Fig.1. Base systolic array $BSA_m = BSA_{m,1} \cup BSA_{m,2}$ (a);
base systolic array $\cup BSA_m$ (b).

A linear operator $\Pi_{3,2}: \mathbb{Z}^3 \rightarrow \mathbb{Z}^2$ with matrix $\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$ maps the array $\cup BSA_m$ onto the planar systolic array presented in Fig.2. The linear systolic array (Fig.3) has been created by the mapping via linear operator $\Pi_{3,1}: \mathbb{Z}^3 \rightarrow \mathbb{Z}^1$ with matrix $[1 \ 0 \ -1]$. The planar systolic array implements m

iterative steps of triangular power method in time $2N(m+1)+N-2$ and the linear one in time $N^2(m+1)$.

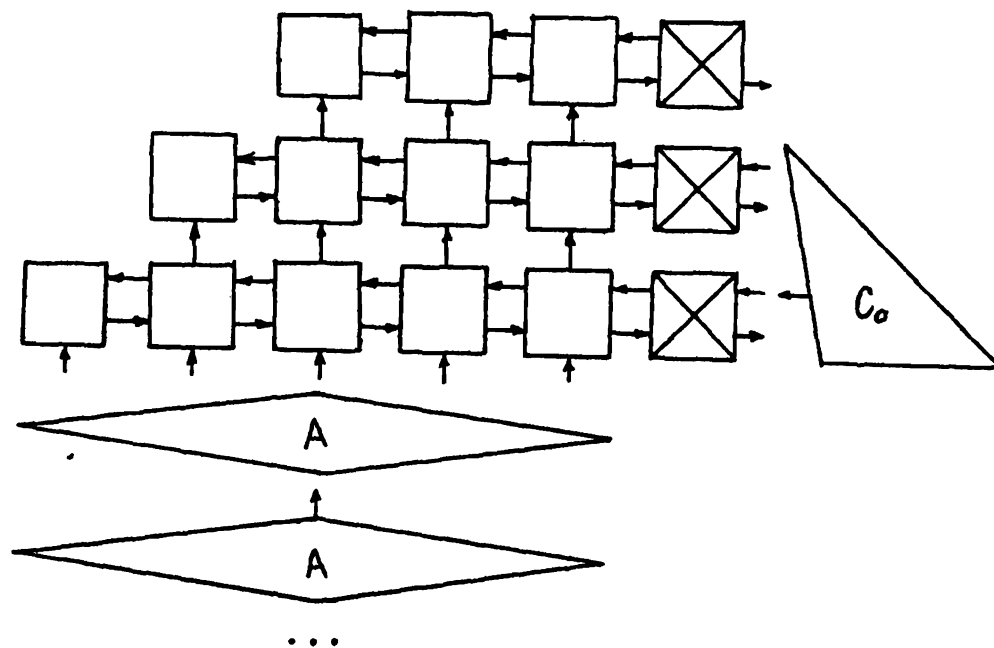


Fig.2. Planar systolic array.

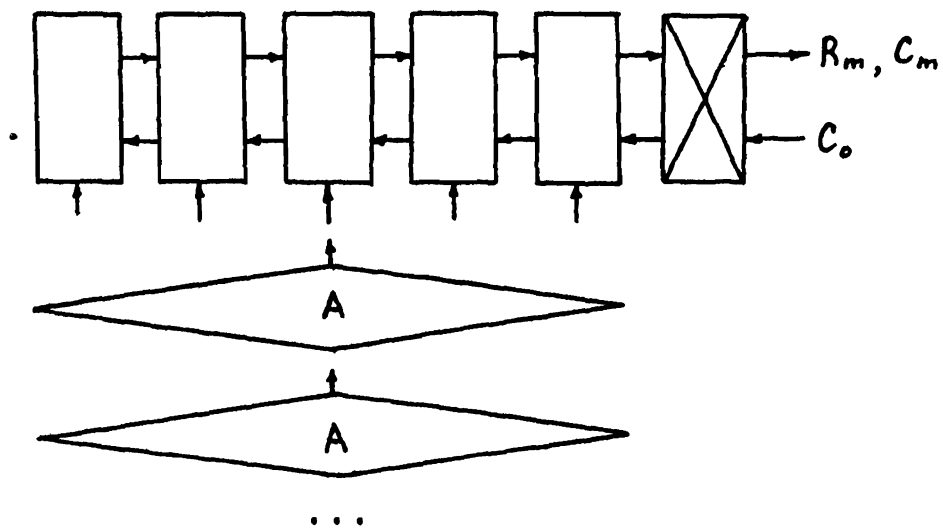


Fig.3. Linear systolic array.

The size of VLSI arrays is constrained by technological factors. Therefore the designed planar systolic array of $(3N^2+N)/2$ PEs and the linear systolic array of $2N$ PEs are sometimes useless for a matrix with a large number of entries. Therefore, a linear fixed size systolic array (Fig.4) is of great importance. This systolic array

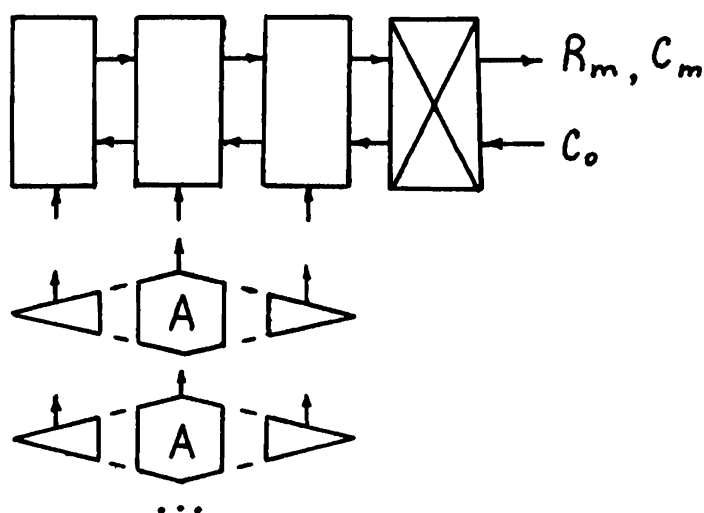


Fig.4. Linear fixed size systolic array.

consists of $\Gamma+1$ ($\Gamma \leq 2N-1$) PEs where Γ may be preassigned and the number of PEs is independent of both the number of iterations and the size of the matrix. The fixed size linear systolic array implements m iterative steps of the triangular power method in time $\lceil (2N-1)/\Gamma \rceil (N^2m+N^2-1)+1$, where $\lceil \cdot \rceil$ is ceiling function.

REFERENCES

1. St.G.Sedukhin, preprint №589 (Novosibirsk, 1985) (in Russian).
2. D.Moldovan, *Proc. of the IEEE* 71 (1983) p.113-120.
3. S.-Y.Kung, *Proc. of the IEEE* 72 (1984) p.867-884.
4. A.Bojanczyk, R.P.Brent and H.T.Kung, *SIAM J. Sci. Stat.*

Comput. 5 (1984) p.95-104.

5. P.I.Sobolevskii and N.A.Likhoded, *preprint №15(415)* (Minsk, 1990) (in Russian).
6. N.A.Likhoded, *Vesti AN BSSR, Ser. phys.-math. nauk*, №3 (1991) p.109-114 (in Russian).
7. N.A.Likhoded, P.I.Sobolevskii and A.A.Tiountchik, *Vesti AN BSSR, Ser. phys.-math. nauk* (to appear) (in Russian).
8. N.A.Likhoded and P.I.Sobolevskii, *Dokl. AN BSSR* 33 (1989) p.389-392 (in Russian).
9. R.P.Brent and F.T.Luk, *SIAM J. Sci. and Statist. Comput.* 6 (1985) p.69-84.
10. D.S.Scott, M.T.Heath and R.Ward, *Linear Algebra and Appl.* 77 (1986) p.345-355.
11. H.J.H.Chuang and L.Chen, *Proc. Int. Conf. Parallel Process., Aug. 17-21, 1987.* (University Park, Pa, 1987) p.550-556.
12. D.K.Faddeev and V.N.Faddeeva, *Vychislitelnyy metody lineinoi algebry* (Fizmatgiz, Moscow-Leningrad, 1963) (in Russian).

**THE DESIGN OF A LINEAR SYSTOLIC ARRAY
FOR THE GAUSS-JORDAN ALGORITHM WITH PARTIAL PIVOTING**

V.V.KOSYANCHUK and V.P.YAKUSH

*Laboratory of Mathematical Problems of
Design Automation, Institute of Mathematics
Byelorussian Acad. Sci., Surganov str. 11,
Minsk, 220604, USSR*

ABSTRACT

One-dimensional systolic array for the solution of dense linear systems based on modified Gauss-Jordan algorithm with partial pivoting is designed. Pivoting operations are necessary to ensure the satisfactory numerical behavior of algorithm. But it disturbs the regular pattern of operations. In order to obtain a regular processor array a modification of the algorithm is implemented and a special spatial mapping of algorithm graph is chosen.

1. Introduction

In a pioneering work of Kung and Leiserson¹ a class of regular locally connected structures known as systolic arrays for high speed parallel implementation of algorithms of numerical linear algebra and graph theory is proposed. Since then many authors have devised systolic arrays for various problems, one of them being the problem of solving systems of linear equations. Many direct and iterative methods have been proposed for this widespread problem. Gauss-Jordan elimination for the direct solution of dense linear systems is one of them. It is well suited to the parallel implementation and has an advantage over similar

methods (LU-decomposition and Gauss elimination) since phases of factorization and back substitution may be executed parallelly. A number of two-dimensional and one-dimensional systolic arrays have been proposed for this algorithm²⁻⁶. But the Gauss-Jordan algorithm as well as LU-decomposition and the Gauss elimination have an essential shortcoming. It is numerically unstable if no pivoting is exploited. There are several works⁷⁻¹⁰ where systolic arrays for the Gauss elimination with partial pivoting are presented. Small number of papers of this direction may be explained that pivoting disturbs the regularity of algorithm structure. In this paper a linear systolic array for the Gauss-Jordan elimination method with partial pivoting is designed. In order to obtain a regular processor array, a modification of the algorithm is implemented and a special type of spatial mapping is chosen.

The design of the systolic array proceeds in the following order:

- obtaining the algorithm graph $G(V, E)$, where V is a set of its vertices, and E is a set of its arcs;

- designing the basic systolic array (BSA), where BSA is a systolic array obtained by placing at each vertex $v \in V$ the processing element (PE) executing corresponding operation. The order of the execution of PE operations is defined by the timing function

$$t(v) = t_0 + (n, v), \quad (1)$$

where $t_0 = \text{const}$, vector $n = (n_1, n_2, n_3) \in \mathbb{Z}^3$ satisfies condition

$$(n, e) \geq 1, \quad e \in E, \quad (2)$$

(n, v) is a dot product of n and v . BSA is a computational model of the algorithm which is defined by the graph G , timing function t and PE functions

- tagging BSA, what means that we shall supply all PEs of the same type by the same control signal (tag)

- choosing spatial mapping $\Pi: \mathbb{Z}^3 \rightarrow \mathbb{Z}^1$ and vector n under condition: $t(v_1) \neq t(v_2)$ for all $v_1, v_2 \in V$ such that

$\Pi(v_1) = \Pi(v_2)$. As proved in works¹¹⁻¹², this condition is satisfied if and only if $(n, s) \neq 0$ for all $s \in S$, where $S = ((V - V) \cap (\text{Ker } \Pi)) - \{0\}$, $V - V = \{v \in \mathbb{Z}^3: v = v_1 - v_2, v_1, v_2 \in V\}$.

$\text{Ker}\Pi = \{v \in \mathbb{Z}^3 : \Pi(v) = 0\}$. These papers also propose a procedure of obtaining the optimal timing function(1) if mapping Π is chosen.

-mapping BSA onto systolic array via Π

2. Modified Gauss-Jordan Elimination Algorithm with Partial Pivoting. Basic Systolic Array

Consider the problem of solving systems of linear equations

$$A_1 \cdot X = A_2, \quad (3)$$

where $A_1 = (a_{ij})$, $1 \leq i, j \leq N$, $A_2 = (a_{ij})$, $1 \leq i \leq N < j \leq N+M$, $X = (x_{ij})$, $1 \leq i \leq N$, $1 \leq j \leq M$. To solve it the following version of the Gauss-Jordan elimination algorithm with partial pivoting is used (we suppose that a_{11} is the pivot for A_1):

$$a(i, j, 0) = a_{ij}, \quad 2 \leq i \leq N, \quad 1 \leq j \leq N+M,$$

$$w(N, j, 0) = a_{1j}, \quad 1 \leq j \leq N+M;$$

for $k=1$ to N do:

1) $b(k, k, k) = w(k+N-1, k, k-1)$

for $i=k+1$ to $k+N-1$ do:

2) $b(i, k, k) = a(i, k, k-1)$

for $j=k+1$ to $N+M$ do:

3) $b(k, j, k) = b(k, j-1, k)$,

$$c(k, j, k) = w(N+k-1, j, k-1) / b(k, j-1, k)$$

for $j=k+1$ to $N+M$ do:

4) $b(k+1, j, k) = b(k+1, j-1, k)$,

$$c(k+1, j, k) = c(k, j, k),$$

$$u(k+1, j, k) = a(k+1, j, k-1) - b(k+1, j-1, k) \cdot c(k, j, k)$$

for $i=k+2$ to N do:

5) $b(i, k+1, k) = b(i, k, k)$,

$$c(i, k+1, k) = c(i-1, k+1, k),$$

$$p = a(i, k+1, k-1) - b(i, k, k) \cdot c(i-1, k+1, k),$$

if $|p| > |u(i-1, k+1, k)|$

then $\begin{cases} u(i, k+1, k) = p, \\ a(i, k+1, k) = u(i-1, k+1, k), \\ \varphi(i, k+1, k) = 1, \end{cases}$

else $\begin{cases} u(i, k+1, k) = u(i-1, k+1, k), \\ a(i, k+1, k) = p, \\ \varphi(i, k+1, k) = 0 \end{cases}$

```

for i=k+2 to N do:
  for j=k+2 to N+M do:
    6) b(i,j,k)=b(i,j-1,k),
       c(i,j,k)=c(i-1,j,k),
        $\varphi(i,j,k)=\varphi(i,j-1,k)$ ,
       if  $\varphi(i,j,k)=1$ 

       then { a(i,j,k)=u(i-1,j,k),
              u(i,j,k)=a(i,j,k-1)-b(i,j-1,k)·c(i-1,j,k),
            else { a(i,j,k)=a(i,j,k-1)-b(i,j-1,k)·c(i-1,j,k),
                  u(i,j,k)=u(i-1,j,k)
        for i=N+1 to N+k-1 do:
          for j=k+1 to N+M do:
            7) b(i,j,k)=b(i,j-1,k),
               c(i,j,k)=c(i-1,j,k),
               u(i,j,k)=u(i-1,j,k),
               a(i,j,k)=a(i,j,k-1)-b(i,j-1,k)·c(i-1,j,k)
        for j=k+1 to N+M do:
          8) a(N+k,j,k)=c(N+k-1,j,k),
             w(N+k,j,k)=u(N+k-1,j,k);

 $x_{ij}=a(N+i,N+j,N), 1 \leq i \leq N, 1 \leq j \leq M.$ 

```

Here search for the pivot are executed in a whole column

For each operation 1)-8) we put into the correspondence the vertex $v(i,j,k) \in \mathbb{Z}^3$. For each pair of informatively connected vertices we put into the correspondence the arc. Obtain the graph $G(V,E)$ of this algorithm. The corresponding BSA is depicted in Fig.1. BSA has eight types of PEs. PEs of the first type are located in the vertices (k,k,k) , $1 \leq k \leq N$, the second type in (i,k,k) , $1 \leq k \leq N$, $k+1 \leq i \leq k+N$, the third type in (k,j,k) , $1 \leq k \leq N$, $k+1 \leq j \leq N+M$, the fourth type in $(k+1,j,k)$, $1 \leq k \leq N$, $k+1 \leq j \leq N+M$, the fifth type in $(i,k+1,k)$, $1 \leq k \leq N-2$, the sixth type in (i,j,k) , $1 \leq k \leq N-2$, $k+2 \leq i \leq N$, $k+2 \leq j \leq N+M$, the seventh type in (i,j,k) , $2 \leq k \leq N$, $N+1 \leq i \leq N+k-1$, $k+1 \leq j \leq N+M$, the eighth type in $(N+k,j,k)$, $1 \leq k \leq N$, $k+1 \leq j \leq N+M$.

3. Tagging of BSA

If the obtained BSA is mapped onto one-dimensional systolic array then some BSA processing elements of

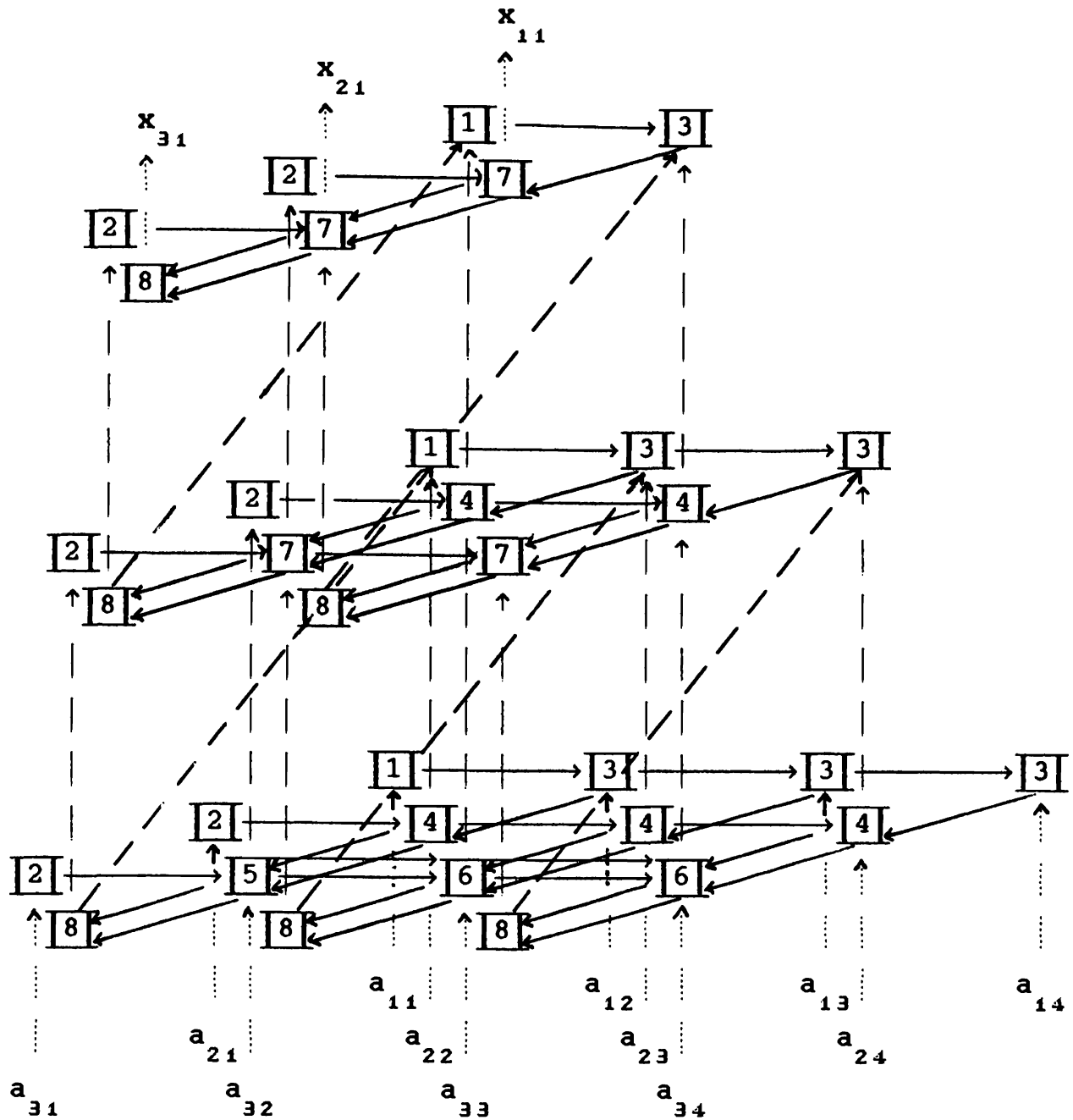


Fig.1. BSA structure.

different types are mapped onto one and the same PE. Hence this PE must operate in several modes. To identify these modes BSA is tagged. With this purpose 3-bit tag $\tau = (\tau^1, \tau^2, \tau^3)$ is entered into each $PE(i, j, 1)$ and then it is transferred to other PEs in the direction defined by the

vector $(1,1,1)$. Tag $\tau_1 = (1,0,1)$ is entered in $PE(1,1,1)$, $\tau_2 = (1,1,1)$ in $PEs(i,1,1), 2 \leq i \leq N$, $\tau_3 = (1,1,0)$ in $PEs(1,j,1), 2 \leq j \leq N+M$, $\tau_4 = (0,1,0)$ in $PEs(2,j,1), 2 \leq j \leq N+M$, $\tau_5 = (0,0,1)$ in $PEs(i,2,1), 3 \leq i \leq N$, $\tau_6 = (0,0,0)$ in $PEs(i,j,1), 3 \leq i \leq N, 3 \leq j \leq N+M$, $\tau_8 = (1,0,0)$ in $PEs(N+1,j,1), 2 \leq j \leq N+M$. But this tagging doesn't distinguish PEs of the sixth and the seventh types. Therefore an additional 1-bit tag $\varepsilon=0$ is entered into each $PE(i,j,1)$ and all PEs of the eighth type are provided with an additional function, which generates the similar tag $\varepsilon=1$. All these tags are transferred in direction $(0,0,1)$ and are assumed unchanged during the propagation. Now the pair of tags $(\varepsilon=1, \tau=(0,*,*))$ defines the PEs of the seventh type, and the pair $(\varepsilon=0, \tau=(0, \tau^2, \tau^3))$ defines the PEs of the fourth, fifth or sixth type depending on the second and the third bits of τ (the sign * point to insignificance of corresponding bit value).

4. The Choice of the Mapping Π and the Optimal Timing Function

The obtained BSA has global interconnections defined by the vector $(1-N,0,1)$. For the designed systolic array to have only local interconnections it is necessary to choose mapping Π such that global interconnections are mapped onto local. One of the mapping, satisfying this demand, is $\Pi=(0,0,1)$. An advantage of this mapping over other ones is that the corresponding systolic array is optimal in the number of processing elements and I/O ports.

The vector n defining the timing function (1), will be chosen to minimize the functional $T(n) = n_1(2N-1) + n_2(N+M-1) + n_3N$, which determines the runtime of a task. Under the scheme proposed in papers¹¹⁻¹² such vector can be found and corresponding value of T equals to $2N^2 + M(N+1) + 2N - 2$.

5. Systolic Array Design

The one-dimensional systolic array obtained by mapping BSA with the operator Π has N identical PEs (Fig.2a).

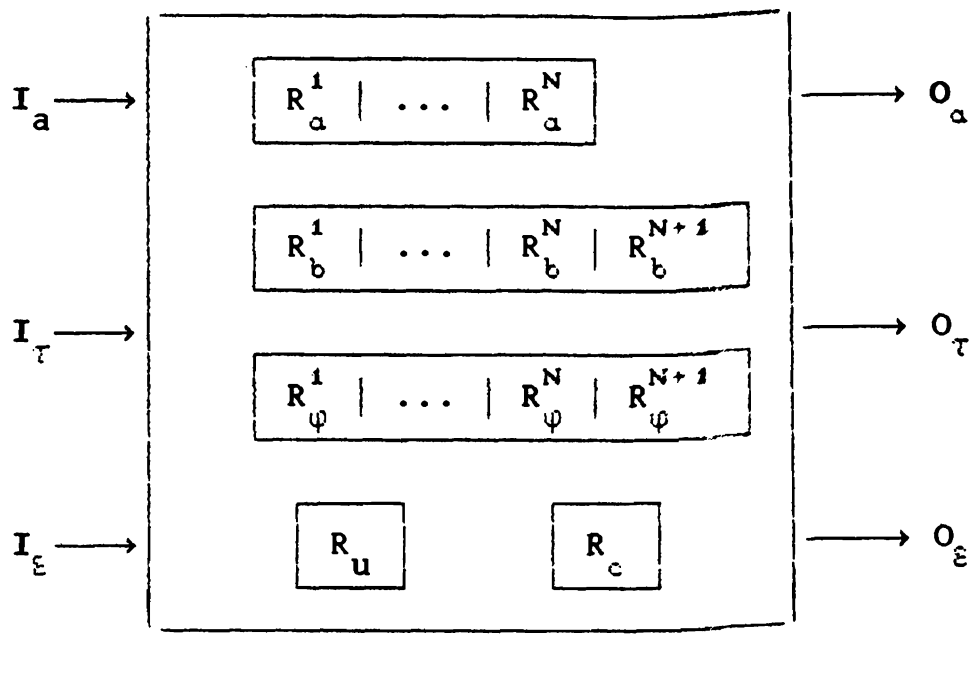
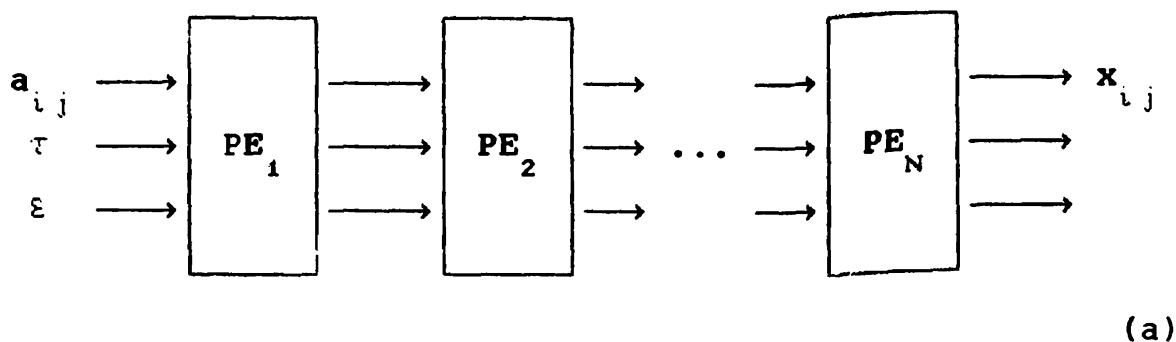


Fig.2. Linear systolic array (a).
The scheme of PE (b).

The processing element is depicted in Fig.2b, where its input/output ports and some shift registers of storage are shown. Each PE operates in eight modes, defined by a pair of arriving tags τ and ε . We shall write " R_x " and a_{in} respectively for the contents of register R_x and for the data arriving at the input I_a of PE at the considered moment. The functions of PE in different modes are the following: The 1st ($\varepsilon=0, \tau=(1,0,1)$) and the 2nd ($\varepsilon=*, \tau=(1,1,1)$) modes: input data a_{in} is stored in the shift register R_b of lengths $N+1$ (cyclic shift of this register

is done in other modes). The 3d mode ($\varepsilon=0, \tau=(1,1,0)$): a_{in} is divided at $"R_b^1"$ (the contents of the first register of R_b), and quotient is stored in register R_c . The 4th mode ($\varepsilon=0, \tau=(0,1,0)$): value $(a_{in} - "R_b^1" \cdot "R_c")$ is computed and the result is stored in register R_u . The 5th mode ($\varepsilon=0, \tau=(0,0,1)$): value $p = (a_{in} - "R_b^1" \cdot "R_c")$ is evaluated, after that magnitudes of p and $"R_u"$ are matched. If $|p| > |"R_u"|$ then contents of register R_u is fed to the output O_a through the shift register R_a of lengths N , p is stored in register R_u and value 1 is stored in the shift register R_φ of lengths $N+1$ containing status bits of change rows. Else value p is fed to the output O_a and value 0 is stored in the R_φ . The 6th mode ($\varepsilon=0, \tau=(0,0,0)$): if $"R_\varphi^1"=1$ then $"R_u"$ is fed to the output O_a after N time units and value $(a_{in} - "R_b^1" \cdot "R_c")$ is stored in R_u , else (if $R_\varphi^1=0$) this difference is fed to the output O_a and the contents of R_u is not changed. The 7th mode ($\varepsilon=1, \tau=(1,*,*)$): value $(a_{in} - "R_b^1" \cdot "R_c")$ is evaluated and then it is fed to the output O_a . The 8th mode ($\varepsilon=*, \tau=(1,0,0)$): contents of registers R_u and R_a^1 (after N time units) are fed to the output O_a .

6. Conclusion

The linear systolic array for the Gauss-Jordan method of solving of a linear system equations is proposed. It contains N identical processing elements and takes $2N^2 + M(N+1) + 2N - 2$ time units for the solution of the task. The above assumption that a_{11} is a pivot for the matrix A_1 is insignificant. Actually, we may extend the initial algorithm with the procedure of partial pivoting for A_1 . This causes the extension of algorithm graph (and BSA). An

additional layer of vertices (PEs) appears and the resulting systolic array will contain an additional PE performing pivoting for A_1 .

7. References

1. H.T.Kung and C.E.Leiserson, in *Sparse Matrix Proc. Soc. for Indust. and Appl. Math.* (1979) pp.256-282.
2. R.Melhem, *Parall. Comput.* 4,3 (1987) pp.339-343.
3. St.G.Sedukhin, preprint 744 (Novosibirsk, 1987) p.44 (In Russian)
4. V.V.Kosyanchuk and N.A.Likhoded, *Vesti AN BSSR, ser. phiz.-math. nauk*, 3 (1990) pp.95-100. (In Russian)
5. I.-C.Jou, S.-C.Tsau and S.Y.Chiang, in *Proc. Comput. Comm. Tech. toward 2000*. 3 (New York, 1987) p.954-957.
6. P.Comon and J.Robert, *IEEE Trans. Acoust. Speech Signal Process.* 35,6 (1987) p.717-723.
7. M.Gentlemen and H.T.Kung, in *Proc. SPIE* 298 (1981) p.19-26.
8. V.P.Roychowdhury and T.Kailath, *Proc. Int. Conf. Systolic Arrays* (1988) pp.237-246.
9. A.El-Amavy and H.Barada, *IEEE Proc* 137,4 (E-1990) p.295-300.
10. R.Wyrzykowski and Yu.S.Kanevskii, *Proc. Conf. "Odnor. Vychisl. Sredy i Syst Struct."* 1 (Lvov,1990) pp.41-47 (In Russian)
11. V.V.Kosyanchuk, N.A.Likhoded and P.I.Sobolevskii, preprint 11(321) (Minsk, 1988) p.34. (In Russian)
12. V.V.Kosyanchuk, *Vesti AN BSSR, ser. phiz.-math. nauk* 2 (1990) pp.88-97. (In Russian)

Parallel Constraint Satisfaction in a Logic Programming Framework

Manfred Meyer*

German Research Center for Artificial Intelligence (DFKI) GmbH
P. O. Box 20 80, D-6750 Kaiserslautern
Germany

Abstract

Constraint satisfaction networks have been shown to be a very useful tool for knowledge representation and problem-solving in different areas such as Operations Research, Hardware Design and Artificial Intelligence applications. Consistency techniques on finite domains are one of the constraint-solving techniques supported by the constraint logic programming language FIDO. They allow the solving of many discrete combinatorial problems within a short development time and offer an efficiency comparable to procedural programs. Nevertheless most of these problems still remain computationally expensive and the emergence of parallel computers opens up a new research area: the combination of constraint satisfaction and parallel execution in the logic programming framework. This paper reports a first step in this direction. It shows that *or*-parallelism and consistency techniques on finite domains can be combined to yield an efficient parallel language for representing and solving discrete combinatorial problems. Some problems which would arise with an *or*-parallel version of FIDO will be discussed in detail and a solution to them will be described as well. The paper then completes with a comparison of this approach to the SRI Model for *or*-parallel Prolog.

1 Introduction

Many problems in different areas such as Operations Research, Hardware Design, and Artificial Intelligence can be seen as constrained search problems [10]. Most of these real-life problems are known to be NP-complete. That means there is no hope to design an efficient (polynomial) algorithm for solving them.

*e-mail: meyer@dfki.uni-kl.de, fax: +49-631-205-3210

Logic programming offers a convenient and declarative way for stating these problems due to its relational form and nondeterminism. The relational form makes it adequate to state constraints while the nondeterminism liberates the programmers from explicitly coding the tree search. Unfortunately, standard logic programming languages such as Prolog are also very inefficient for solving them because they are based on a generate and test paradigm.

Therefore during the last years a lot of research has been done to improve the efficiency of logic programming for combinatorial problems. This has led to the definition of several constraint programming languages (e.g., [5, 8, 13, 26, 20]). These languages combine the declarative aspects of logic programming with the efficiency of constraint-solving techniques and open new application areas for logic programming. Among these languages is FIDO [20], a constraint logic programming language developed at DFKI. FIDO extends a usual logic programming language by providing finite domains together with efficient consistency techniques on them such as forward checking. Forward checking prunes the search space in an 'a priori' way instead of the 'a posteriori' way of current logic languages. This extension leads to a substantial improvement of efficiency for this class of combinatorial problems and contributes to make constraint logic programming an appropriate tool for expressing and solving constraint satisfaction problems.

Concurrently to the above research, much attention has been devoted to parallelizing Prolog. The motivation behind this research stems from the emergence of commercial multiprocessor computers and the declarative aspects of logic programming which, in principle, liberates programmers from managing parallelism. Different computation models have been proposed to exploit either *or*-parallelism (e.g., [4, 16, 21, 28]) either *and*-parallelism (e.g., [7, 12]) or a combination of both (e.g., [29]). Several of these models have been successfully implemented and show promising results [2, 16].

The results in these two different directions, together with the fact that most constrained search problems are computationally expensive, were a major impetus to investigate parallel implementations of constraint logic programming languages. In this context, finite domains and consistency techniques for them such as forward checking are of particular interest. They have a large variety of applications in discrete combinatorial problems and offer large potentiality for parallelization. Currently solving a discrete combinatorial problem in parallel requires either the writing of a specialized program in a procedural language (e.g., [15, 27]) or the use of a special-purpose package [9]. The first approach is programming-intensive; solving a discrete combinatorial problem in sequential requires much programming effort and parallelism further increases the programming complexity. Special-purpose packages relieve programmers from managing parallelism but they are, at the moment, rather inefficient. Compared to the above approaches, a parallel version of FIDO is very appealing. Not only it reduces the programming effort to the sequential part itself simplified by the use of FIDO, but it should also result in a good efficiency provided that the overhead of par-

allelism is kept small. It is therefore of much interest to study whether the sequential efficiency of FIDO can be preserved in a parallel implementation.

This paper reports a first step in that direction: the design of an *or*-parallel version of FIDO, which is very similar to a restricted version of that presented in [25].

It shows that the principles behind the sequential implementation can be generalized to support *or*-parallelism while keeping the overhead small. In particular, basic operations on domains remain fast, constant-time operations.

The rest of this paper is organized as follows. Section 2 gives an overview on finite domains and consistency techniques. Section 3 identifies different opportunities for parallelism in FIDO. Finally, section 4 shows how to exploit *or*-parallelism together with finite domains and consistency techniques.

2 Finite Domains and Consistency Techniques

In this section, we introduce some foundations and characteristics of FIDO, the **F**inite **D**omain extension of SEPIA, which in turn is an advanced Prolog implementation developed at ECRC [18]. Besides a large variety of UNIX platforms it runs also on the KCM [1], a very fast dedicated Prolog system¹ used at DFKI.

The main characteristics of SEPIA include an incremental compiler, module system, a fast incremental garbage collector, an external interface to predicates written in C, and – most important for the implementation of FIDO – a declarative coroutining facility.

Using this coroutining facility, programs written in FIDO can be compiled into ordinary SEPIA programs. The forward checking mechanism of FIDO can be implemented by making use of SEPIA's `delay`-statements [19, 6].

2.1 Domain-variables

The basic feature of FIDO for solving discrete combinatorial problems is its ability to work on domain-variables, i.e., variables ranging over a finite domain. Domains can be either a set of natural numbers or a set of constants.

2.2 Constraints over Finite Domains

FIDO provides a large variety of constraints on domain-variables. It contains not only **arithmetic** but also **symbolic** and even **user-defined** constraints. Arithmetic constraints define the usual relations on arithmetic terms over domain-variables (e.g., $\leq$, $\geq$, $=$, and $\neq$). Symbolic constraints define symbolic relations between several

¹The KCM (Knowledge Crunching Machine) is a single user, single task high-performance back-end processor which, coupled to a UNIX desk-top workstation, provides a fast and user-friendly Prolog environment catering for both development and execution of significant Prolog applications. It runs a speed of 833 KLips (Kilo Logical inferences per second) on list concatenation.

variables. They often allow more natural problem statements and more efficient problem-solving. Besides these primitive constraints, programmers can also define constraints of their own as logic programs and tell the system how to handle them using forward and lookahead declarations.

2.3 Consistency Techniques

All constraints involving domain-variables are solved by using consistency techniques, an AI paradigm for solving discrete combinatorial problems (e.g., [11, 17]). The principle behind these techniques is to use constraints to reduce the domains of variables and hence the search space. Different kinds of pruning (i.e., reduction of the domains) have been identified and efficient ways to achieve them have been devised. A theoretical framework has been proposed to embed these techniques inside logic programming and serve as basis for the implementation of the built-in constraints and the general control mechanism [23, 24]. Consistency techniques are usually not able to solve the constraints by their own. It follows that solving a discrete combinatorial problem using them consists in iterating the following two steps

- propagating the constraints as far as possible
- making a choice

until a solution is reached.

3 Selecting the Parallelism

In this section, we review the main opportunities for parallelism arising in FIDO. As is well-known, there are mainly two kinds of parallelism inside logic programming: *and*- and *or*-parallelism. However, since the computational model of FIDO differs from the one of ordinary Prolog, it is appropriate to reconsider that issue. As mentioned previously, the computational model of FIDO for solving discrete combinatorial problems is best viewed as the iteration of two steps: propagation and choices. Both steps provide opportunities for parallelism, three of them being considered now.

First, a constraint can be *solved* in parallel. This opportunity for parallelism arises inside the forward and lookahead declarations. Indeed, these control mechanisms solve a constraint by trying different value combinations for the variables appearing inside the constraint and these combinations could possibly be tried in parallel. This kind of parallelism is implicit in the bitwise forward checking version described in [11] and is expected to be of small granularity in most cases.

Second, the constraints can be *propagated* in parallel. During the propagation step, several constraints have to be considered and we might think of propagating

them in parallel. This leads to a kind of *and*-parallelism. Once again this parallelism is expected to be of small granularity in most cases² and to introduce some synchronization problems since several constraints might share the same variable. It has attracted several researchers in the AI community (e.g., [22]). There also exist some theoretical limitations for its exploitation [14].

Third, *choices* can be made in parallel. This is the usual *or*-parallelism of Prolog except that FIDO has much more to offer in that context than usual logic languages. Indeed, choices can take various forms from instantiation (i.e., giving a value to a variable), to domain-splitting (i.e., dividing the domain of a variable in several parts), and to case analysis (i.e., using constraints for making choices). In all cases, a choice means choosing among different alternatives and the basic idea here is to explore several of them simultaneously. This parallelism has attracted many researchers outside the logic programming community (e.g., [9, 15, 27]) and can be of large granularity.

Among the above sources of parallelism, *or*-parallelism seems to be the most promising. It applies to the computation as a whole, not to a specific part of it (e.g., the constraint propagation). It can also be of large granularity and thus more amenable to implementation on a multiprocessor computer. Finally, it amounts to adding a breadth-first component which can be valuable for the class of problems considered. Therefore we decided to restrict ourselves in the design of a first prototype to the combination of *or*-parallelism and constraint-solving. The constraint imposed on the design of the prototype is rather standard now in the area of parallel logic programming: the efficiency of the implementation when run on a single processor has to be comparable to a good sequential implementation.

4 Towards an *or*-parallel implementation of FIDO

In order to support *or*-parallelism, it is necessary to adapt the current implementation schemes of FIDO. This section reviews the problems raised by *or*-parallel FIDO and proposes several solutions to them.

4.1 *The Problems*

The computation of a logic language can be seen as a tree whose root node is the initial goal and other nodes are obtained by reducing a goal using SLD-resolution or an extension of it. Different branches of the tree come from the different clauses used to reduce a goal. Prolog explores the computation tree using a depth-first search with chronological backtracking. In other words, when several clauses can be applied to reduce a goal, one of them is selected. The remaining clauses will be tried on backtracking. In *or*-parallel Prolog, different processors are used to explore several branches of the tree in parallel, i.e., they try simultaneously several clauses for

²This is not the case for *and*-parallelism in general.

reducing the same goal. The main problem raised by *or*-parallel Prolog is how to represent different bindings of the same variable on different branches of the computation tree. Different schemes have been proposed to deal with that problem (e.g., [3, 28, 29]). In addition to the binding problem, FIDO introduces a further difficulty due to domain-variables. Let us illustrate the problem through an example. Suppose X is ranging over $\{1, \dots, 10\}$ and the constraint $X \neq 3$ has to be solved. In theory, this constraint is solved by binding X to a new variable Y ranging over $\{1, 2, 4, \dots, 10\}$. However, for efficiency and memory reasons, the implementation scheme of FIDO removes the value 3 from the domain of X and trails the modification to undo it on backtracking. Therefore the new problem to be solved by *or*-parallel FIDO is how to represent different states of the same domain on different branches of the tree³. Although this problem looks similar to the binding problem of Prolog, it has a main difference. In Prolog, a variable is free or bound. When the variable is bound, it never changes its value. This is not the case for domain-variables. The domain fields (i.e., the fields representing the domain of a variable) can be updated several times. It follows that existing binding schemes do not apply directly although they provide useful insights on the present issue.

4.2 *A Solution with Copying*

A straightforward solution to the above problem consists of requiring that *or*-parallel FIDO follows precisely the theory, i.e., each time the domain of a variable is updated, a new variable ranging over the reduced domain is created and the variable is bound to this fresh variable. This solution enables existing binding schemes of *or*-parallel Prolog to be used for solving our problem. Its basic advantage is its simplicity. Its drawback is the unacceptable overhead it induces. In sequential FIDO, basic operations on domain-variables require constant time and do not depend on the domain size. This property is obviously violated by the copy solution. Since we aim at an efficiency on a single processor comparable to a good sequential implementation, a better scheme has to be devised.

4.3 *Avoiding Copying the Domains*

In order to avoid copying the domains, a new data structure local to a process, the domain frame, is introduced. It contains the state of domains for the branch the process is working on. When a domain-variable is created, its domain is stored in the domain frame instead of in the global stack for the sequential implementation. It follows that each process has its own view of the domains and that modifications of the domain fields only occur within the domain frame without affecting the view of

³A similar problem arises with the constraints. Since the solution is the same in both cases, we only discuss domains in this paper.

other processes. The use of a domain frame makes sure that all basic operations on domains (e.g., testing if a value is in the domain) remain constant-time operations.

The main overhead of this approach arises when switching tasks. In that case, the process has to update the contents of the domain frame to reflect the state of the domains at the branching point it is grabbing work from. The update of the domain frames is achieved by using the usual value-trail which has to be generalized for that purpose to include three fields:

- the entry in the domain frame;
- the old value of the entry;
- the new value of the entry.

The *old value* field is necessary to restore the previous value of the entry on backtracking while the *new value* field is needed to update domain frames at switching tasks.

4.4 *Avoiding Trailing*

Since the value-trail is used not only on backtracking but also to update the domain frames, it is necessary to make sure that it contains all information necessary to perform these two activities. An obvious way to enforce this requirement consists of trailing all modifications to the domain frames. However, this simple solution would induce a severe overhead and requires much more memory for the parallel version than for the sequential one. Moreover the size of the value-trail directly defines the cost of switching tasks and thus reducing the amount of trailing decreases the cost of switching tasks. For these reasons, special care should be given to avoid trailing. The sequential optimizations have to be generalized to make sure that the amount of trailing of *or*-parallel FIDO is proportional to the one of sequential FIDO.

4.5 *Relation to the SRI Model*

The above scheme has some similarities with the SRI model [28]. Each process (resp. worker in the SRI model) has a local data-structure for storing the domains (resp. bindings) which make sure that the basic operations on domains (resp. variables) remain constant-time operations. The basic overhead arises at task switching due to the need to update the domain frame (resp. the binding array). However there are also some important differences between them.

The SRI model distinguishes between conditional and unconditional bindings. Only conditional bindings are recorded into the binding array and trailed. Conditional bindings are directly stored into the variable cell. This distinction comes from the single assignment property of the logical variable. Once bound, a variable never changes its value. This is not true for domain fields which can be updated several

times and differently on different branches. It follows that all the domain fields have to be stored in the domain frame.

The scheme does not assume a global address space as does the SRI model. During dereferencing, a worker in the SRI model can access other workers' stacks. Since access to a non-local stack is transparent in that model, it is necessary to have a global address space. Conversely, the domain frame does not contain references but only pure data (i.e., the value of the domain fields) so that no global address space is necessary.

Finally, it is worth noting that the scheme presented here is independent of the building scheme used for variables.

5 Conclusion

In this paper, we have considered the issue of exploiting parallelism and constraint-solving together in the logic programming framework. Starting with the computational model of FIDO for discrete combinatorial problems, different opportunities for parallelism have been identified. The most promising alternative, the combination of *or*-parallelism and consistency techniques on finite domains, has been considered in some detail. The problems raised by an *or*-parallel FIDO have been defined and several possible solutions have been proposed. Among them was an efficient scheme which makes sure that all operations on domains remain constant-time operations and preserves most of the efficiency of the sequential implementation.

Acknowledgements

This paper would never have been written without the work done in the constraint group at DFKI. In particular, we are indebted to Klaus Elsbernd, Hans-Günther Hein, and Bernd Bachmann for many valuable discussions about parallelism and constraint-solving techniques. We also greatly benefitted from the work of Stefan Schrödl who implemented the first sequential FIDO system. Thanks are also due to Jörg Müller for many discussions on the compilation of FIDO to SEPIA and for proofreading earlier versions of this paper.

This work was done within the ARC-TEC project at DFKI and was partly supported by the German Federal Ministry of Research and Technology (BMFT) under grant ITW 8902 C4.

References

1. H. Benker, J.M. Beacco, S. Bescos, M. Dorochevsky, T. Jeffré, A. Pöhlmann, J. Noyé, B. Poterie, A. Sexton, J.C. Syre, O. Thibault, and G. Watzlawik. KCM: A Knowl-

- edge Crunching Machine. In *The 16th Annual International Symposium on Computer Architecture - ISCA '89*, pages 186–194, June 1989.
2. J. Chassin, J.C. Syre, and H. Westphal. Implementation of a Parallel Prolog System on a Commercial Multiprocessor. In *ECAI'88*, pages 278–283, Munich, Germany, August 1988.
 3. A. Ciepielewski, B. Hausman, and S. Haridi. Or-parallel Prolog Made Efficient on Shared Memory Multiprocessors. In *Symposium on Logic Programming*, August 1987.
 4. W. Clocksin and H. Alshawi. A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors. *New Generation Computing*, 5:361–376, 1988.
 5. A. Colmerauer. Opening the Prolog III Universe. *BYTE*, pages 177–182, August 1987.
 6. D. de Schreye, D. Pollet, J. Ronsyn, and M. Bruynooghe. Implementing Finite-domain Constraint Logic Programming on Top of a PROLOG-System with Delay-mechanism. In N. Jones, editor, *Proc. of ESOP 90*, pages 106–117, 1990.
 7. D. DeGroot. Restricted And-Parallelism. In *Proc. FGCS'84*, pages 471–478, Tokyo, Japan, 1984.
 8. M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proc. FGCS'88*, Tokyo, Japan, December 1988.
 9. R. Finkel and U. Manber. DIB – A Distributed Implementation of Backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, April 1987.
 10. M. Fox. Observations on the Role of Constraints in Problem Solving. In *Proc. 6th Canadian Conference on Artificial Intelligence, Montreal, 1986*, pages 172–187. Presses de l'Université de Québec, May 1986.
 11. R. M. Haralick and G. L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.
 12. M. Hermenegildo. An Abstract Machine for Restricted AND-parallel execution of logic programs. In *Third International Conference on Logic Programming*, pages 25–39, London, July 1986.
 13. J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. POPL-87*, Munich, Germany, 1987.
 14. S. Kasif. On the Parallel Complexity of Some Constraint Satisfaction Problems. In *AAAI-86*, pages 349–353, Philadelphia, Pa., 1986.
 15. V. Kumar, K. Ramesh, and V. Nageshewara Rao. Parallel Best-First Search of State-Space Graphs: A Summary of Results. In *AAAI-88*, pages 122–127, Saint Paul, Minnesota, 1988.

16. E. Lusk et al. The Aurora Or-Parallel Prolog System. In *Proc. FGCS'88*, Tokyo, Japan, December 1988.
17. A. Mackworth. Consistency in Networks of Relations. *AI Journal*, 8(1):99–118, 1977.
18. M. Meier, A. Aggoun, D. Chan, P. Dufresne, R. Enders, D.H. de Villeneuve, A. Herold, P. Kay, B. Perez, E. van Rossum, and J. Schimpf. SEPIA – An Extendible Prolog System. In G. Ritter, editor, *Proceedings of the IFIP 11th World Computer Congress*, pages 1127–1132, August 1989.
19. J. Müller. Entwurf und Implementierung eines Constraint-Logic-Programming-Systems auf der Basis von PROLOG mit Coroutining. Diplomarbeit, Universität Kaiserslautern, FB Informatik, Postfach 3049, D-6750 Kaiserslautern, 1991. Forthcoming.
20. S. Schrödl. FIDO: Ein Constraint-Logic-Programming-System mit Finite Domains. DFKI Document, DFKI GmbH, Postfach 2080, D-6750 Kaiserslautern, 1991. Forthcoming.
21. E. Shapiro. An OR-Parallel Execution Algorithm for Prolog and its FCP Implementation. In *Fourth International Conference on Logic Programming*, pages 311–337, Melbourne, Australia, May 1987.
22. M. Swain and P. Cooper. Parallel Hardware for Constraint Satisfaction. In *AAAI-88*, pages 682–686, Saint Paul, Minnesota, 1988.
23. P. van Hentenryck. A Framework for Consistency Techniques in Logic Programming. In *IJCAI-87*, Milan, Italy, August 1987.
24. P. van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, Ma., 1989.
25. P. van Hentenryck. Parallel Constraint Satisfaction in Logic Programming (Preliminary results of CHIP within PEPsSys). In *Proc. of ICLP 89*, pages 165–180, 1989.
26. P. Voda. The Constraint Language Trilogy: Semantics and Computations. Technical Report, Complete Logic Systems, North Vancouver, British Columbia, Canada, 1988.
27. B. Wah and Y. Ma. MANIP: A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems. *IEEE Transactions on Computers*, 33(5):377–390, May 1984.
28. D.H.D. Warren. The SRI Model for Or-Parallel Execution of Prolog. Abstract Design and Implementation Issues. In *International Symposium on Logic Programming*, pages 46–53, September 1987.
29. H. Westphal, P. Robert, J. Chassin, and J.C. Syre. The PEPsSys Model: Combining Backtracking, AND- and OR-parallelism. In *International Symposium on Logic Programming*, pages 436–448, September 1987.

INTEGRATED TOOLS FOR FUNCTIONAL, LOGICAL AND DATA FLOW PARALLEL PROGRAMMING AND CONTROLLING PARALLEL COMPUTATIONS ON COMPUTER SYSTEMS

V.Kutepov and V.Falk

*Department of Applied Mathematics of Moscow Power
Institute, Krasnokazarmennay 14, Moscow, USSR*

ABSTRACT

In the paper the main aspects of the same name project being realized at the Department of Applied Mathematics of Moscow Power Institute are outlined.

The project is aimed to integrate the functional, logical and data flow parallel programming and to develop the tools in order to support the process of automated programming and controlling parallel computations on different configuration distributed computer systems.

1.Introduction

The problem of integration of different programming styles (functional, logical, object-oriented etc.) and tools for computer aided developing programs and controlling parallel computation on computer systems is the main feature of many projects of the fifth generation computer systems and new information technologies[1]. However, the problem has at present more questions then answers.

Firstly, it is not clear what is the best way of the unification of different programming styles and languages. For example, the attempts of unification functional and logical programming languages by means of logical constructs complicates significantly the operational semantics and execution of programs based on various resolution procedures [2,3]. As a result, many advantages of functional languages (simple parallel applicative semantics, lucid compositional program form, etc.) have happened to be lost. At the same time the logical specification and inference rules are necessary when a functional program should be verified or tranformed to another equivalent form.

Secondary, many computer scientists underline that new difficulties are added to the process of parallel programming and new programming languages should be invented to express explicitly parallelism of computations. The various languages based on the data flow model parallel computations are the examples of an implementation of this requirement in practice. In spite a programmer needs new philosophy to use efficiently these languages the explicit representation of parallelism gives him opportunity to display and vary directly the degree of parallelism in program being designed.

At the same time many functional and logical languages have a parallel operational semantics resulting in implicit representation of parallelism in a program. However, the problem of designing the efficient parallel program in this case becomes more difficult and intrinsic. At present it isn't clear how to unify the advantages of the both approaches to parallel programming.

Thirdly, many new methods and programming tools were invented last decade to enhance the productivity of programming. A top-down technology

of designing programs, different verification strategies, knowledge bases and expert systems for mathematics deserve the special attention. Although these innovations extend significantly the computer aided technologies of programming, nevertheless, they are yet used as isolated mechanisms and should be integrated in order to give the effect they are worth.

Finally, the problem of realization of parallel computations on computer systems needs a short discussion. It seems a paradoxical, but today the software for controlling parallel computations is very specialized and strongly dependent on used programming language and architecture of a computer system. This is obvious that programming tools for running parallel programs on Cray computer should be distinct from those that are used for distributed computer systems, for example, transputer systems. But this is not obvious that different distributed multiprocessor and multicomputer systems should have the different operational systems to control parallel computations. However, the general application software for controlling parallel computations on distributed computer systems waits the time of its creation.

In the following we describe the main features of "mini" project called "Integrated tools for functional, logical and data flow parallel programming and controlling parallel computations on computer systems", which have grown out our own attempts to find the compromise for practical solution of outlined problems.

The tools are developed as two independent parts, one of which is intended to support automated designing of functional, logical and data flow programs. The tools of another part should allow efficient controlling the parallel execution of the programs on distributed computer systems, in particular, on the multicomputer and multiprocessor computer systems.

2. Tools for automated programming

The constituent elements and general organization of software for automated programming and computer aided designing of parallel programs depicted in Fig.1.

The programmers use personal computers as intellectual terminals to interact with programming tools which are implemented on VAX-computer. In general, the group of programmers may simultaneously interact with the software of Fig.1 cooperating their efforts in order to solve a complicated problem.

Functional, parallel graph-scheme or logical programming language may be chosen by user in respect to the nature of problem he is going to solve. These languages have been realized as independent programming systems long before the project was started and they are integrated via the tools for automated development of programs (see Fig.1). These tools are intended to support the process of decomposition the complex problem into subproblems and top-down strategy of developing programs. They can also be used to verify program being designed and evaluate its complexity (the parallel execution time) or transform the program into suitable equivalent form. In order to do these problems tractable a special theoretical work has been done relating to the used programming languages [4-9]. The functional and parallel graph-scheme languages are new ones and have resulted from our numerous trials and errors to devise

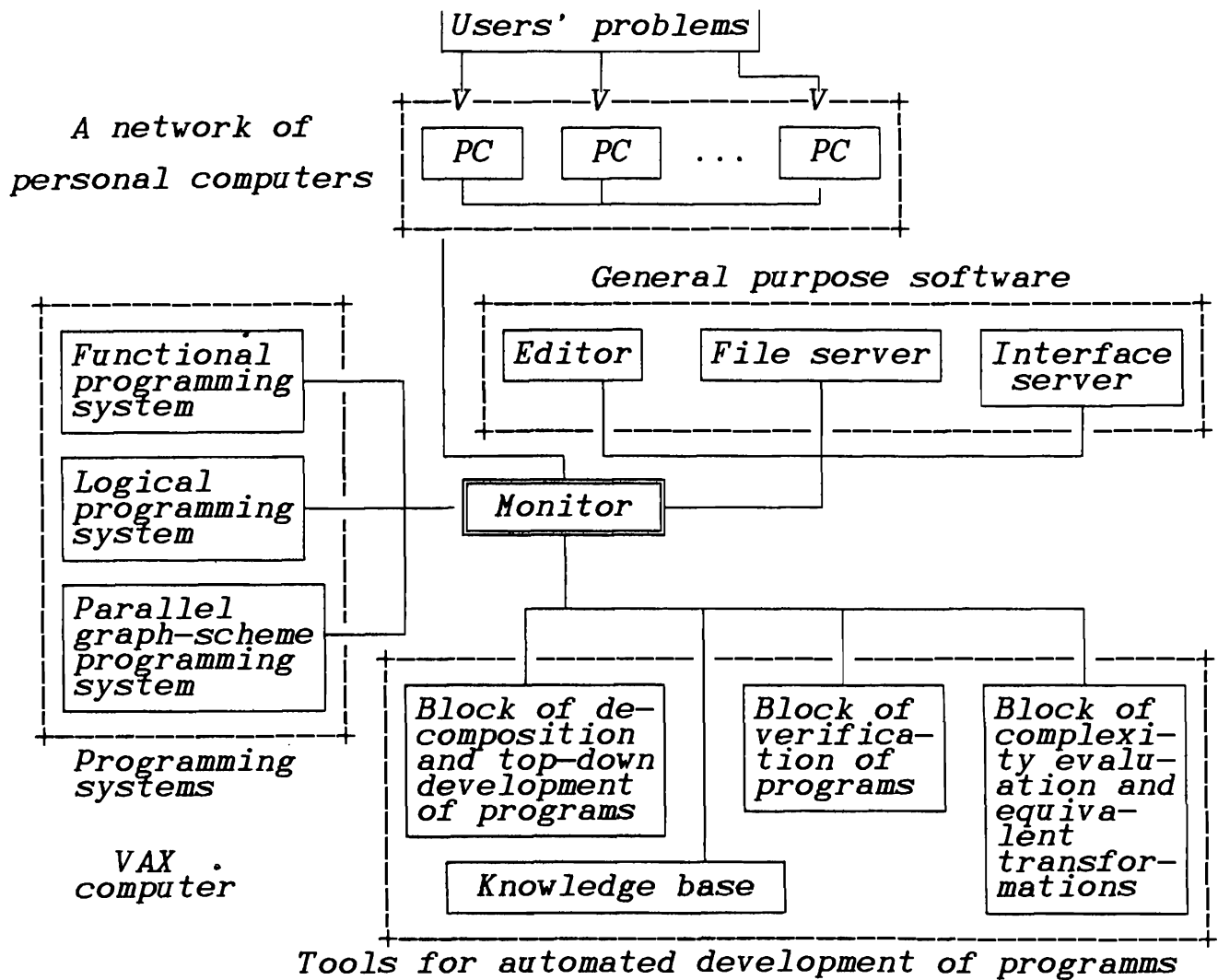


Fig.1. Tools for automated programming.

the parallel programming languages with a rich syntactical forms aimed to represent explicitly the structure and parallelism of a program by means of its scheme. The functional language has also the means for data and type definition, a static type-checking algorithms and parallel operational semantics independent of denotational semantics.

The functional language is combinatory by its nature in contrast to the languages based on lambda-calculus, eg. LISP, ML.

A set of simple binary operations over functions are used in it to combine new functions from existing ones:

{ . , -> , * , ++ , [+] }.

The syntactic form and semantics of the operations are given below.

We use the symbols f' , f'' and P to denote the existing functions and predicates respectively ($\neg P$ denotes negation of P), f denotes new defined function. The semantics of the operations is introduced via definition of the application function to its arguments.

Syntax and semantics of the operations

1. If $f=f'.f''$ then $f(x)=f''(f'(x))$.
2. If $f=f'*f''$ then $f(x)=f'(x), f''(x)$,
that is the value of $f(x)$ is ordered pair of values of $f'(x)$ and $f''(x)$ if they both are defined, otherwise $f(x)$ is undefined.
3. If $f=P \rightarrow f'$ then $f(x)=f'(x)$, if $P(x)=true$ and $f(x)$ is undefined if $P(x)=false$ or if $P(x)=false$
or if $P(x)$ or $f'(x)$ or the both are undefined.
4. If $f=f'++f''$ then $f(x)$ is one of the values of $f'(x)$ or $f''(x)$ which is defined, otherwise

$f(x)$ is undefined. In order to guarantee the functionality of f the operation $++$ is allowed

to combine only disjoint functions: f' and f'' are disjoint if at most one value $f'(x)$ or $f''(x)$ is defined.

For example, the functions $P \rightarrow f'$ and $\neg P \rightarrow f''$ are disjoint and function $(P \rightarrow f')++(\neg P \rightarrow f'')$ represents well known condition operator *if $P(x)$ then $f'(x)$ else $f''(x)$* .

5. If $f=f'[+f]f''$ (iteration) then, by definition,
 $f=f''++f'.f''++f'.f''++ \dots$

Analogously to $++$, the operation $[+]$ should combine disjoint functions to guarantee the functionality of f . For example, operator *while $P(x)$ do $f(x)$* is written as iterative function $(P \rightarrow f')[+](\neg P \rightarrow e)$ where e is identity function: $e(x)=x$.

The operations $.$, $++$, $\rightarrow$ and $*$ are associative, $++$ is also commutative. The operations $*$ and $\rightarrow$ are parallel, because simultaneous evaluation of the values combined with these operations functions is possible.

The operation $[+]$ is inheritably parallel: sequential evaluation of values $f'(x)$ and $f''(x)$ for $(f'[+f''])(x)$ is not generally correct (for theoretical treatment this and other problems see [4-6]).

For data definitions the operations of cartesian product and union of sets, application of constructors to data and the recursion are used in the language,

The functional program is a sequence of definitions of data and functions represented generally by systems of recursive equations and interpreted as minimal fixpoints over data or function domains respectively. The parameterized definitions of data and functions are allowed and nested

definitions and modules (with the imported and exported constructs) may be used a programmer for the better organization of a program.

The language is *statically typed* and has the efficient type-checking algorithm. Presently, there are two modes of realization of language: interpretative one which is used to design and execute functional programs on personal computers and parallel one which is used to run in parallel the functional programs on distributed computer systems, in particular on the multiprocessor and multicomputer systems. The parallel realization of the language is rather complicated and based on tree like model parallel process representation[5].

Theoretical results on complexity of the parallel computations of functional programs and their directed equivalent transformations which were received earlier [4,6] have been used for designing the tools for automated development of functional programs (see Fig.1).

The parallel graph-scheme programming system uses modular programming language which is based on the explicit representation of parallelism in a modular program by means of its graph-scheme [7-9]. The modules are interpreted as many inputted and many outputted procedures or subprograms of available conventional languages (Fortran, Algol, C or others). The arcs connecting the modules of graph-scheme used as the buffers for data transmission between the modules. As the consequence, the parallelism of data independent modules and data flow or assembly line parallelism can be expressed simultaneously and flexibly in a program. Being realized initially on multicomputer systems (conventional computers connected via channels) this language has been used with the high efficiency in practice to speed up solution of the systems of linear and differential equations, modelling distributed real-time computations and many other problems.

In order to simplify the planning of parallel computations and minimize overhead in a system due to controlling functions and interacting and data transmitting between computers the program is mapped onto computers by preliminary decomposition of its graph-scheme into subschemes.

The logical programming system is an implementation of the first order predicate language by means of Prolog programming system. It uses the original algorithm of parallel inference supporting also its efficient sequential running on uniprocessor conventional computers, in particular, personal computers.

Thus, the first order logic and Horn clausal logic (Prolog logic) can be simultaneously supported by the system giving a user opportunity to choose any formalism with respect to the nature of a logical problem he is going to solve.

This system is also aimed to be used for verification of the functional and parallel graph-scheme programs. To do this it is necessary to transform the program into logical form firstly. In order to simplify the process of verification the special inference rules are introduced which are based on the inductive strategy of proving very similar to Hoare logic for sequential programs.

Three blocks for automated development of programs shown in Fig.1 provides a user the functions supporting

-top-down stepwise process of designing program, storing and

displaying at the screen of personal computer a history of program development,
 -verification of each step of program development or resulted program,
 -evaluation of complexity characteristics of being designed program (the time of parallel execution of a program on computer system with infinite or finite number of processors or computers),
 -performance of directed equivalent transformations of the program in order to improve its characteristics.

The knowledge base represents at present time a simple experimental expert system for the solution of the problems of linear algebra.

A user may inquire the knowledge base about theory, methods (in particular, parallel methods) of the solution of different problems linear algebra and receive appropriate functional, logical or parallel graph-scheme programs.

The system is implemented on the multiwindow base, it uses the graphical representation of information widely and has the facilities of cooperating the efforts of a group of programmers, attacking the complicated problem. We expected that the system will significantly enhance the computer aided process of automated designing of programs.

3. Tools for controlling parallel computations on computer systems

The functional structure of the software for management parallel computations on computer systems is shown on Fig.2.

The most important objectives in designing of these tools are given below.

1) The software must be able to run in a wide variety of configurations of *multicomputer and multiprocessor computer systems*, types and characteristics of their components. In order to simplify the tuning of the software in respect to concrete computer system a specialized language was constructed for structural description of computer system. This description is used by processors or computers of system for controlling functions and re-created dynamically following all changes in computer system configuration because of failures and recoveries of its components (block of "Error detection and reconfigurations facilities" in Fig.2).

2) *Controlling functions in computer system may be centralized or decentralized* depending on its application and partition into subsystems. In accordance with this, the various strategies of distributed control may be implemented in computer system. At maximal level of decentralization each computer or processor may perform the controlling functions cooperating its efforts with other processors or computers of computer system.

The distributed (decentralized) control is more flexible and can significantly improve the reliability and efficiency of parallel computations. At the same time the implementation of distributed control needs very fast intercomputer communication network and efficient routing algorithms and protocol management.

The software of Fig.2 is designed in such a way that the different patterns of distributed control may uniformly be implemented independently of used programming language and computer system structure.

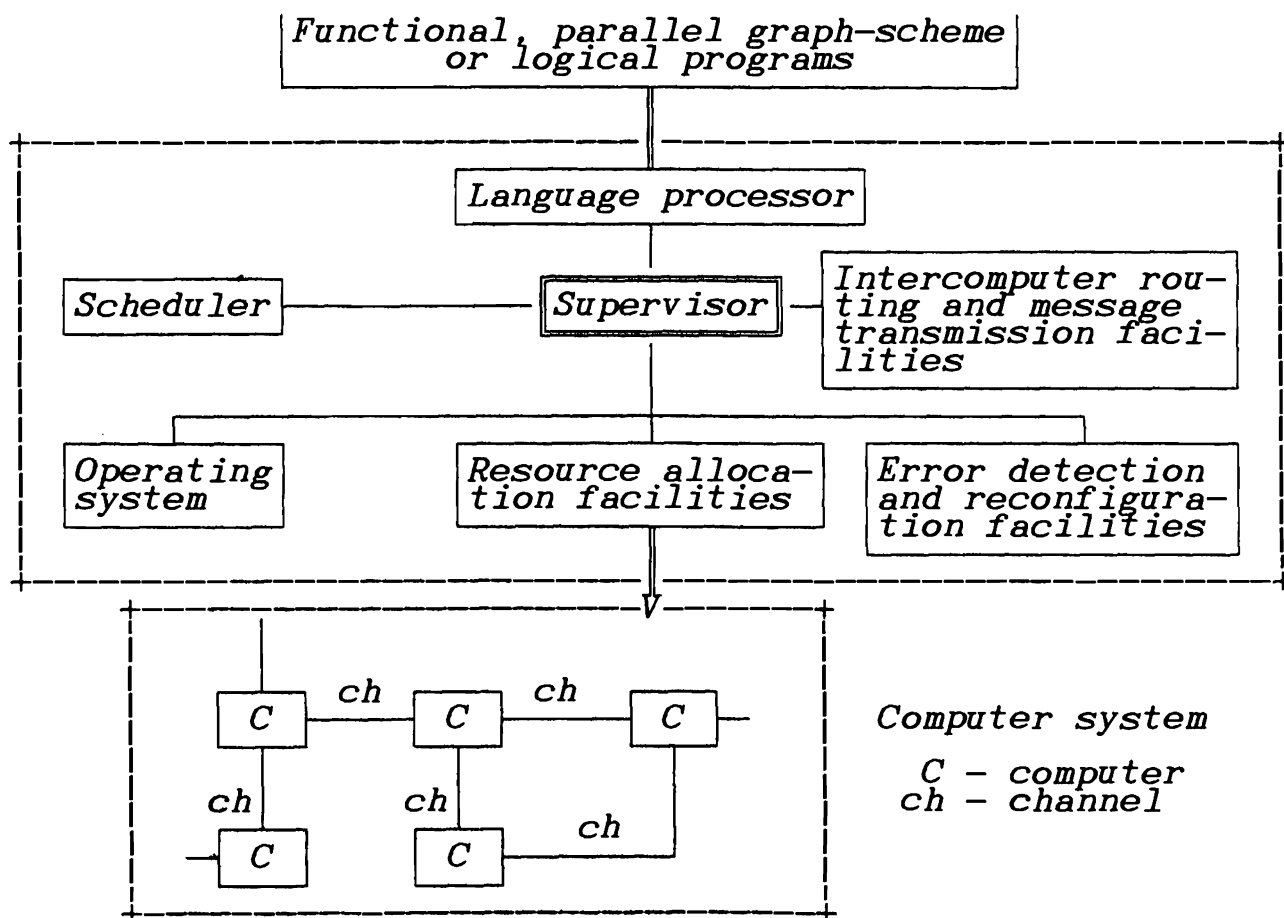


Fig.2. Tools for management parallel computations
on computer systems.

3) *In order to provide the uniformity of parallel process representation and management* in computer system during execution of functional, logical or parallel graph-scheme programs the *language processor* is added to software (see Fig.2). It supports the transitions between the states of running program in respect to its computational model and handles the processes generated during state transitions.

4) There is well known that *efficiency of parallel computations is very dependent on scheduling strategies and resource allocations algorithms* ordering the execution of the processes generated by a program.

Unfortunately, the problem of optimum planning parallel processes in computer systems has no satisfactory solution at present. In our approach to this problem we have developed *heuristic scheduling algorithms* which capture structural features of executed programs. These algorithms can be static or dynamic depending on a programming language. *For parallel graph-scheme programs an efficient static scheduling algorithms have been invented* which are based on partitioning graph-scheme into subschemes and assigning them to computers or processors of computer system before execution of a program.

For functional and logical programs the scheduling of parallel computations becomes very complicated. *The tree like computational model for these recursive by their nature programs results in dynamic and rather stochastic generating of parallel processes and needs more creative scheduling algorithms.* We have developed some satisfactory heuristic scheduling algorithms for this kind of programs, at least for functional ones. The main idea is to give the highest priority to those processes induced by running functional program for which there is known that the results of associated with the processes computations will certainly be necessary in order to form final result of program execution. This kind of a lazy evaluation strategy diminishes unnecessary computations and overhead process management. The scheduler (see Fig.2) realized the planning algorithms also taking into account the load of computer system resources to improve controlling functions.

5) *The routing algorithms which determine the paths of data and control information transmission between processors or computers of a computer system present another very important problem of controlling parallel computations.* The routing algorithms must be able to adapt to change of a configuration of computer system because of failures and traffic patterns. An intercomputer (or interprocessor) interaction time is also very depended on a used protocol organization. The block "Intercomputer routing and message transmission facilities" supports the routing and protocol functions in computer system adapting to its configuration, network characteristics and traffic patterns.

6) In order to simplify the realization of controlling software of Fig.2 *there is assumed that many standard tools of conventional computer software should be used for performing controlling functions.*

In particular, it is taken for granted that the operating system of involved computers or processors of computer system should perform all functions, associated with execution of the processes allocated to them.

The software of Fig.2 is organized on modular basis and is programmed in programming language C.

4. Summary

Besides purely scientific interest, the outlined project of integration of different approaches to the developing and programming parallel algorithms and unifying tools for controlling parallel computations on computer systems has also tutorial value. It is assumed that the students, specializing in computer system organization, will have opportunity to study in practice the basic mechanisms involved in a complicated process of designing parallel algorithms and programs and investigate the efficiency of their execution on computer systems.

5. References

1. *Proceeding of international conference on fifth generation computer systems*, ICOT, Tokio, 1984 and 1988.
2. Hassn Ait-Kasi and Roger Nasr, *Integrating logic and functional programming. LISP and symbolic computation*, 2, 1989.
3. Darlington J., *The unification of functional and logic languages in Logic programming: functions, relations and evaluations*, ed. Debroot D. and Lindstrom G. (Prentice-Hall, 1986).
4. Kutepov V. and Falk V., *Functional systems: theoretical and practical aspects*, Kibernetika, 1, 1979.
5. Kutepov V. and Falk V., *Models of asynchronous computations of function values in language of functional schemes*, Programmirovaniye, 3, 1978.
6. Kutepov V., *The calculus of functional schemes and parallel algorithms*, Programmirovaniye, 6, 1976.
7. Kutepov V. and Korablin Yu., *The language of graph-schemes of parallel algorithms*, Programmirovaniye, 1, 1978.
8. Stroeve T. and Falk V., *The asynchronous computational nets (ACN): ACN models and ACN programming system*, Kibernetika, 3, 1981.
9. Borzdova T. et al., *The multilanguage programming system based on a set of functional languages*, Programmirovaniye, 2, 1984.

PARALLELIZED QUICKSORT AND RADIXSORT WITH OPTIMAL SPEEDUP

DAVID M. W. POWERS[‡]

Fachbereich Informatik, Universität Kaiserslautern
6750 Kaiserslautern, West Germany

ABSTRACT

This paper introduces a parallel sorting algorithm based on QuickSort and having an n -input, n -processor, time complexity of $O(\log n)$ exhibited using a CRCW PRAM model. Although existing algorithms of similar complexity are known, this approach leads to a family of algorithms with a considerably lower constant. It is also significant in its close relationship to a standard sequential algorithm. It is also interesting in that it was discovered as a possible CONG execution of the standard PROLOG quicksort. We also address some hardware considerations and show how the techniques promise further improvements.

1 Introduction

1.1 *Sorting*

Knuth¹ notes that sorting is estimated to take up 25% of the world's computer time. With the advent of the microcomputer this may well have changed, but it is nonetheless a both practically and theoretically interesting task. Sorting, in the sense of bringing together related things, has now been subsumed by the more specific task of ordering, and has spawned an enormous number of serial sorting algorithms.

Whilst for specific cases, faster algorithms are known, in general sorting requires $O(n \log n)$ comparisons, and hence time. The algorithms meeting this expected complexity are based on ideas of either partitioning or merging, usually with the aid of explicit or implicit list and/or tree data structures. These are prototypically represented by the algorithms QuickSort and MergeSort (Knuth,1973).

[‡]The work reported was undertaken while the author was at Macquarie University NSW 2009, AUSTRALIA, and was supported by IMPACT Ltd, PETERSHAM NSW 2049, AUSTRALIA. The author is currently supported under ESPRIT BRA 3012: COMPULOG.

Logically these algorithms require two phases: placement into the tree, and extraction from the tree. In some cases, one or other of these phases can be left implicit. As the tree has logarithmic depth, and each element needs to be placed and/or extracted, the $O(n \log n)$ complexity follows immediately.

1.2 *Parallel Sorting Algorithms*

In view of this sequential result, we would hope for an optimal linear speedup to $O(\log n)$ when we move to parallel processing on n processors. In practise, however, this is not easily achieved.

The infamous AKS result², which was the first to demonstrate the theoretical feasibility of $O(\log n)$ time on $O(n)$ processing elements, is based on a sorting network entailing a fixed sequence of comparisons - but with a rather impractical constant of 6100! This is for achievable datasets ($<2^{6100}$ elements) actually worse than the performance of the older networks/algorithms due to Batcher³ - with $O(\log^2 n)$ and a subunit constant⁴.

1.3 *Parallelized Sorting Algorithms*

Several authors have noted that the standard $O(n \log n)$ serial algorithms seem to have serial constraints which prohibit linear parallelization. Knuth¹ proposed that a "strategy which uses the result of each comparison to determine what keys are to be considered next ... is inappropriate for parallel computations". Only recently has this state of affairs changed.

Cole⁵ has achieved the $O(\log n)$ bound with a variation of MergeSort - and a considerably smaller constant (normalized at 12 for best, expected and worst case). This makes clever use of a sorted sample to partition the streams to be merged into bounded chunks which may be merged in parallel, thus avoiding a linear cost in the merging/extraction process.

The obvious parallel form of QuickSort exhibits an $O(n)$ time complexity, and although a number of variants have been proposed, none have hitherto proven to have the desired $O(\log n)$ complexity⁶.

1.4 *A Parallelized QuickSort with Linear Speedup*

The present work has achieved the $O(\log n)$ bound with a variant of QuickSort⁷ - achieving a still smaller constant of 2.4 (in terms of the *expected* number of comparisons). The algorithm performs the partitioning implicitly, avoiding the linear number of overt moves normally performed at each level of the tree. The implicitly sorted elements are eventually moved to their desired position (if necessary) using standard techniques.

The algorithm can also be viewed as sorting by placement (of the pivot elements) into a binary (inorder) tree. In this case the constant can be seen to derive from the expected

depth of the tree¹, which corresponds to the number of parallel partition steps before partitioning is complete.

2 Algorithm

The problems of QuickSort and its parallel variants lie in the partition procedure – the recombination is trivial and can be implemented as an implicit concatenation step.

The problems with the partition step have three aspects: the choice of *pivot* (from or appropriate to the present partition); the division of the partition into two (one containing elements “*less than*” and the other those “*greater than*” the *pivot*); and the handling of the *equality* case (do elements *equal* to the *pivot* go into a partition, and if so which).

The choice of *pivot* is important, as a poor choice will result in little or no reduction in the size of one of the partition. In a serial context, this is responsible for the $O(n^2)$ worst case of QuickSort. In the parallel context, it reduces the utilization of processors, leading to a worst case linear number of parallel partitioning steps. If these cost linear time, this can result in a similar quadratic time execution for a parallel algorithm!

The partitioning step is normally conceived of in the context of an *in situ* partitioning, or perhaps in terms of operations on *linked lists*. In both cases the procedure tends to be serial, as the placement of an element to be moved can only be determined in relation to those which have already been processed. If the partitioning step requires linear time in the size of the partition, then the linear time for the initial partition step will dominate any later savings due to the parallelism.

The handling of equal elements can also lead to a worst case situation if elements that compare equal to the pivot are naïvely placed consistently in the same partition (always in the righthand partition in terms of the above description).

2.1 Description

The worst cases resulting from poor choice of *pivot* or poor handling of *equality* correspond to fairly common situations – and can be attacked with *ad hoc* heuristic techniques applicable only to specific distributions (e.g. use of a sample median where a linear distribution is expected), or by *randomization* techniques (i.e. the pivot is chosen randomly and any element equal to it is distributed randomly).

In the algorithm described here, the latter approach is used, and a worst case performance would imply that the *random* choices have been a perfect *oracle* for either the sort or the reverse ordering, with equal elements being consistently distributed to the same (longer) side.

Once the *pivot* is chosen, it is straightforward to decide in parallel which partition an element should go into. The problem of choosing the *pivot* can be easily solved in parallel in the CRCW model, as all processors working in a partition can simultaneously propose themselves as the new pivot by *Concurrently Writing* to an agreed location, and can subsequently obtain the actual selected *pivot* by *Concurrently Reading* that location. In

view of the preceding discussion, it is desirable that the choice of which of the contending processors succeeds in writing the values be *non-systematic* so that it resembles a *random* choice rather than a likely *oracle* for *systematic* data, viz: we require a *probabilistic* CRCW PRAM. We discuss later the reasonableness of this assumption.

The final partitioning problem concerns the moving of the partitioned data to suitable locations without meeting a sequential bottleneck. The solution proposed here, which is the primary augmentation of the algorithm, is that the movement of the elements is not performed at this stage, but sufficient information about the partitioning structure is retained to allow logarithmic time relocation of the elements in a subsequent phase.

Making the partition tree structure explicit provides this information. The algorithm exhibited here relocates the elements in unit time following two logical logarithmic phases. The first sizes the partitions, giving the *relative location* of each *pivot*. The second adds in the sizes of the left partitions at each level to give the implicit *absolute location* of each *pivot*. Each element is at some level a pivot, so it is now possible to read each element in parallel from its original location and relocate it directly to the specified *absolute location*.

This provides the motivation for the first form of our parallel sorting algorithms, which for clarity we expose in a Pascal-like language. We omit proofs of correctness as the first phase is only a minor variation of the standard QuickSort or TreeSort algorithms, and the subsequent phases are not at all novel. The only 'tricks' are the CRCW voting and the implicit rather than explicit partitioning.

2.2 Code

In this section, extracts from the code of a procedure implementing the described algorithm are shown, and in the following section results from an instrumented version are given as empirical confirmation of the number of parallel steps required. Note that as no machine implementing the (probabilistic) CRCW PRAM model is at present available, the algorithm has been simulated in a sequential environment and the *Concurrent Read* and *Concurrent Write* steps are expressed in separate pseudo-parallel loops - the *par* in the code shown is precisely equivalent to a PASCAL *for* executed in *random* order. The *par* and *for* statements are all to be executed in parallel.

The basic data structure assumes that we have direct access to the element associated with a given processor as *elt[p]*, and variables belonging to another processor using a record-like notation: *proc[p].var*.

Associated with each *pivot* is its immediate *parent*, which partition of the *parent* it is in (either *left* or *right*), and pointers to the two partitions of the *next* subdivision. The initialization of these variables is not shown, but *whichis* is initialized to *right* and the remainder to *undef*. At each level each processor remembers its current *pivot* as *parent* and determines which new partition it is in with a comparison. Then with a *Concurrent Write* to the corresponding *next* pointer of the *parent*, it nominates itself as the *pivot* for this new partition. The result of the ballot is discovered with a *Concurrent Read* and stored away as its new *pivot*.

```

repeat                                     { ** PHASE A ** form binary tree by partitioning}
  finish:= true;

  par r:= n-1 downto 0 do                  {random synchronization}
    with proc[r] do
      if pivot <> r then begin
        finish:= false;
        if elt[pivot] = elt[r]             {fair<= >= comparison}
          then which:= boolean(random(2))
          else which:= elt[pivot] < elt[r];
        parent:= pivot;
        proc[pivot].next[which]:= r;        {concurrent write}
      end;

  for p:= n-1 downto 0 do                  {selection synchronization}
    with proc[p] do
      if pivot <> p then
        pivot:= proc[pivot].next[which];   {concurrent read}

until finish;

```

Figure 1. Procedure powersort (PHASE A)

At the end of this process, the partition tree is explicitly represented with bidirectional pointers, *parent* and *next*. The number of parallel *fair* comparisons is clearly equal to the height of this tree, namely an expected $2.4 \log n$.

```

while proc[root].count = undef do        { ** PHASE B ** calc partition size}
  begin

    for p:= n-1 downto 0 do
      with proc[p] do
        begin
          if next[left] = undef then
            pos:= 0
          else
            if proc[next[left]].count <> undef then
              pos:= proc[next[left]].count;
            if pos <> undef then
              if (next[right] = undef) then
                count:= pos + 1
              else
                if (proc[next[right]].count <> undef) then
                  count:= pos + proc[next[right]].count + 1
            end;
          end;
        end;
      end;
    end;
  end;
end;

```

Figure 2. Procedure powersort (PHASE B).

In the next phase we first calculate (bottom up) the size of each partition which gives us the relative position of each element (*pivot*) in the partition of which it is the pivot. The

variable *root* is initialized to the index of the first pivot chosen. The leaves have trivially defined counts, and calculation of further counts is possible as the defined count condition propagates up the tree. The parallel overhead here is clearly also proportional to the height of the tree.

For the final pass, it is necessary that the *root* processor (owning the first *pivot*) has *pivot* and *parent* initialized to *undef.* and -1 respectively, indicating that it is the root and has the imaginary element -1 to its left (recall that we initialized the input partition as a right partition). The undefined elements are those which are processed and defined on each iteration (working top down). The defined absolute position of the parent plus one (since we count from zero and need to count the parent too) is added to the processed element's position relative to the parent. This phase also requires time proportional to the height of the tree.

The final move of each element from the input vector to the output vector is also shown. This is clearly achievable in constant parallel time.

```

repeat                                     { ** PHASE C ** calc absolute position}
  finish:= true;

  for p:= n-1 downto 0 do
    with proc[p] do
      if pivot = undef then
        begin
          pos:= pos + parent + 1;
          proc[next[left]].parent:= parent;
          proc[next[right]].parent:= pos;

          proc[next[left]].pivot:= undef;
          proc[next[right]].pivot:= undef;
          pivot:= p;
          finish:= false;
        end;
      until finish;

      for p:= n-1 downto 0 do sort[proc[p].pos]:= elt[p];
    end;
  end;

```

Figure 3. Procedure *powersort* (PHASE C).

2.3 Results

The parallelized QuickSort procedure '*powersort*' was run on a series of distributions with results for Phase A that accorded well with the expected $2.4 \log n$ parallel comparison time. Note that the algorithm and results for Phases B and C have been included only for completeness as the logarithmic complexity of parallel counting operations is well known. For this reason, no effort has been made to enforce simultaneous simulated parallel instances of the **for** loops, and the sequential simulation results in fortuitous premature

propagation of counts and positions within a single iteration of the main loop, so on average half the degenerate subtrees are pruned away faster.

Table 1. Empirical measurements.

Size n	Distribution	Phase A		Phase B		Phase C	
		c	$c/\log n$	c	$c/\log n$	c	$c/\log n$
25	sorted	7	1.8	5	1.3	7	1.8
25	reversed	9	2.3	7	1.8	5	1.3
25	normal	10	2.5	5	1.3	7	1.8
25	linear	7	1.8	4	1.0	6	1.5
25	quadratic	8	2.0	6	1.5	5	1.3
25	quartic	8	2.0	4	1.0	7	1.8
100	sorted	12	2.0	8	1.3	8	1.3
100	reversed	12	2.0	8	1.3	11	1.8
100	normal	12	2.0	7	1.2	8	1.3
100	linear	15	2.5	7	1.2	10	1.7
100	quadratic	14	2.3	8	1.3	8	1.3
100	quartic	11	1.8	7	1.2	7	1.2
400	sorted	19	2.4	14	1.8	11	1.4
400	reversed	16	2.0	12	1.5	13	1.6
400	normal	16	2.0	9	1.1	10	1.3
400	linear	24	3.0	12	1.5	16	2.0
400	quadratic	20	2.5	9	1.1	13	1.6
400	quartic	16	2.0	9	1.1	10	1.3
1600	sorted	24	2.4	14	1.4	15	1.5
1600	reversed	21	2.1	13	1.3	15	1.5
1600	normal	20	2.0	12	1.2	12	1.2
1600	linear	25	2.5	16	1.6	15	1.5
1600	quadratic	27	2.7	14	1.4	17	1.7
1600	quartic	21	2.1	12	1.2	13	1.3

3 Discussion

3.1 Parallelized QuickSort

The algorithm, analysis and results here all indicate that QuickSort can easily be executed in logarithmic expected parallel time. (The expected constant is 2.4, the best case constant is 1.0, and the worst case is actually linear.) The simple CRCW algorithm of the first phase is the key, and there is considerable scope for utilization or relocation of the elements in ways other than that used here in the subsequent phases.

To give an example of such an alternative, it is noted here that this algorithm is actually a biproduct of research into Parallel Logic Programming^{8,9}. The input in this case is a list, and the output is a relinking of this list. Preliminary results indicate that the standard PROLOG QuickSort admits a logarithmic expected parallel logical inference complexity when executed on the CONG system, that the unification involved could also be executed in logarithmic parallel time¹⁰, and that heuristics can be added to a parallel theorem prover which will guide it into such a proof.

In fact another, still faster, algorithm has been discovered by examining the CONG execution further, and modelling it more closely: instead of translating it into an *in situ* style, retaining the linked list data structure. In this case it turns out that only the pass A (with small modifications) is necessary to produce the sorted list. But again, an additional pass is necessary to bring the i 'th element to the i 'th processor.

3.2 Probabilistic CRCW Hardware

It was noted earlier that no CRCW parallel computer was available for the empirical verification of the algorithm. The use of a probabilistic CRCW model as opposed to CREW is not itself a bar to achieving the expected efficiency, as it can be implemented as efficiently as any other.

With regard to shared memory models, we note that for the large number of processors necessary to make this algorithm practical ($O(n)$), shared memory implementations are already problematic. However, we note that the circuitry to avoid addressing conflicts in CREW can as easily implement CRCW in any desired form. Implementing probabilistic CRCW given priority CRCW, for example, involves merely permuting the inputs to the priority encoder on each cycle. It is also, of course, straightforward to implement the algorithm using indivisible semaphore or fetch-and-add instructions.

In the case of a distributed model, since many processors send their messages to the same *pivot* processor, the network can detect that it has identical addresses at each stage of *funneling* and probabilistically choose amongst them.

3.3 Network Overheads

A more serious bar to achieving the logarithmic time is the additional overhead of communication cost between processors, or processors and memory. Assuming that it is necessary to use constant-degree processor networks with serial communications, this introduces an $O(\log^2 n)$ overhead according to the conventional wisdom.⁴ By contrast Batcher's Bitonic Exchange requires only constant-degree communication and thus incurs only an $O(\log n)$ overhead, and with its smaller constant will always be ahead.

However, the $O(\log^2 n)$ overhead is not a lower bound, and using a pipelined $O(\log n)$ wormhole communication scheme, the present algorithm would be well ahead of Batcher's by $n = 10^6$ elements/processors. This scheme involves nodes forwarding the data packet with only a single bit delay for each dimension. Thus the serialization overhead in the data is of the same order as the routing overhead, and additively rather than multiplicatively combined.

3.4 Parallelized RadixSort

One major disadvantage of QuickSort in general is its terrible worst case performance. As discussed above, the problem can largely be avoided by randomization techniques, but we can never guarantee that the unlikely 'oracle' behaviour will not occur.

In the presentation of the algorithm, it was implicitly assumed that the communication costs were constant. In fact there is a tacit charitable assumption that we may assume just that, at least for communication between neighbouring processors. But in practice communication costs are proportional to the data length – in this case to the size of our keys. There is also a $\log n$ overhead in communication costs in a distributed network.

We saw in the above section that Batcher's Bitonic Exchange could avoid the distribution overhead as it communicates with only nearest neighbours in a hypercube. Radix Sort has the potential for avoiding the overhead in the size of the keys.

In RadixSort¹, instead of choosing an element as pivot, an implicit pivot can be used by making the *left-right* decision on the basis of successive bits. Suppose the keys are $w < 2.4 \log n$ bits long. This then guarantees the tree will have a height less than the expected height for QuickSort.

Furthermore, it is now no longer necessary to transmit entire keys around the network. Thus this binary form of RadixSort may be a viable alternative even for other values of w . For any w the algorithm (ignoring communication overheads) is *worst case* $O(w)$. Considering respectively the conventional and proposed networks discussed in the last section, whereas QuickSort will have respectively a $w \cdot \log n$ or $\max(\log n, w)$ overhead hidden in the communication costs, RadixSort will reduce this potential deterioration of the communication to $\log n$ in either case, and will thus *in both cases* have a true worst case $w \cdot \log n$ complexity. On the other hand QuickSort would have true respective expected complexities of $2.4 w \cdot \log^2 n$ or $2.4 w \cdot \log n$ (making the reasonable assumption that $w > \log n$ – otherwise you have an instant perfect hash!)

Thus, even on the 'best' hardware, the worst case RadixSort matches the best case Quicksort and is several times better than the expected case, which is still better than the *best* case for any other known general algorithm! On present day hardware, QuickSort will actually be $\log n$ slower than its order would suggest, and thus about 5 times slower than BitonicSort. However, RadixSort should not deteriorate (if single bit messages are possible), and will be $0.5 \log n$ faster than BitonicSort, and like it – and unlike Quicksort – have no looming linear worst case.

References

1. M. Ajtai, J. Komlós & E. Szemerédi, An $O(\log n)$ sorting network. *Combinatorica*, 3 #1 (1983) pp.1–19.
2. K. E. Batcher, Sorting Networks and their Applications. *Proc. AFIPS Spring Joint Computer Conference* (1968) pp.303–314.
3. R. Cole, Parallel Merge Sort. *Proc. 27th Ann. IEEE Symposium on Foundations of Computer Science* (1986) pp.511–516.
4. R. S. Francis and L. J. H. Pannan, (1989): Parallelism in QuickSort. *Proc. 12th Australian Computer Science Conference* pp.353–361.
5. C. A. R. Hoare, QuickSort, *Computer Journal*, 5 (1962) pp.10–15.
6. D. E. Knuth, Sorting and Searching. *The Art of Computer Programming*, 3. (Addison–Wesley, Reading MA 1973).
7. I. Parberry *Computing Reviews*, #8911 (1989) pp.578–580.
8. D. M. W. Powers, Implementing Connection Graphs for Logic Programming. *Cybernetics and Systems '88*, ed. R. Trappl (Kluwer, 1988) pp.957–964.
9. D. M. W. Powers, Compartmentalized Connection Graphs for Concurrent Logic Programming I: Compartmentalization, Transformation and Examples, *SEKI Report SR-90-16* (University of Kaiserslautern, FRG).
10. D. M. W. Powers, Compartmentalized Connection Graphs for Logic Programming. II: Parallelism, Indexing and Unification, *SEKI Report SR-90-17* (University of Kaiserslautern, FRG, 1990).

KNOWLEDGE-BASED PARALLEL SCHEDULING STRATEGIES IN PARALLEL LOGIC PROGRAMMING SYSTEMS

Gao Yaoqing, Wang Dingxing, Qiu Xiaolin,
Shen Meiming, Zheng Weimin and Wen Dongchan
Department of Computer and Technology
Tsinghua University, Beijing 100084, P.R.China

Hu Shouren
Department of Computer Science
Changsha Institute of Technology
Changsha, Hunan 410073, P.R.China

ABSTRACT

Logic programming offers a great deal of scope for parallelism. But the parallel execution of a task incurs various overheads. This paper places the emphasis on the parallel scheduling strategies in parallel logic programming systems. The dynamic, static and hybrid approaches are used to generate some knowledge information which guides task scheduling. A simulation system of the parallel inference machine RAP/LOP-PIM has been implemented on the VAX 11/780 and SUN workstations.

1. Introduction

Considerable progress has been made in developing efficient approaches to executing logic programs in parallel. These approaches can be classified into two categories: the first uses the "Committed Choice" paradigm to select clauses, and input/output dependencies between goals. The second focuses on exploiting AND and OR parallelism inherent in Prolog programs.

Our research goal is to design a high-speed parallel inference machine on shared-memory multi-processors, for executing logic programs in both AND and OR parallel. Our research falls into the latter category. We start from abstract levels and move step by step towards concrete implementation details^{1,2,3,5,10}. We present a new efficient parallel execution model of logic programs--RAP/LOP (Restricted And-Parallelism and Limited Or-Parallelism). In this model, the OR-forest description is used to describe the search space. Based on the RAP/LOP model, we design a parallel abstract machine and its corresponding multiprocessor architecture. A software simulation system of the parallel logic programming system RAP/LOP-PIM-SES is implemented on the VAX 11/780 and SUN workstations.

As we know, parallel processing can be generally used to improve the computational efficiency of solving a given problem, but not to extend the solvable problem space, especially when the problem space is exponentially large. Parallel scheduling is an important problem in parallel inference machines. In this paper, we place the emphasis on the knowledge-based scheduling strategies in the parallel logic programming system RAP/LOP-PIM. These scheduling strategies are general enough to be used for other parallel logic programming systems.

The remainder of this paper is organized as follows: Section 2 gives a brief overview of the parallel execution model RAP/LOP. Section 3 presents the intelligent parallel scheduling strategies. last part of this paper gives some experimental results and conclusions.

2. An Overview of the Parallel Execution Model RAP/LOP

We provide here a brief overview of the RAP/LOP execution model. The efficient of solving a logic programming problem depends strongly on the ways of describing the logic program because different description methods provides different frameworks for developing execution models of logic programs. We traditionally use the OR tree, AND tree or AND/OR tree description methods to describe the execution of logic programs. After analyzing the features of the different description methods, we present the OR-forest description. An OR-forest is a collection of several OR trees. There are two types of nodes in the OR-forest: Seed nodes which fork parallel AND goals, and ordinary nodes which are similar to those of the OR tree. Each parallel AND goal is correspond to a separate subtree.

The merits of the OR-forest description are as follows:

(1) Both AND- and OR-parallelism can be explicitly described instead of only one type of parallelism described in the OR-tree and AND-tree.

(2) Redundances which exist in the OR-tree description are avoided because the execution of each independent subgoal is described by one separate subtree.

(3) Since the branches of the AND/OR-tree reflect only the static structure of the program, in AND/OR-tree-based execution models the execution of a program is carried out by passing messages among a hierarchy of processes corresponding to nodes of the tree. The communication overhead of OR-forest-based models is less than that of AND/OR-tree-based execution models.

Consider the Example 1:

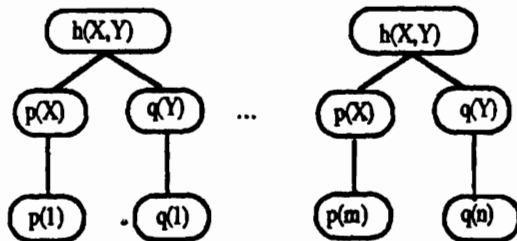
?-h(X,Y).

h(X,Y):-p(X),q(Y).

p(1). p(2). ... p(m).

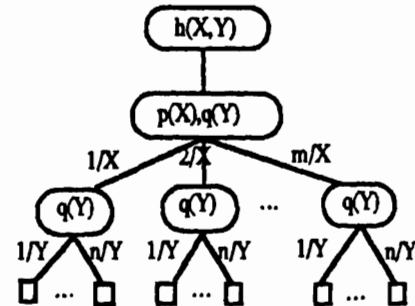
q(1). q(2). ... q(m). ... q(n).

Its different descriptions are shown in Fig. 1.

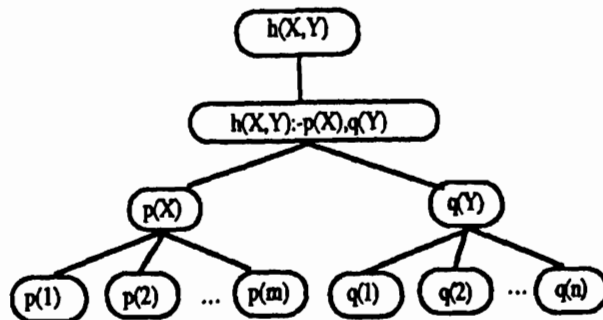


(a) The AND-tree description of Example 1.

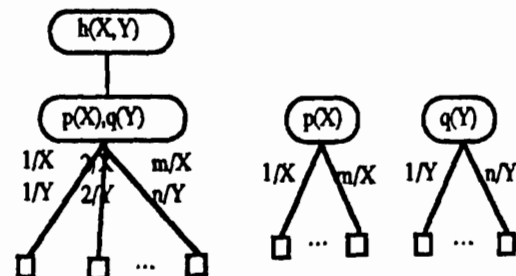
Each tree corresponds to a solution.



(b) The OR-tree description of Example 1.



(c) The AND/OR-tree description of Example 1.



(d) The OR-forest description of Example 1.

(m-1)*n redundances q(Y) which exist in the OR-tree are avoided in the OR-forest.

Fig. 1 The different descriptions of the Example 1.

In the framework of the OR-forest description, we present a parallel execution model RAP/LOP. The parallel language, detection methods of parallelism, memory management schemes and parallel scheduling strategies are involved in the parallel execution model.

In our model, computations can be considered as searching an OR-forest by a number of processes. It has the feature as follows:

1. CGE-Prolog is used which is an extension of Prolog by introducing some control mechanisms to support exploiting parallelism in logic programs explicitly and implicitly¹².

2. We regard some part of the OR-forest as a process, a node as a process instead. Therefore moderate granularity of parallelism can be exploited to reduce the overhead of control and management.

3. An approach of combining dynamic and static detection of AND-parallelism is used⁵. We predict the properties of a program at compile-time through abstract interpretation. Based on the analysis, the information such as modes is obtained to guide a precompiler to generate conditional-graph-expressions of the program. At run-time, simple checks are used to determine whether the goals in the body can be executed in AND-parallel.

4. OR-parallelism is limited by the number of processors available. When all the processors are busy, every processor runs in some part of the OR-forest and works on the candidate OR branches through backtracking.

5. The multi-stack memory management scheme is introduced in the model which uses both data coping and data sharing. The environment reduce approach is used to increase the memory efficiency.

Based on the RAP/LOP parallel execution model, we design a parallel abstract machine RAP/LOP-WAM which can be viewed as an extension of the WAM^{10,13}.

3. Parallel Scheduling strategies

3.1 *Philosophy of Scheduling Strategies*

The efficiency of a parallel inference machine depends highly on its scheduling strategy. Multi-processor scheduling is the problem of assigning tasks to processors in a parallel machine to meet some optimal criterions. In our case, the criterions are those: the degree of load balance, the efficiency of executing as necessary computations as possible, and overhead of control and management. Most multi-processor scheduling problems turn out to be NP-Complete. Therefore, in practice multi-processor scheduling is done according to some heuristic information. But in most existing logic programming systems, the simple blind search strategies such as depth-first, breadth-first and etc. are usually used. Here we present intelligent scheduling strategies. An intelligent scheduling strategy is a knowledge-based scheduling one which has the property called "work conserving" (i.e., whenever a processor becomes idle, some executable task is given to it. No processor is idle unless there are no executable tasks in the system.) and "no detrimental anomalies" (i.e., The parallel speed of a program is lower than its sequential speed).

In our system, a logic program LP consists of a source code S and control information C (or called knowledge), i.e., $LP = \langle S, C \rangle$.

The control information of a program come from: (1) User annotation. That is, users give some information in their programs by their expertise. (2) Static automatic generation. In the precompile phase, global information such as global model, and granularity is generated by abstract interpretation. (3) Dynamic automatic generation. At run-time, the statistics information, such granularity, frequently-used mode and success branch with high possibility, is collected. These statistical information collected before will guide next execution of the machine. (4) Both static and dynamic automatic

generation.

In the initial stage of software development, a programmer can write a non-annotated program (a naive user), annotated program (an expert) or combination of both. The program is compiled by the precompiler and compiler. After the code generated by the compiler running several times in a machine, the historical control information is accumulated and is fed back into the precompiler. The precompiler and compiler use the knowledge, generated by static analysis, dynamic statistic or combination of both, to produce executable code which is as time and space efficient as possible (Fig.2).

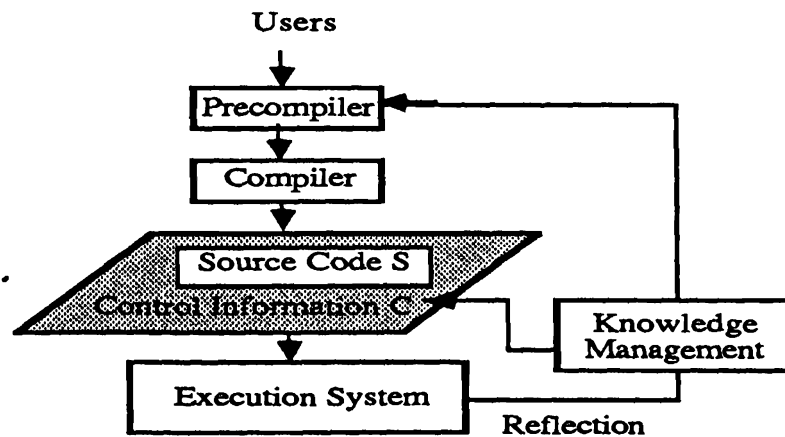


Fig.2 Reflective Architecture

3.2 Indexing and Pruning Mechanisms

Pruning is used to eliminate unnecessary expansions when it is determined that a solution tree cannot be found from a subgoal or clause.

For OR branches, efficient clause indexing is created at compile-time, which guide the goal to match the success clause with high possibility. The indexing instructions are introduced to link together the different clauses in a procedure and filter out a subset of those clauses which could potentially match a given procedure. The exclusive-OR procedures (XOR) are detected at precompile-time. For a XOR procedure and given a calling goal, at most one clause of the procedure can match it. An optimization is to find a key to guarantee that only one clause is to be unified with a calling goal every time a procedure is called.

Definition: Exclusive-OR (XOR) clause subset

Let P be a procedure, X be a clause subset of P, $C_1 (C_1^+:-C_1^-)$, $C_2 (C_2^+:-C_2^-)$ be two arbitrary clauses of X. X is an exclusive subset of P, if,

- (1) C_1^+ , C_2^+ are not unifiable; or,
- (2) $C_1^- \& C_2^-$ is contradictory.

Here, we introduce an algorithm which can detect a class of XOR procedures.

Algorithm: A XOR procedure is identified.

(1) The parameters corresponding to mode N are removed from the clause heads C_1^+ , C_2^+ . If even one pair of the modified clause heads unify, the procedure is not XOR one, and goto (2). Otherwise, the procedure is a XOR one.

(2) The first several arithmetic comparison predicates (build-in predicates) whose arguments are all ground of C_1^- and C_2^- are analyzed to find exclusive build-in predicates (a restricted scheme).

For AND goals, when one parallel AND goal in a seed node fails, the execution of its siblings will be killed immediately. The scheme is called restricted semi-intelligent backtracking.

3.3 knowledge-based Scheduling

Generating Knowledge Information. In a multi-processor system, a task can be executed in parallel does not necessarily mean, in practice, that it should be done in parallel, because the parallel execution of a task incurs various overhead, e.g. task creation and scheduling. The problem of granularity is the tradeoff between abundance of parallel tasks and the overhead of managing the tasks. Because of non-determinism of logic programs, it is more difficult to determine the granularity of parallelism. We use static, dynamic and hybrid scheme to distribute only high-granularity goals.

Static analysis: At compile-time, the source program is first converted into a cyclic AND/OR hierarchical call graph $G=(N,E)$. Here N is a set of nodes and E is a set of edges. A node in the graph denotes a literal or a clause. There are two kinds of edges in the graph: AND branches and OR branches which occur alternatively.

A procedure's granularity is the sum of the granularities of all of its clauses, and a clause's granularity is the sum of the granularities of the procedures it calls. The database predicates and build-in predicates are one by definition. Granularity estimate for a set of non-recursive predicates is relatively simple. The granularity of a self-recursive clause (as append) is represented by its recursive times. It is difficult to determine the recursive strength of a procedure. Because the work done by a call to a recursive predicate typically depends on the size of its input, e.g., term-size, list-length, etc.. The measure appropriate in a given situation generally be determined by examining the operations used in the program (the list-length is used to represent the recursive times in append). Here we assume that the recursive times of a recursive predicate is approximately equal to the length of its dominant argument. The dominant argument of a recursive procedure is defined as the one which the recursive times of the procedure depend strongly on. Therefore, the recursive times of a predicate can be determined by tracing its dominant argument size at compile-time and by statistics at run-time. The granularities of mutually recursive goals are calculated by cutting a call cycle at the lowest level in the graph and calculating the granularities of the lowest nodes.

Consider the example quicksort, its AND/OR hierarchical call graph has the cycles (see Fig.3 (a)). After cutting the cycles by computing the argument sizes, we got an acyclic AND/OR hierarchical call graph (see Fig.3 (b)). Therefore the granularity of self recursive procedure append is $\text{gain}(\text{append}(X1,[H|Y1],S))=5$, where $\text{gain}(*)$ is the granularity of $*$. $\text{gain}(\text{split}(H,T,X,Y)) = \text{size}(T)=9$, where $\text{size}(*)$ is the size of $*$. $\text{gain}(\text{quicksort}(X,X1)) = \text{size}(X)=5$, $\text{gain}(\text{quicksort}(Y,Y1)) = \text{size}(Y)=5$, $\text{gain}(\text{quicksort}([],[])) = 1$ by definition. Therefore $\text{gain}(\text{quicksort}) = (9+5+5+5)+1=25$.

Dynamic statistics: In our system, for OR parallelism, we have the consideration: the frequency of process migration increases dramatically near the end of program execution. Therefore, the most effective positions for the OR-parallel nodes tend to be higher up in the search tree. Our scheduling strategy picks tasks from the node nearest to the root of the tree.

The reflective computations are introduced in our system. For AND-parallelism, the scheduling is based on the granularity (an integer) of a parallel AND goal. The granularities are generated through collecting statistical information at run-time. Every time a program is executed, its historical control information is stored into the machine. At run-time, the new control information of an AND goal will be collected into its corresponding environment, such as statistical granularity at the i -th depth. After finishing the execution of a program, the information of each goal will be formed in a list and inserted into the corresponding goal of the program as a extra parameter. In addition to the granularity information, we collect the information such as user-frequently-used mode information for compiler to reduce code space in our system.

Hybrid scheme: The compile-time granularity analysis is a conservative estimate of the amount of work performed at run-time. In other hand, it is expensive to determine

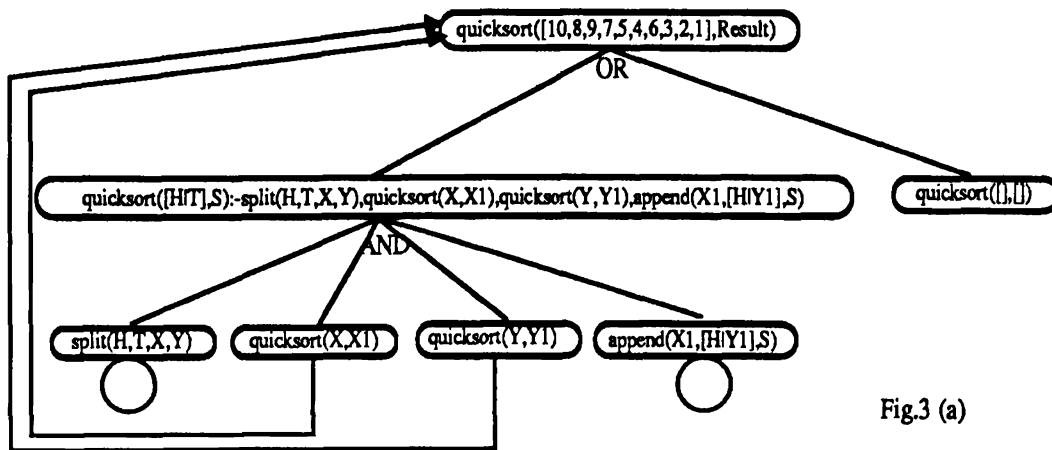


Fig.3 (a)

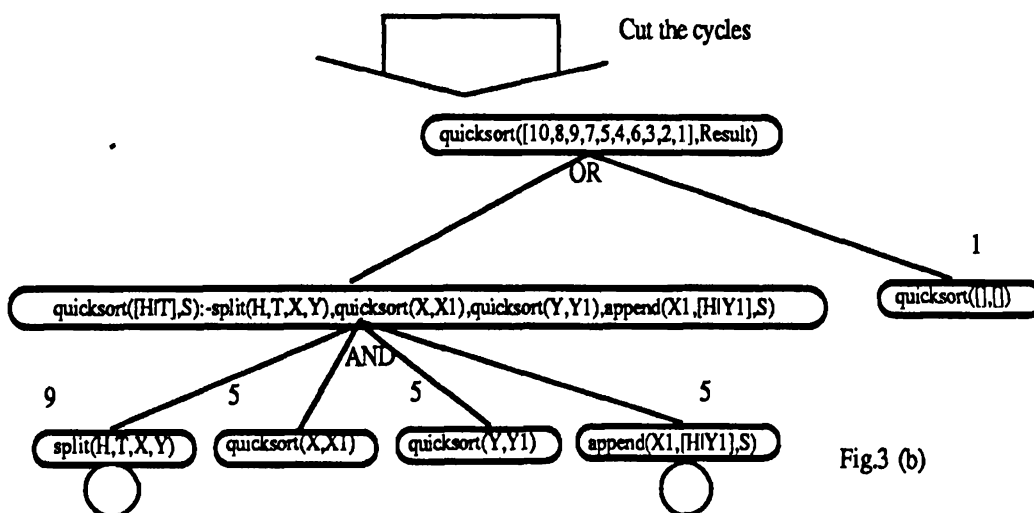


Fig.3 (b)

Fig.3 The AND/OR hierarchical call graph of quicksort

the granularity information at run-time. Therefore we can use the hybrid scheme which combine the static and dynamic schemes. In the hybrid scheme, the granularities of non-recursive procedures are first computed. Then as much of the granularity analysis for recursive procedures is done at compile-time as possible, but the actual computations of granularities are postponed until run-time. We can hire the techniques similar to ⁹.

Consider the Example 2:

C1: fib(0,0).

C2: fib(1,1).

C3: fib(M,N) :- M>1, M1 is M-1, M2 is M-2, fib(M1,N1), fib(M2,N2), N is N1+N2.

Its granularity can be computed through the following difference equations:

gainC1 = 1,

gainC2 = 1,

gainC3 = gainfib(n-1) + gainfib(n-2) + 1, n>1,

gainfib(n) = gainC1 + gainC2 + gainC3

= gainfib(n-1) + gainfib(n-2) + 1

= 2*gainfib(n-1) + 3

This yields the solution $\text{gainfib}(n) = 2^{n+1} - 1$ which can be evaluated at run-time, when the size of the dominant argument is known.

Using Granularity Information. The appropriate granularity of parallelism is determined by the cost involved in task switch, communication and scheduling of underlying multiprocessor architecture. In the RAP/LOP-PIM, the cost of task switch is

proportional to the size of the current binding array (BA). Therefore, given a goal or a clause G ,

(1) We compute the expression $T(\text{gain}(G))$, where $T(*)$ is the cost of executing a goal or a clause G whose granularity is $*$.

(2) According to the size of the current BA, we evaluate $\text{COST}(G) = \partial * \text{size}(\text{BA}) + f$, where ∂ and f are the constant factors which are determined by the underlying multi-processor architecture. $\text{COST}(G)$ is the threshold of determining The appropriate granularity of parallelism.

(3) If $T(\text{gain}(G)) > \text{COST}(G)$, G can be executed concurrently as a separate task.

In fact, the granularity information is also used to guide scheduling to solve the "trapped goal" and "garbage slot" problem¹⁴ in the stack-based memory management of the parallel logic programming system.

4. Measurements and Conclusions

In this paper, the efficient scheduling strategies in the parallel logic programming system RAP/LOP-PIM is described. The scheduling strategies uses the pruning mechanism to reduce the search space of a given problem, the granularity at compile-time and run-time to exploit appropriate parallelism.

In order to validate our model, we have implemented a simulation system of the RAP/LOP-PIM parallel inference machine in the programming language C on the VAX11/780 and SUN workstations.

Lots of benchmarks such as append, reverse, quicksort, 8queen, maze, map_coloring, matrix_multiplication small_database, etc. have been executed in our system. we use different scheduling strategies to measure the performance improvement ratio of the system. Here the performance improvement ratio of the system is defined as its speedup. The values were obtained by averaging measured values of all the benchmarks (see Fig.4). The results show that with the number of processors increasing, the performance improvement ratio of the intelligent scheduling strategies is better than that of the simple blind strategies such as the depth-first and breadth-first because the more processors are there, the more possibly the tasks with fine granularities are scheduled as separate tasks.

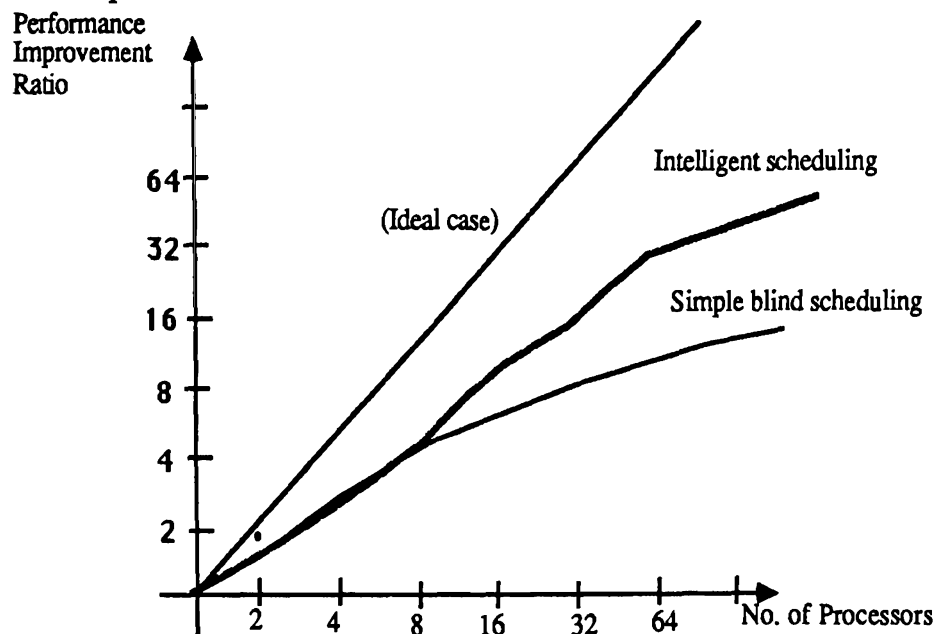


Fig.4 Performance improvement ratio due to intelligent scheduling

The substantial speedup (at least an order magnitude) are attainable for logic programs when we choose more than 16 processors.

5. References

1. Gao Yaoqing, Sun Chengzheng and Hu Shouren, *Study of a Parallel Inference Machine for Parallel Execution of Logic Programs*, Int'l Joint Conf. on Vector and Parallel Processing, Zurich, Switzerland, Nov. 1990.
2. Gao Yaoqing, Hu Shouren and Sun Chengzheng, *Design and Implementation of a Parallel Abstract Machine Model RAP/LOP-WAM for Parallel Execution of Logic Programs*, Proceedings of an International Conference on Information Processing, Tokyo, Japan, Oct. 1990.
3. Sun Chengzheng and Tzu Yungui, *The OR-forest Description for the Execution of Programs*, Lec. Notes in Computer Sci., the Proc. of the 3rd Int'l Conf. on Logic Programming. July, 1986.
4. A.Ciepielewski and S.Haridi, *Control of Activities in an OR- Parallel Token machine*, Proc. of logic programming workshop 83, July, 1983.
5. Hwang Zhiyi and Hu Shouren, *A Compiling Approach for Exploiting And-parallelism in Parallel logic Programming Systems*, Proc. of Parallel Architecture and Language, Europe, Netherlands, 1989.
6. Gao Yaoqing and Hu Shouren, *Design of a Parallel Abstract Machine Based on the RAP-PSOF Parallel Model of Logic Programs*, Proc. of Int'l Conf. on Computer and Communication, Beijing, 1989.
7. L.V.Kale, *Completeness and Full Parallelism of Parallel Logic Programming Schemes*, Proc. Fourth IEEE Symp. on Logic Programming, San Francisco, CA, IEEE, 1987.
8. E.Tick, *Compile-Time Granularity Analysis for Parallel Logic Programming Languages*, Int'l Conf. on FGCS, Tokyo, Japan, 1988.
9. S.K.Debray, Nai-Wei Lin and M.Hermenegildo, *Task Granularity Analysis in Logic Programs*, Proc. of the ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation. White Plains, New York, Jun., 1990.
10. Gao Yaoqing, *A RAP/LOP-WAM Parallel Abstract Instruction Set*, Tech. Report. Changsha Institute of Technology. 1988.
11. J.S.Conery, *The AND/OR model for parallel interpretation of Logic Programs*, Ph.D thesis, Dept. of Infor. and Computer Sci., UC Irvine, 1983.
12. Gao Yaoqing, Wang Dingxing and et al, *Development of the Parallel Inference Machine RAP/LOP-PIM and its Optimized Parallel Compiler*. COMPSAC'91, Tokyo, Japan.
13. D.H.Warren, *An Abstract Prolog Instruction Set*, Tech. Note 309, SRI International, AI Center, Computer Sci. and Technology Division, 1983.
14. M.V.Hermenegildo, *An Abstract Prolog Machine based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*, Ph.D thesis, The University of Texas at Austin, 1986.
15. B.Hausman, A.Ciepielewski, and S.Haridi, *OR-parallel Prolog made efficient on shared memory multiprocessors*, In The 1987 Int'l Symp. on Logic Programming, San Francisco, California, IEEE 1987.
16. David H.D.Warren, *The SRI Model for OR-Parallel Execution of Prolog--Abstract Design and Implementation Issues*, In The 1987 Int'l Symp. on Logic Programming, San Francisco, California, IEEE, 1987.
17. Sun Chengzheng and Ci Yungui, *PSOF: A Process Model Based on the OR-forest Description*, Proc. of the int'l Conf. on Computer and Communication, 1986, Beijing.

18. C.S. Mellish, *The automatic generation of mode declarations for Prolog programs*, DAI Research paper 163, Dept. of Artificial Intelligence, Univ. of Edinburgh (August 1981), U.K.
19. S.K. Debray and D.S. Warren, *Automatic Mode Inference for Prolog Programs*, Journal of Logic Programming, 207-229, Sept. 1988.
20. D. DeGroot, *Restricted And-parallelism*, Proc. of the Int'l Conf. on FGCS, Tokyo, (Nov. 1984).
21. D. DeGroot, *A technique for compiling execution graph expressions for restricted And-parallelism in logic programs*, Proc. of the 1987 Int'l Supercomputing Conf., Athens, Greece, (June 1987).
22. J.-H. Chang, A. M. Despain and D. DeGroot, *And-parallelism of logic programs based on a static data dependency analysis*, COMPCON 85, San Francisco, Feb., 1985, pp. 218-225.
23. Tung Yu-Wen and Dan I. Moldovan, *Detection of And-parallelism in logic programming*, Proc. of the 1986 Int'l Conf. on Parallel Processing, IEEE, Pennsylvania, 1986, pp. 984-991.
24. K. Muthukumar and M. Hermenegildo, *The CDG, UDG and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism*, Technical Report ACA- ST-023-90, MCC, Austin, TX 78759, 1990.
25. C.S. Mellish, *Abstract Interpretation of Prolog Programs*, In 3rd International Conf. on Logic Programming, Imperial College, Springer-Verlag, July 1986.
26. D.S. Warren, *Efficient PROLOG Memory Management for Flexible Control Strategies*, in Proc. of 1984 Int'l Symp. on Logic Programming, Atlantic City.
27. K. Ueda, *Guarded Horn Clauses*, Ph.D. Thesis, Information Engineering Course, Faculty of Engineering, University of Tokey, 1986.
28. Lusk, E., Warren, D.H.D, Haridi, S., Butler, R., Calderwood, A., Disz, T., Olson, R., Overbeek, R., Stevens, R., Szerdi, P., Brand, P., Carlsson, M., Cipelewski, A. and Hausman, B., *The Aurora OR-Parallel PROLOG System*, in Proc. of Int'l Conf. on FGCS 1988.
29. Baron, U., de Kergommeaux, J.C., Hailperin, M., Ratcliffe, M., Robert, P., Syre, J.C. and Westphal, H., *The Parallel ECRC PROLOG System PEPSys: an Overview and Evaluation Results*, in Proc. of Int'l Conf. on FGCS 1988.
30. K. Muthukumar and M. Hermenegildo, *Methods for Automatic Compile-time Parallelization of Logic Programs using Independent/Restricted And-parallelism*, Technical Report ACA-ST- 233-89, MCC, Austin, TX 78759, March 1989.

VERY-HIGH-LEVEL LANGUAGE PARIS

Korneev V.D., Mirenkov N.N., Nepomniaschaya A.Sh.

*Computing Center, Siberian Division of USSR Academy of
Sciences, Novosibirsk, 630090, USSR*

ABSTRACT

A very-high-level language PARIS related to AI applications and their parallelization is considered. One of the principal target applications is the efficient interpretation of other well-known very-high-level languages. Among aims of the project PARIS is the formulation of requirements for future parallel machines which comprise units of different kinds of parallelism and allow direct manipulation of composite objects.

1. Introduction

Very-high-level languages LISP, APL, PROLOG, SEQUEL-2, SETL and some others have rich expressive features. Therefore they can be used as convenient facilities for program specification. It is not evident however that these languages can be regarded as facilities for efficient programming. On the one hand, this is related to their expressive features and, on the other hand, this is often related to their sequential machine realization. But it is well known that parallel machines like associative computers have architectures allowing efficient manipulation of composite objects such as vectors, matrices, sets and others ¹⁻³.

So, we came to the idea of trying to design a parallel computing system for which very-high-level (VHL) languages may be used as the programming means. In order to realize this idea we are performing three-step plan:

- designing the VHL language PARIS oriented to AI applications;
- realization of a pilot system on parallel computing complex "Siberia" ⁴ including associative units;
- formulation of the requirements for architectures of future parallel machines.

In this paper we consider some aspects of the first and second

steps of the plan.

The basis of PARIS is the construction of kernel and different problem-oriented shells over it. The PARIS kernel has been chosen to allow efficient interpretation of well-known VHL languages. In order to meet it we have considered the languages LISP, APL, PROLOG, SEQUEL-2, SETL⁵⁻¹¹ and made special "selection" of constructs. Besides we added some new extensions. Therefore PARIS can be considered as an intermediate language for the interpretation of other VHL languages. To underline the basic role of data types and constructs of SETL we named our language PARIS following the abbreviation of PARAllel Interpretation of SETL. Unlike SETL, in PARIS the following constructs are important:

- set transformer and tuple transformer,
- operations for object names appearing as the tuple elements,
- statements for explicit description of parallelism and message passing.

Like SETL, our language has some facilities for supporting object-oriented programming as well as for defining new types of data and control.

It is also essential to note that PARIS constructs are oriented to express opposite and dialectical language properties: procedureness and declarativeness, typed and untyped data, explicit and implicit parallelism, dynamic and static objects, etc.

2. PARIS kernel

A part of SETL means is used as basic types and constructs of PARIS. The following reasons promote this selection:

- SETL types "set" and "tuple" have rich expressive facilities for composite object representation and realization of the powerful mathematical operations upon them;
- SETL constructions set former and tuple former allow in a way to say "what to do" without a more precise definition "how to do";
- most operations with basic types can be efficiently realized by associative or quasi-associative processors and some other parallel

machines.

2.1. Basic type features

In the language PARIS there is no special operator to declare the basic types, and they are recognized by the interpreter. Let us consider some particularities of set and tuple operations.

A set or a tuple consists of elements, and each of them belongs either to a basic type or to a defined type. An item name can be a set element or a tuple element if a value has been assigned to it. When performing set operations, each element, being an item name, is changed by its corresponding value. Set and tuple operations manipulate with values. But there are tuple operations allowing to manipulate with item names.

It should be noted that the set operations are rather efficiently realized by machines of both SIMD-architecture and MIMD-architecture.

2.2 Operations with basic data types

Basic types consist of standard (*integer*, *boolean*, *atom*) and compound (*character_string*, *bit_string*, *set*, *tuple*) types. Basic type operations except *atom* type are well-known. Operations with atoms are defined as in ¹⁰. It should be only noticed that the atom value may be uncertain (that is referred to *om*) and it can be an element of the set or tuple. Operations with compound data types are divided into two groups: formers and transformers. Formers construct new objects without changing source objects, while transformers change them.

Operations with strings. Let *S* and *S'* be character strings, *i* and *j* be integers. Consider the essential formers for the character strings:

S+S' is a concatenation of strings *S* and *S'*;

*i*S* is a concatenation of *i* copies of *S*;

S(i) yields the *i*-th symbol of *S*;

S(i..j) yields a substring which consists of the *i*-th through the *j*-th symbols, inclusive, of *S* ($1 \leq i < j$);

#S yields the length of the string *S*.

Let α and β be bit strings of the same length. In addition to the mentioned above, there are the following formers which are defined in

the obvious way: $\alpha \vee \beta$ is a disjunction, $\alpha \wedge \beta$ is a conjunction and $\neg \alpha$ is a negation. Other logical operations are introduced from these operations

by means of superposition.

A character (or bit) string transformation is performed by application of the assignment statement of the form: $S(i) := S'$; $S(i..j) := S'$. In these cases the string S' is inserted into the string S instead of $S(i)$ or $S(i..j)$, respectively.

Tuple operations. Let T and T' be tuples, i and j be integers. Consider the following main tuple formers:

$T + T'$ is a concatenation of tuples T and T' ;

$i * T$ is a concatenation of i copies of T ;

$\#T$ yields the index of the last element of T which differs from *om*;

$T(i)$ yields the i -th element of T ($1 \leq i \leq \#T$);

$T(i..j)$ yields the i -th through the j -th elements, inclusively ($1 \leq i < j \leq \#T$).

Let x and y be variables. Consider the following tuple transformers.

T *with* x adds the value of x to T as its last attached element;

x *app* T adds the value of x to T as its first element which attaches left;

y *fromb* T removes the first element from T and assigns its value to variable y ($y := T(1)$);

y *frome* T removes the last element $\neq om$ from T and assigns its value to variable y ($y := T(\#T)$);

Let z be an object name which should be attached to the end or to the beginning of the tuple T . Then transformers T *with name*(x) or *name* (x) *app* T , respectively, are used.

The change of tuple elements can be performed with the help of assignment statement which is defined as it is for the strings: $T(i) := T'$; $T(i..j) := T'$.

A tuple can be defined either by enumerating its elements or with the help of the former or transformer. The transformer of tuple elements is the following:

$$* < H(x) : x \text{ in } T \mid P(x) > * , \quad (1)$$

where $H(x)$ is an expression which differs from the identical function, T is a tuple or its name ("the source" of elements), $P(x)$ is a predicate. Let K be the name of the new tuple which is constructed by the transformer from (1) and x be the i -th element of the initial tuple T , where $1 \leq i \leq \#T$. Then if the value of $P(x)$ is *true*, then $K(i) = H(x)$ otherwise $K(i) = x$.

Note that this construction is applicable when only some of the elements should be changed.

The tuple former is the following:

$$\langle H(x) : x \text{ in } \tau \mid P(x) \rangle, \quad (2)$$

where $H(x)$ is an expression, τ is a tuple or a set or their names, $P(x)$ is a predicate. Let K be the name of the new tuple constructed by (2). The tuple K consists of all values $H(x)$ formed for those x in τ for which $P(x)$ has the value *true*.

Operations with sets. Let S and S' be sets. The following set operations are introduced:

$S + S'$ is a union of sets S and S' ;

$S - S'$ is a disjunction;

$S * S'$ is a intersection;

$S \text{ mod } S'$ is the symmetric difference of S and S' which is equivalent to $S + S' - (S * S')$.

The operation $\#S$ yields a number of elements of the set S .

Let y be a variable. In addition, two following transformer operations are used:

$S \text{ with } y$ adds the value of y to S if this value is not an element of S ;

$S \text{ less } y$ removes the value of y from S if y is an element of S .

In order to form sets we use the following construct

$$\{ H(x) : x \text{ in } \tau \mid P(x) \}, \quad (3)$$

where $H(x)$ is an expression, τ is a set or a tuple, or their name, $P(x)$ is a predicate.

Like (1) the set transformer is written as

$$* \{ H(x) : x \text{ in } S \mid P(x) \} * , \quad (4)$$

where S is a set, $H(x)$ and $P(x)$ are defined as above. Set constructs (3) and (4) are defined in the same way as they are for tuples.

It is important to note that in expressions (1-4) $H(x)$ and $P(x)$ may be complex structures and use recursively constructs like (1-4). In addition, in a sense the expressions are the production constructs. So, these language means give us an opportunity to support logical and functional programming.

2.3. Predicates and quantifiers

Let C and C' be variables of compound types. For these types we use the following predicates which are introduced in the conventional way:

$C=C'$, $C \neq C'$, $C \text{ in } C'$, $C \text{ notin } C'$, $C \text{ subset } C'$.

In addition, if C is a set name or a tuple name, we introduce the predicates $\text{name}(C) \text{ in } C'$, $\text{name}(C) \text{ notin } C'$,

which describe the occurrence of name C into C' .

The following predicates are used for symbol strings C and C' to describe their lexicographic order: $C < C'$ (the string C comes earlier than the string C' in standard lexicographic order), $C \leq C'$, $C > C'$ (the string C comes later than the string C' in lexicographic order), $C \geq C'$.

Notice that quantifiers are introduced as in predicate calculus.

2.4. Control structures and some other facilities

The language PARIS uses conventional control structures: *if*, *while-do*, *for all* and some other statements. It should be only noted that these structures are oriented to the objects which characterize PARIS.

In addition, in PARIS there are such notions and respective constructs as program, module, function, procedure and statements for describing the constants and variables, as well as for communications with external environment.

For an explicit definition of parallel execution of some functions, procedures or modules, the statement $\text{execute}(x_1, \dots, x_n)$, is used where $x_1, \dots, x_n$ are the object names.

In order to start a module and to send a message to it we use the statement $\text{message passing}(x, y_1, \dots, y_n)$, where x is the module name, and $y_1, \dots, y_n$ are the sending parameters. The statement can be utilized only for posting the events. The posting module can use the

statement *wait*.

On the one hand, all these facilities allow us to introduce new data types and, on the other hand, to structure programs. To a certain extent they may be considered as kind of support to the object oriented programming.

3. PARIS LIBRARY

A library of standard functions and procedures is used to extend of language opportunities for different applications. The extension is based on various shell construction. First, items for extending the object identification are included. Primitive functions for comparison with the pattern from SNOBOL are used for symbol string processing. There also are the following functions: string reverse; replacement in the string X the left occurrence (or all occurrences) of the string Y by the string Z; deletion from the string X the left occurrence (or all occurrences) of the string Y.

Similar to SETL, a tuple element in PARIS kernel is identified only by its number. Therefore a group of standard functions are introduced to yield numbers of all occurrences (or of the left occurrence only) of the element value x into the tuple K. Also there is a function to get the total number of all occurrences of the value (or the name) of element x into K.

Thus, standard items for tuple processing are used for changing the source tuple K. Moreover, there are procedures for changing or suppressing all occurrences or only the left occurrence of the value (or name) of the element x in the tuple K. Besides there is a procedure to insert a new element in K after the i-th element.

This is certainly only the first group of items in our library.

4. Conclusion

It should be noted that the language PARIS has means for combining the two approaches. One approach considers objects as the

foundation. This is expressed by the facilities which support the object-oriented programming. The other approach considers relations as the foundation. This is expressed by the facilities which support the logical and functional programming.

The project PARIS is realized on the computer system "Siberia" including associative units and modules of parallelism of other types. The three main software components are: compiler, loader and interpreter. The design of the system takes into consideration language evolution, including new knowledge about different applications.

It is well-known that the interpreting system is often not efficient enough. In spite of that we expect an efficient realization for the following reasons:

- orientation to various hardware types of parallelism;
- use of flexible combinations of coarse-grained and fine-grained parallelism;
- possibility of evolution of the language base and representation of different structures and types by different internal forms;
- use of a set of virtual compatible machines within the framework of one computing system.

5. References

1. T.Kohonen, *Content-Addressable Memories*. (Springer-Verlag, Berlin, 1980).
2. Ya.I.Fet, *Parallel Processors in Control Systems*. (Energoizdat, Moscow, 1981).
3. C.C.Foster, *Content - Addressable Parallel Processors*. (Van Nostrand Reinhold Company, New York, 1976).
4. N.N.Mirenkov, in *Lecture Notes in Computer Science*, vol. 457 (Springer-Verlag, Berlin, 1990), p.806.
5. N.Wirth, *Programming in MODULA-2*. (Springer-Verlag, Berlin, 1985).
6. W.F.Clocksinn, C.S.Mellish, *Programming in Prolog*. (Springer Verlag, Berlin, 1984).

7. W.D.Maurer, *The Programmer's Introduction to LISP*. (American Elsevier inc., New York, 1972).
8. J.D.Ullman, *Principles of Database Systems*. (Computer Science Press, 1980).
9. L.Gilman, A.L.Rose, *APL: An Interactive Approach*. (John Wiley, New York , 1976).
10. J.T.Schwartz, R.B.K.Dewar, E.Dubinsky,E.Schonberg. *Programming with Sets. An Introduction to SETL*. (Springer-Verlag, New York, 1986).
11. D.Ya.Levin, *The Very-High Level Language SETL and its Realization (for BESM-6)*.(Nauka, Novosibirsk, 1983).

DISTRIBUTED-MEMORY PROGRAMS DEVELOPMENT
IN DECLARATIVE & PROCEDURAL ENVIRONMENT

SERGEI P. GORLATCH

Institute of Cybernetics, Kiev 252207, USSR

and

GALINA N. GORLATCH

Institute of Cybernetics, Kiev 252207, USSR

ABSTRACT

We describe an environment intended to simplify programs development for parallel distributed-memory machines. It is an attempt to integrate two techniques - declarative and procedural. Declarative specification embodies whatever parallelism has existed in the original algorithm and is realized automatically. Procedural tools aid the user to improve the performance, efficiency and reliability of target program. A distinguishing feature of the environment is user's flexibility in choosing the most appropriate combination of language interfaces and tools.

1. Introduction

Efficient software development is a critical bottleneck in the utilization of new parallel architectures today. Even subtle bug can make the behaviour of parallel program nondeterministic, that is, time- and configuration-dependent, thereby creating nightmare during debugging.

In addition to all problems of sequential programming, in parallel systems there are also problems concerning:

- partitioning tasks for parallel execution;
- debugging/preventing deadlocks and races;
- production of efficient solutions and performance prediction.

In this paper we describe an environment intended to simplify parallel programs development. Examples of systems motivated by this goal include Suprenum¹, PTOOL², MODEL³, TDS⁴. Our main objective is to enhance parallelism in solutions rather than to use only modular program design concepts^{5,6}. We try to integrate two different techniques - automatic program synthesis from declarative equational specification and stepwise refinement of executable procedural specification - within one environment. It allows the user to choose the most appropriate abstraction, or their combination for the program under development.

Section 2 gives environment overview; in sections 3 and 4 the principles underlying the individual tools of the environment are presented; section 5 discusses the use of the integrated environment. In conclusion we describe the current status of the environment.

2. The environment overview

Problems whose solutions are most naturally stated in declarative form can be presented to the environment in that form - without a detailed procedural specification. The mathematical nature of declarative language does not presuppose any extraneous dependencies in the implementation: declarative specification embodies whatever parallelism has existed in the original algorithm.

In our environment we stay within the limitations of recursively defined arrays rather than extending to general functions. The main reasons for that are the following:

- 1) the area of applications we have in mind is that of data processing and numerical methods usage; in this area, arrays play important role and the direct specification of relations between input and output data seems both natural and adequate;
- 2) imposing syntactical restrictions on the expressions that appear as

array subscripts enables us to perform a much deeper analysis of the specification and to hope for a compiling system rather than interpreting one.

Many applications may be most naturally specified in an explicitly parallel style. For example, the control of a real-time distributed process in a flexible manufacturing system is expressed by explicit control parallelism.

In the integrated environment a parallel procedural solution may be either conceived by the user or synthesized automatically from declarative specification. In both cases, semiautomatic debugging and analysis tools are employed in the environment to predict performance and to reproduce sequencing of asynchronous events. Thus, the implementation of hardware for debugging parallel control is avoided by utilizing the environment to simulate a virtual multiprocessor in software.

In the environment, from the user's point of view, a choice of language interfaces (declarative or/and procedural) provides the programmer with the ability to select the most appropriate or natural specification of a problem or a solution to a problem.

Current version of the environment is based on the so-called data-driven parallelism. A crucial point when implementing parallel program for distributed-memory multiprocessor is the distribution of data across the processor memories. Because processors may only have reference to data stored locally, it is vital that each processor does as much computation as possible on the data it stores. This requirement was called the macroconveyor principle⁷. To meet it, target program is constructed as a set of parallel components (logical analogues of processors) each of which operates mainly on its local data. If non-local data are needed they are received from other components by messages. A good data distribution allows the workload to be well balanced among processors while minimizing interprocessor communications.

3. Declarative specifications

Declarativeness does not encourage the programmer to describe the

microscopic behaviour of each individual processor. Rather, it is best described what is to be done by the processors collectively. Thus, it allows the user to focus on algorithmic issues leaving the error-prone aspect of managing parallelism to the compiler and run-time system.

The approach avoids machine-oriented semantics and consists only of representing an algorithm by variables (arrays) and composing rules in the form of equations that define the variables. Following⁸ the set of recursive equations is called an equational specification.

Let $x_1, \dots, x_m, y_1, \dots, y_l$ be arrays on integer lattice Z^n , i.e. partial functions from Z^n to data domain D . Adding to D symbol w for "undetermined" value, we will consider arrays and other functions to D as completely defined functions. We consider system of equations with variables $x_1, \dots, x_m$:

$$x_k(j) = F_k(x_1(g_{k1}(j)), \dots, y_l(h_{kl}(j)), \dots), \quad j \in J_k, \quad k=1, \dots, m. \quad (1)$$

Here $j=(j_1, \dots, j_n)$ are free indexes ranging over $J_k \subset Z^n$, F_k - functions over D , g_{k1}, h_{kl} - index functions. This system of equations defines (recursively) the values of the elements of arrays $x_1, \dots, x_m$. We are interested in such equations, because computational methods in many applications are represented by similar systems for concrete interpretation of symbols F, g, h . We call $x_1, \dots, x_m$ output arrays; arrays $y_1, \dots, y_l$ and other values from the right sides of equations are called input data.

System Eq.1 is considered as nonprocedural specification of the algorithm, which computes output arrays as solution of the system Eq.1 for given values of input data. There are two main problems for nonprocedural specification :

- 1) to analyze a specification for being consistent and complete, i.e. for being realizable by some program;
- 2) to find the method for transforming a specification into procedural form realizing the specification, i.e. computing such values of output

arrays, which turn all equations of the specification into equalities.

Some methods for synthesizing parallel program from such a specification are surveyed in⁹. Having in mind distributedmemory virtual machine we develop the method based on datadriven parallelism. Below we outline main steps of the method.

1. Choosing data partition: decides how the specification domain is divided among the processors. We define the partitioning function giving for each array and each processor, the set of array elements which the processor stores locally. The goal of partitioning is to minimize references to non-local elements. It seems unlikely that a compiler could choose an optimal variant automatically (see¹⁰). In the current environment the programmer is required to provide annotations describing the partition desired.

2. Components generation: gives the rough form of the target program as a set of parallel components executing their parts (blocks) of the domain. Obviously, there may be non-local references making this collection of components yet unsuitable for direct implementation.

3. Send/receive messages generation: using subscript analysis we determine which array elements have to be moved into/from the local memory of given processor. This means generating corresponding send/receive statements. The generation procedure must guarantee dead-lock prevention, otherwise debugging by simulation is used.

4. Procedural specifications

The procedural specification language is intended to meet two main requirements: 1) to maintain all basic means of distributedmemory parallelism; 2) to allow the user to construct an executable specification leaving some details of the program under development undetermined.

We use language for Parallel Control-Time Executable Specification (PCTES). Below we characterize this language.

Parallel: the specification is a collection of components working in parallel. Each component has the program loaded in it. An array of

components may have the same program which uses component coordinates in the array as parameters. There may be global parameters in the specification which may be used in declarations and statements of component programs. Components communicate by message-passing implemented as rendezvous, through virtual universal system of communication each-to-each. The components names are used in send-receive statements; the notions of channels and ports are not used in the language.

Control: the procedural (sequential) features of the language are fairly standard. There are assignments, loops, conditionals, gotos, etc. The behaviour and communication of components may be represented in the specification as precisely as the user can at the current level of the development process.

Time: is explicitly used in the specification in some units chosen by the user. Any program fragment may be specified simply as time delay, e.g., `COMPUTE_A(<time>)` or `READ_B(<time>)`. Expression `<time>` is of integer type and may include program parameters and variables.

Executable: specification can be investigated using simulation tools. The user has an illusion of parallel program execution and receives time-traces of all components. These traces include all actions executed by components as well as waitings during rendezvous. Analyzing them the user can easily find the source of wrong behaviour.

Specification: the fragments of program under development which either are not designed yet or may be hidden at the current level of development, are presented in the specification simply as time delays. These delays are used mostly for representing computational parts of the program (nested loops). Naturally, if an assignment has an impact on the program behaviour, e.g., computes the name of partner for communication then this assignment may be explicitly included into the specification.

So, the implementation of the PCTES language provides the user with information useful in refining and tuning parallel algorithms. The analysis provided has three uses:

to allow the user to debug interprocess communication and synchro-

nization aspects of asynchronous computations without requiring instrumentation of the target hardware;

to support analysis of efficiency of algorithms in order to permit the user to make intelligent modification to improve them;

to evaluate the performance of an algorithm in terms of the objective of the algorithm itself.

5. Integration of software tools

Integrated environment provides different tools at different levels of abstraction to automate the writing of parallel programs, to analyze and optimize their performance. For system prototyping, the user can test various strategies for choosing algorithms, controlling the sequence of algorithms to be executed and integrating from various algorithms. For algorithm prototyping, the user executes a particular algorithm using different data to examine the performance of the algorithm.

Declarative tools in the environment allow the user to ignore parallelism (via automatic parallelization) and use special algorithms to simplify software development. Procedural tools within the environment aid the user in improving the performance, efficiency and reliability of target programs.

A distinguishing feature of the environment is user's flexibility in choosing the most appropriate combination of development tools. This was used for wide range of applications. One of them - ocean climate modelling - was developed using quite a complex mixture of declarative and procedural means. At the first level, four parallel threads were determined in accordance with physical values being computed: temperature, salinity, and two velocity coordinates. General structure of threads and their communication were described by procedural specification and debugged using simulation. Then within each thread the corresponding grid method was specified declaratively. Choosing various variants of partition and comparing the resulting programs through simulation, data-driven parallelization was implemented. The performance

ce of each thread as a parallel program was estimated and appropriate processors number was found. This enabled us to refine the initial procedural specification of the whole program and to predict its global performance.

The environment provides a uniform interface between the user, the tools, and the program. An automatic consistency maintenance mechanism is being developed to insure the consistency among different versions of the same algorithms.

5. Conclusion

Program development for new parallel architectures must include both: highly capable automatic tools, and strategic decision-making by expert user.

The present experimental version of the environment is implemented as two systems on IBM PC under MS DOS: automatic program synthesis in the algebraic programming system APS-1 environment and the simulation system written in C.

7. References

1. H.Zima, in *Vector and Parallel Computing*, 1989, p.395-404.
2. D.Callahan, K.Kennedy, in *Journ.Supercomput.*, 2 (1988), p.151-169.
3. I.Lee, N.Prywes, B.Szymansky, in *Advances in Computers*, 25 (1986), p. 215-276.
4. *Transputer Development System*, Inmos Ltd., 1989.
5. R.Cook, R.Auletta, in *SIGPLAN'86 Symp.Lang.Env.*, 1986, p.102-110.
6. R.Sweet, in *SIGPLAN'85 Symp.Lang.Interf.Progr.Env.*, 1985, p.216-229.
7. V.Glushkov, Y.Kapitonova, A.Letichovsky, S.Gorlatch, in *Cybernetics*, 1981, N 4, p.13-21.
8. A.Pnueli, R.Zarhi, in *LNCS*, 1981, 115, p.459-478.
9. Y.Kapitonova, A.Letichovsky, S.Gorlatch, G.Gorlatch, in *Cybernetics*, 1989, N 1, p.22-35.
10. C.Koelbel, P.Mehrotra, J.Rosendale, in *Int.Journ.Paral.Proc.*, 16(5), 1987, 365-382.

RECOGNIZING FACES WITH A MASSIVELY PARALLEL SYSTEM

Paola Flocchini[°] – Francesco Gardin ^{°,°°} – Giancarlo Mauri[°]
 Maria Pia Pensini^{°°} – Paolo Stofella^{°°}

[°] *Dip. di Scienze dell'Informazione , Università di Milano, Via M. da Brescia 9
 20133 Milano, Italy*

^{°°} *Advanced Computing Systems , Via Rombon 4
 20133 Milano, Italy*

ABSTRACT

The purpose of this work is the development of a system able to recognize faces in different perspectives, introducing some kind of noise and having different expressions.

The problem of faces recognition has been approached using a complex architecture based on a hierarchy of neural networks and on a particular self-referencing structure. The system, in fact, is structured as a tree in which nodes correspond to neural networks each one having different tasks and each leaf is a recognition module composed by some networks with different characteristics depending on the different preprocessing operators used. These networks are coordinated by a supervisor in a self-referencing structure; the supervisor, called Meta-Net, in training phase observes the behaviour of recognition nets while in test phase it decides, given an input image, which weights to assign to each network and modifies their output in order to obtain the final result.

This architecture shows a high generalization capability and allows the recognition of images with different kind of noise.

1. Introduction

Pattern recognition, in particular visual pattern recognition, is a task easily accomplished by human beings, but it presents significant difficulties for computers: even a small child can recognize faces, whereas a supercomputer is stretched to its limits when performing such kind of task. In contrast, even a small personal computers can quickly perform a series of laborious calculations beyond most humans. This contrast between computational capabilities of humans and computers suggests that computation can be divided in two domains: symbolic processing, in which computers excel, and pattern processing, based on subsymbolic computation, most suited to human capabilities.

The ability of humans in pattern processing is originated in the structure of the human brain: a massively parallel natural computer composed of many billions of neurons, each one of which performs very simple computations, and is connected with 10,000 other neurons. Essentially, each neuron computes a weighted sum of its inputs, compares this sum with a threshold value, and outputs a signal if the threshold is exceeded.

Since the earlier days of automatic computation, the idea of realizing a machine with a structure, and hence capabilities, similar to that of human brain fascinated many researchers: a formal model of neuron was defined by MacCulloch and Pitts, and artificial networks of formal neurons were studied. However, both for technological difficulties in

constructing such kind of nets and for conceptual difficulties in understanding how a net could be trained to perform a given task, computer science and artificial intelligence took a different way, based on explicit programming and deterministic algorithms.

Recently, the evident impossibility of efficiently solving some classes of problems in a symbolic way, the technological opportunity of constructing computers with some thousands of processors, and some new ideas and techniques on training, programming and using massively parallel computer systems¹ renewed the interest for neural networks.

In this paper we will describe a system based on the neural paradigm for the recognition of faces. In the next section a short description of the neural paradigm in computing and of its applications in the domain of pattern matching is given. Then, Sec.3 describes the goals and the general architecture of the system, and discusses the concept of meta-neural net, which is introduced for enhancing flexibility and efficiency of the system. In the next sections, more details on the system and some experimental results are given.

2. The Neural Paradigm

A neural network is a system of highly interconnected elementary processors which operate at discrete time steps through mutual interaction. At any step each unit is at a certain level of activity, which is determined by the levels of activity of the units which transmit signals to it, weighted by the strengths of the interconnections. In general terms, the task of the network is to learn a mapping from inputs to outputs. Learning the required mapping involves gradually changing the weights of the interconnections, through an iterative procedure involving error correction. Each input is presented in turn, the resulting output is compared with the desired output and the weights are modified in order to make the correct output more likely to occur on subsequent presentations of this input.

Neural networks are specified by node characteristics, in particular response function, net topology and training rules. The iteration mode, i.e. the updating rule of the neurons, completely specifies the dynamics of the net as a finite discrete iterative system.

As for nodes, the simpler case is that of threshold neurons, for which the response function is the step function from 0 to 1. In this case, the neuron performs a weighted sum of its inputs, compares this to some internal threshold level, and turns on only if this level is exceeded. If not, it stays off. In mathematical terms, if the neuron has n inputs with associated weights w_i ($i=1, \dots, n$), and a threshold θ , then it computes the function:

$$f(x_1, \dots, x_n) = \begin{cases} 1 & \text{if } \sum_{j=1}^n w_j x_j \geq \theta \\ 0 & \text{otherwise.} \end{cases}$$

where x_i is the input signal (0 or 1) on the input i .

Connecting these neurons together may well produce networks that can do something, but we need to be able to train them in order for them to do anything useful. The network topology has an important role in the possibility of training the network, and varies significantly between neural network models. The largest variation is in the training or *learning rules*. These rules specify an initial set of weights and indicate how the weights should be modified in order to improve performance. The simpler case is that of the

perceptron, firstly described by Rosenblatt², in which a single layer of units is used in order to classify input patterns in two or more classes. Perceptrons have a very simple learning rule, for which convergence to a stable state is guaranteed³, but they can "learn" to compute only linearly separable boolean functions, and hence are unable to compute for example such an important function as the "exclusive or". Due to this strong limit in their computational power, perceptrons were considered practically useless for about 20 years.

In 1986, Rumelhart, Hinton and Williams⁴ showed that a slightly more complex network, the multi-layer perceptron, or multi-layer feed forward network, in which the connections are just between layers of units and in which the information flows in one direction, can be trained with the so-called *error backpropagation* algorithm, so as to compute a larger class of functions. This algorithm signalled the renaissance of the whole subject, and was applied with success to many different fields.

Neural networks were applied to many problems of machine vision and object recognition, for example using backpropagation. However, this is not the only possibility; other kind of approaches are based for example on unsupervised learning and associative memories^{5,6}. One of the fields in which neural networks are applied is speech processing, recognition and synthesis. Other widespread applications concern character recognition, for example for signature verification. Human experts obtain a 50–60% accuracy; neural networks trained with about 100 examples of the signature can achieve an accuracy of between 92 and 98%.

In the following, we will describe an application of neural networks to the recognition of faces, in which an architecture based on a hierarchy of neural networks is experimented in order to improve performances.

3. The Face Recognition System: Objectives and Architecture

The system conceived for the solution of the face recognition problem doesn't represent only an interesting application of neural nets, but also an opportunity to test the limits of this technology and to extend the notion of neural net while attempting to overcome these limits. The most relevant notion generated in this respect is the concept of meta-neural net, that is the use of neural nets applied to neural nets.

The typical use of neural nets for face recognition⁵ has been made through the application of standard filtering techniques to the raw digital data of the picture representing the face. Although sophisticated filtering techniques can contribute dramatically to improve the recognition rate, they cannot deal with major changes in the original information. For example a profile is too different from the front of a face and we cannot expect that a neural net can synthesize a function with some meaning if the input data are completely different. The same argument can be applied to expressions of the face, scaling, lightening, etc. On the other hand, since recognition by humans takes place under all the above noise conditions, the problem has proved to be empirically solvable by an appropriate physical system.

In order to address the face recognition problem through a neural net, where the efforts to increase the recognition rate above 90% becomes extremely hard, we have taken a different approach, akin to similar problem solving strategies in Artificial Intelligence. Our

approach is based on the notion of *meta-neural network*, that is a neural net whose associated classification function correlates face's raw data to the neural net more likely to succeed in its recognition. In other words the knowledge of the envisaged neural net is about the performance of other neural nets. In Fig. 1 a schematic description of the proposed architecture is shown. At the bottom level, we have a series of filtering techniques applied to the face raw data; a number of neural nets have been trained individually on face recognition of data processed by specific filtering techniques. At the top level the meta-neural net has been trained on the recognition performance of each net to the same unfiltered face raw data fed into the meta-neural net. The output of this net is represented by the neural net which is believed to be the most appropriate for the recognition task.

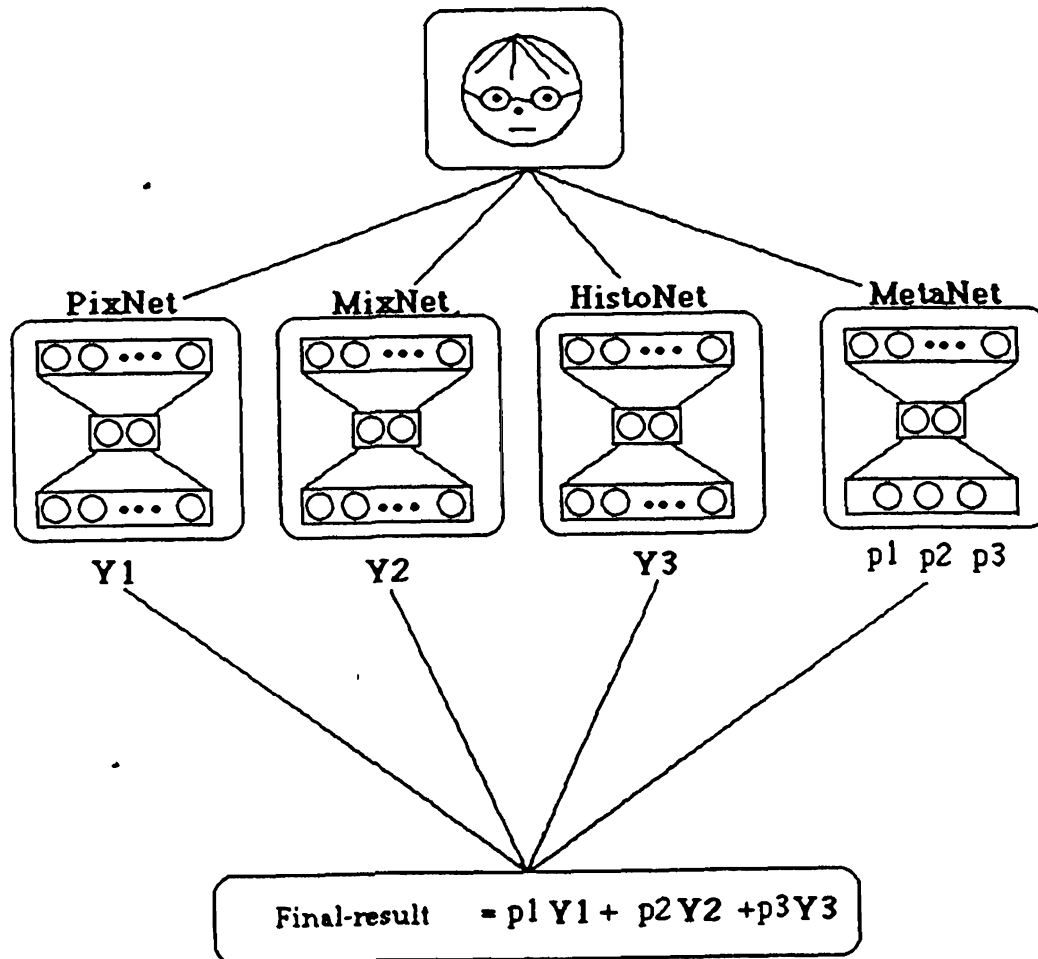


Fig. 1 – System Architecture

By using this architecture we have attempted to overcome the intrinsic limitation represented by the ability of a single neural net to generalize effectively on a data set containing data belonging to a same class, which are too heterogeneous. For example the same face with different expression, scaling, roto-translation, lighting, etc. Combinations of such differences are also dealt with.

While the rest of the paper will deal with the specific aspects of the above architecture, in this section we would like to further investigate the notion of meta-neural net. We can

define the meta-neural net as the classification system which associates faces to filters (and the corresponding net). Similarly we can add data sets which deal with profiles, and any other kind of drastically different view of the face, and repeat the training process with new individual neural nets and the associated filters.

The use of a meta-analogical net is therefore a new integration technique which allows to add incrementally new filters, data sets and neural net groups. This is possible since the knowledge embedded in such a net is about the individual neural nets rather than the specific recognition aspects of the face per se.

The results obtained so far are very encouraging, although they can be claimed only on a purely experimental evidence, since a mathematical model of the combination of neural and meta-neural nets has not been defined; nevertheless they are better than any other recognition rate presented in the literature under the same experimental conditions in the variety, noise of the input data and number of faces to be recognized.

4. The Need for Image Preprocessing Operators

The general objectives of this work include the ability to deal with different kind of noise, such as light, position, expression of face, orientation of face etc. Only some of these aspects have been treated at this stage of the work. In particular, the first experimental tests were devoted to the research of effective solutions to problems of translations, rotations and change of face expression (laugh, smile, cry ...).

In the researches^{6,7,8,9} on neural face recognition, usually a backpropagation network has been used with the full set of image pixels as input pattern. This approach gives good results only for images with fixed light, position and face expression; introducing one of the previous alterations in the input patterns causes a dramatic decrease in the network recognition capabilities.

The above experimental observations suggest that there are strong limitations in the learning capabilities of backpropagation networks when simple pixel input patterns are presented. A solution to this problem has been investigated, by introducing and combining different preprocessing operators to increase independency properties in the input pattern. Different operators and a neural based feature extractor have been used which are able to produce patterns that are independent with respect to geometrical transformations such as translations and rotations and to face expressions.

5. Preprocessing Operators

The network ability to characterize different faces results dramatically improved when part of the analysis is performed using image processing operators; in this section some of these operators are described. It is important to note that we did not perform an exhaustive analysis of the known image processing algorithms; in fact, the main objective of the research has been to test the suitability of different approaches for the combination of preprocessing operators and neural networks. In the future engineering phase of the project, a more accurate analysis of preprocessing techniques will be performed. Histograms, ADG (see below) and a neural feature extractor were used in the experiments.

5.1 Histogram

This is the simplest way to study and analyze an image; it contains frequencies of each grey level intensity giving a global information about the image. Histogram doesn't maintain spatial relations between pixels, but it has the interesting property of invariance to position, scale and orientation. This characteristic makes it a suitable tool when geometric transformations in the plane are foreseen.

5.2 Averaged Squared Distance from Homogeneous Area Centre of Gravity (ADG)

This is another way to extract information from an image characterized by invariance properties. After having calculated the coordinates (x_m, y_m) of the centre of gravity of points belonging to each grey level, it has been made the sum of euclidean distances from each of these point to it averaging this value with the frequency of the considered grey level. To obtain also invariance to scale this value has been divided by image size. Each component of ADG vector has been obtained in this way:

$$\left\{ \sum_{i=1}^n [(x_i - x_m)^2 + (y_i - y_m)^2] / n \right\} / M$$

where:

- n is the number of pixel belonging to the considered grey level.
- M is the number of image pixels

The resulting vector represents spatial pixel disposition for each intensity of grey and it gives a value which indicates relative positions of these points.

5.3 Neural Based Feature Extractor

The third preprocessing operator considered in our study is a neural network accomplishing the double task of image compression and feature extraction. Following Cottrell and Fleming's suggestion¹⁰, we used a neural network to obtain from the original image global "semantic" features in unsupervised fashion. This method permits to obtain a good image compression in the same vector representation of the extracted features. This solution is particularly suitable for our purposes, due to the need of small input vector size in order to reduce the network dimensions.

A three layer backpropagation network has been used. The original image of 128x128 pixels is scanned by a "patch" array of 8x8 values that constitutes the input vector; the hidden layer consists of 4 neurons, the size of the output layer is the same as the input layer. The network is aimed at learning the same input vector, so that the hidden layer, having less neurons than the input and output layers, can be considered the compressed representation of the image. The hidden layer is in its turn the input layer of another network, called HistoNet.

In compressing the original image, the network automatically extracts its relevant features; for this reason we can say that the training is supervised with respect to the compression, but it is unsupervised with respect to the feature extraction. The training has

been done on 24 images, and the test on 156 images, resulting in a success percentage of 94%. The compression ratio is 16 : 1.

We noticed that the unsupervised training in feature extraction has been fundamental in giving the system the knowledge about "global" characters in human faces; further work will be done in order to better understand this behaviour.

Different networks have been built that use the above described preprocessing operators. The experiments revealed that the recognition performance of networks whose input patterns were built by applying the above operators reflects the operator's invariance properties: good performances were obtained using histograms, ADG and the neural feature extractor. These results were not yet satisfactory when multiple transformations were present in the set of images: a way to combine multiple operators was needed in order to further improve the network performance.

6. Complex Input Structure Can Increase Performance

Using an input pattern with complex structure resulting from a combination of different kind of pre-processing operations has been demonstrated a suitable way to solve the problem of robust recognition with images presenting multiple noise characteristics. The input pattern can be built in such a way that each part maintains the invariance properties with respect to a specific set of image transformations: the neural network has the ability to process these parts of the pattern in such a way that the overall recognition capability results dramatically improved.

Introducing the histogram information in the input pattern produced a good recognition performance for translated images: it results in a robust behaviour of the network under image translation operations and scaling. We faced more complex problems trying to obtain better performance from the same network with rotated images as input patterns. Even training the network with a huge set of rotated images lead us to unsatisfactory results. The use of the ADG method introduces a component in the input vector that is independent of rotation and scaling transformations .

However, we had to proceed in different ways for input representation of global ("semantic") features of the images. In fact, the simple attempt of using classical image processing techniques to extract this kind of features resulted in a too complex pre-processing phase. Previous works employing recurrent networks⁸ didn't seem satisfactory. So, we considered the suggestion given in¹¹, but in a different way: employing the compression ability of the network to obtain a good neural representation of global characteristic; that consideration suggested us to study an auxiliary network whose hidden neurons could be a part of the complex input of another network.

This work demonstrated that features that are not pre-specified may be extracted automatically for the recognition process. The obtained data have been inserted as components of input patterns of a recognition network. In Sec. 9 the different networks are described; in Sec. 10 the experimental results of this work are described, on the basis of the data set described in the next section.

7. Data Set

The data set construction criteria reflect the actual objectives in this phase of the project: faces were acquired introducing different face expressions, head rotations, image translations and rotations. A set of 60 pictures representing 12 persons in 5 different expressions have been acquired by a camera obtaining 256 grey levels images with 512x512 pixel resolution. All these images have been treated with appropriate image processing algorithms with the purpose of deleting noisy elements and evidentiating contour. The aim of this preprocessing phase was to obtain images with the highest informative content deleting noise without losing significant details for recognition.

In this phase median and high-pass filter have been used and some operators for rearrangement of grey levels intensities have been applied to images on the base of information derived from their histograms. Finally the images have been reduced to 64 x 64 resolution.

8. Networks Description

Three networks were built with different input structures, derived from the combination of preprocessing algorithms. Their structure is described in the following:

PixNet consists of:

- 4096 input units
- 100 hidden units
- 12 output units

This network uses the simplest way of representing an image; in fact its input data consists of a vector of 4096 components corresponding to 64 x 64 grey levels intensities of the pictures. The number of hidden units has been chosen after experimentations and tests, while 12 output correspond to 12 faces to recognize.

Characteristics of this network are related to the choice of this kind of representation; in fact, the network obtained by using pixel intensities as input data is robust respect to changes in expression but is totally unreliable in recognizing images with different orientations and scale. Such an image-vector does not have the property of invariance, a change in face position led to a significant change in input data and recognition does not succeed.

HistoNet has been implemented trying to solve this limit. It consists of:

- 512 input units
- 20 hidden units
- 12 output units

The 512 values correspond to 256 components of the histogram and 256 of ADG; the number of hidden units has been determined by experimentations. Using this representation input data are completely invariant to position, orientation and scale, in fact information extracted from the images depends only on the distribution of grey levels

intensities. As a consequence, the network shows a good behavior on rotated and translated images.

This net is also strong with respect to changes in expression of the face and to presence of noise, but to increase these characteristics another network has been created adding a vector of 1024 components to the one previously used with the purpose of improving net generalization capability.

MixNet has:

- 1536 input units
- 50 hidden units
- 12 output units

The input vector is composed by the 512 values described before and 1024 values obtained as compression of the image. This component represents a set of features extracted in an unsupervised way using a neural network for image compression.

In this situation input data consist of heterogeneous components with different meanings. Our main goal is to maintain the information about pixel disposition, geometric features and grey level distribution together in a vector and to use it as the image code with the purpose of obtaining a network robust both to changes in expression, because of the component of image compression, and to modifications in position thanks to the components of Histogram and ADG.

The integration of these three networks with the coordination of a meta-net supervisor permits to take advantage of their different characteristic in one system and, in this way, to obtain a better result than using them separately.

Meta-net

The main goal of this network is to coordinate the use of the three recognition nets. It is a meta-net which is trained by observing the behaviour of other nets on the same training images and generalizing results. The result of this structure is a module in which, given as input a face to recognize, the system automatically decides which weights should be given to the different networks on the basis of the characteristics of the observed image. This architecture gives the possibility of creating a system which is able, alone, to decide which kind of preprocessing operator has to be used in order to improve recognition.

9. Experimental Results Combining Different Techniques

Tests of the above nets have been carried out with the purpose of comparing the recognition percentages of the single nets with the ones of the system built as an integration of them. Our main goal is to show the validity of this architecture in performance terms. Two kind of tests have been performed using:

1. 24 images of T.S. disturbed by different percentages of casual noise (10%, 20%, 25%, 33%) for a total of 96 images;
2. 36 "new" images in standard frontal position and the 24+36 images corresponding to T.S. and "new", rotated of 90°, 180° obtaining 156 images.

The first test has been carried out on the 96 images with the purpose of proving the strength of the system at the introduction of gaussian noise.

The second test is meant to verify how strong are the nets are when generalizing their results on images not belonging to the Training Set.

First Test

Results of this test are summarized in the following table and in Fig.2.

Noise:	10 %	20 %	25 %	33 %
PixNet	100%	100%	92%	80%
HistoNet	100%	100%	95%	92%
MixNet	100%	100%	100%	94%
System	100 %	100 %	100 %	100 %

We can see that PixNet is the less robust to image recognition in noisy conditions, while using different kinds of representations for input data, with second and third net, the performance increases showing a better robustness to noise. Comparing performances of single nets with the ones of the global system we can also observe that, using this architecture, we've obtained a robust system in which the percentages of recognition on training set are not affected by the introduction of 33% of noise.

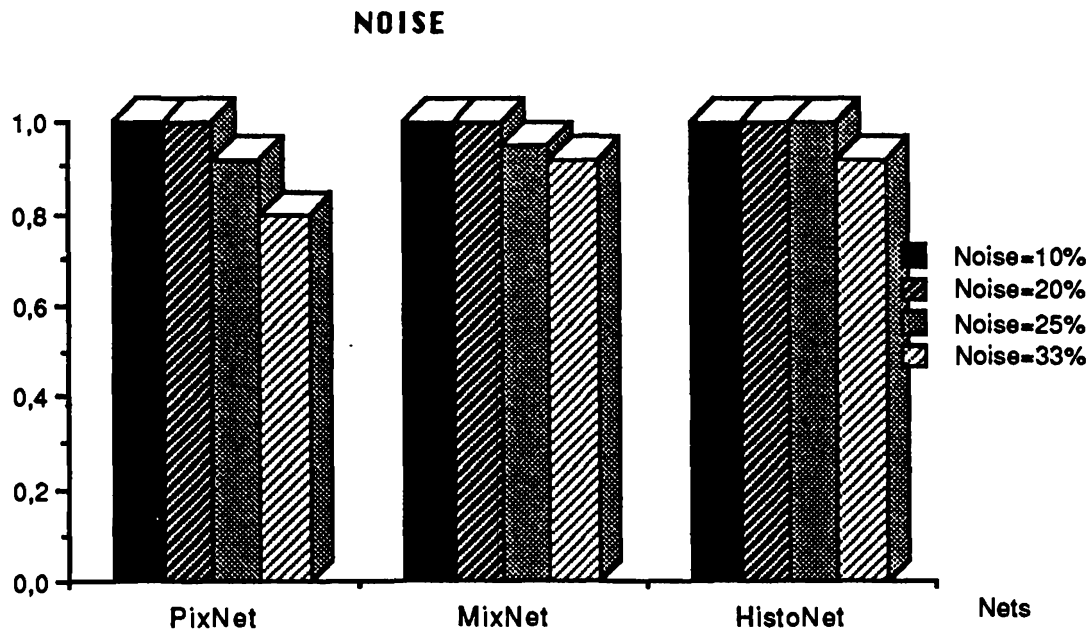


Fig.2 – The behaviour of networks on images disturbed by gaussian noise

Second Test.

Purpose of this test is to verify the generalization capability of the three neural networks and of the system on different images in standard frontal position and on rotated images in the plane.

In the first situation (Fig. 3) we can observe how performance increases using different techniques of input data representations. From this tests it is clear that the

introduction of meta-net has significantly improved the system performance, giving 100% of recognition on the set of "new" images.

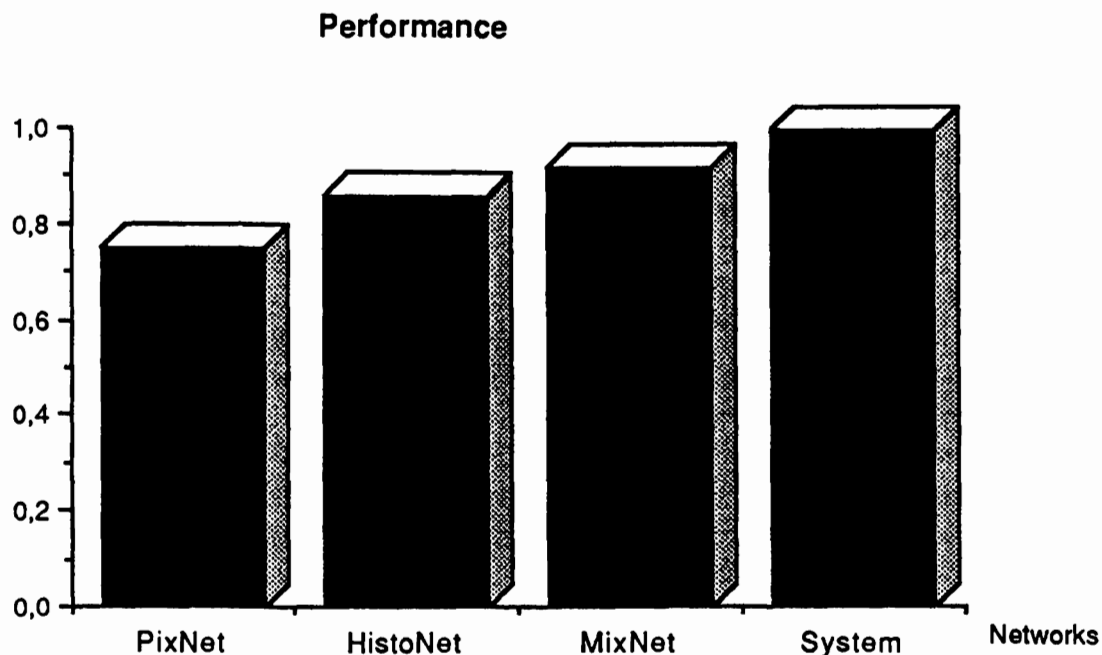


Fig. 3 – The networks performance on images in standard frontal position

In the second situation (Fig. 4) we can see that only HistoNet is able to recognize rotated images because of its input data representation.

When a rotated image is presented to the system, the meta-net addresses it only to HistoNet and this network replies correctly with a percentage of 92%.

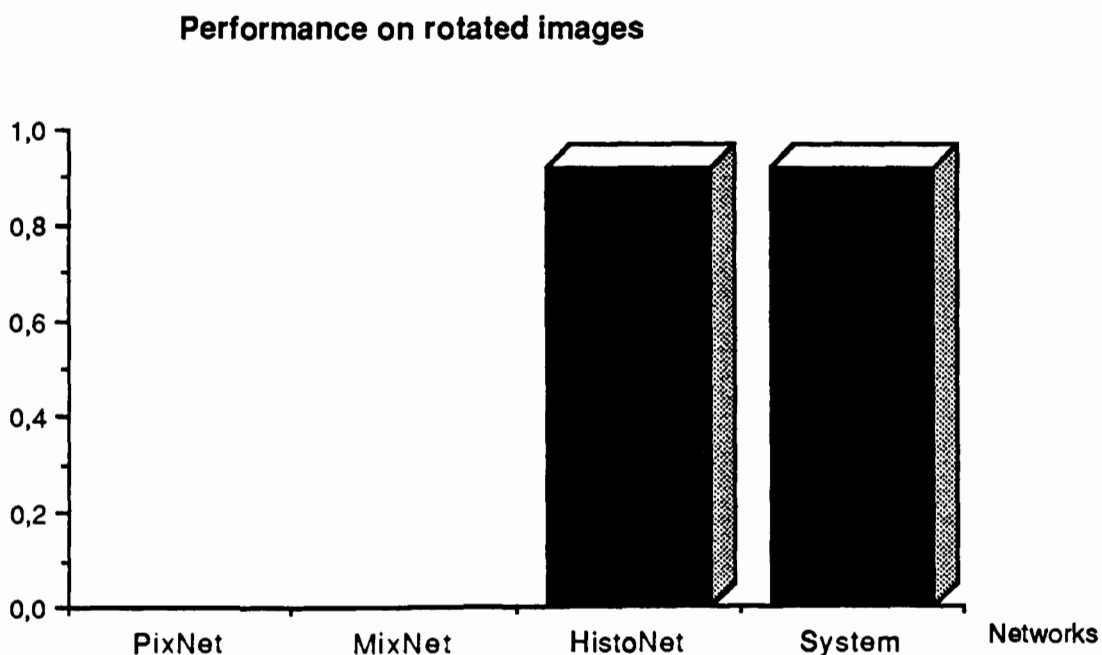


Fig. 4 – The networks performances on rotated images.

10. Acknowledgements

This paper has been supported by a grant in the frame of Progetto Finalizzato Calcolo Parallelo of the Italian National Research Council.

11. References

1. D.E. Rumelhart and J.L. McClelland, Eds, *Parallel Distributed Processing: Exploration in the Microstructure of Cognition* (MIT Press, Cambridge, MA, 1986).
2. F. Rosenblatt, *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms* (Spartan, New York, 1962).
3. M.A. Arbib, *Brains, Machines, and Mathematics* (Springer-Verlag, Berlin, 1987).
4. D.E. Rumelhart, G.E. Hinton and R.J. Williams, *Learning internal representations by error propagation*, in ¹.
5. T. Kohonen, P. Lehtio and E. Oja, *Storage and processing of information in distributed associative memory systems*, in G.E. Hinton & J.A. Anderson (Eds.), *Parallel models of associative memory* (Erlbaum, Hillsdale, NJ, 1981) 251–269.
6. T. Kohonen, *Self organizing and associative memories* (Springer-Verlag, Berlin, 1984).
7. A. O'Toole, R.B. Millward and J.A. Anderson, *A physical system approach to recognition memory for spatially transformed faces*, *Neural network*, 1 (1988), 179–199.
8. S. Rousset, A.C. Schreiber and S. Wang, *Modélisation et simulation connexioniste de l'identification des visages en contexte: le système Facenet*, RR 742-M-IMAG Grenoble, 1988.
9. K. Von der Malsburg, *Pattern recognition by labeled graph matching*, *Neural Network*, 1 (1988), 141–148.
10. G.W. Cottrell and M. Fleming, *Face recognition using unsupervised feature extraction*, (Proc. of INNC, Paris, 1988), 322–335.
11. G.W. Cottrell, P. Munro and D. Zipser, *Learning internal representations of gray scale images: an example of extensional programming*, in Proc. Ninth Annual Cognitive Science Society Conference, Seattle, WA, 1987.

THE USE OF A PRAM ARCHITECTURE FOR PARALLEL IMAGE RECONSTRUCTION

Wiesław Lucjan NOWINSKI

*Institute of Computer Science, Polish Academy of Sciences
P.O. Box 22, PKiN, 00-901 Warsaw, Poland*

ABSTRACT

The paper discusses problems concerning the use of a PRAM architecture for parallel image reconstruction. A short introduction to parallel image reconstruction has been given. SIMD and MIMD architectures for parallel image reconstruction have been reviewed taking into account the forms of parallelism exploited for parallelization as well as the ways of ensuring a conflict free access to the projection memory and image memory. Next a PRAM architecture has been presented and its properties described. Some examples of implementation of parallel image reconstruction algorithms on a PRAM architecture have been discussed considering, especially, the distribution of data among memories. It has been proposed to consider the PRAM to be the image reconstruction architecture.

1. Introduction

Image reconstruction from projections allows to visualize the internal structure of an investigated object [3,5,6]. It is used in many fields, such as radiology, radio astronomy, electron microscopy, industrial applications, among others. The reconstruction problem consists in evaluation of the 2-D or 3-D distribution of a physical property of an object from its projections taken through different directions.

To parallelize image reconstruction algorithms, forms of parallelism specific to image reconstruction have been introduced [7]. A form of parallelism determines a way of decomposing a given process into component parallel processes. The forms include:

- *pixel parallelism* (i.e., pixels are processed concurrently);
- *projection parallelism* (i.e., projections are treated in parallel);
- *ray parallelism* (i.e., rays are processed simultaneously);
- *operation parallelism* (refers to concurrency in operations);
- *plane parallelism* (i.e., cross sections of a 3-D scene are reconstructed in parallel).

Pixel, projection, ray, and plane parallelisms result from the data structure regularities, and each of them has been defined as an array of similar processes executed in parallel. Operation parallelism, on the other hand, is very heterogeneous and has

been defined as a list of concurrent processes. By using those forms, many parallel algorithms for transform and series expansion methods of image reconstruction have been formulated, as well as properties of parallel image reconstruction have been discussed, see e.g. [7 – 10].

In general, parallel image reconstruction is required in two types of systems: real-time systems and systems using scanners generating a huge amount of measurement data. Real-time systems are needed in industrial applications, e.g., to monitor on-line continuous processes. Huge data sets are generated by scanners used, e.g., in diagnostic medicine, like the Dynamic Spatial Reconstructor (DSR) [12]. The DSR is a multiple x-ray tube volume scanner allowing to generate fast scans of the beating heart, lungs, and circulation. It is designed to collect the measurement data sufficient to reconstruct up to 60 complete volumetric images per second, where each volume consists of up to 240 adjacent cross sections. Experimental protocols for determination of the presence and extent of coronary artery disease demand the analysis of up to 4 sec (i.e., four heartbeats) of 3-D data, i.e., up to 57600 cross sections. Computation of this number of reconstructions in a reasonable time requires parallel processing.

Parallel image reconstruction demands adequate architectures. The existence of numerous architectures for image and signal processing makes it attractive to adapt them to image reconstruction. For example, the Fourier transform (being used in transform methods of image reconstruction) may be implemented on SIMD, MIMD, or systolic architectures. However, many differences which exist between image reconstruction and image processing demand in developing architectures specific to image reconstruction.

For parallel image reconstruction, both SIMD and MIMD architectures have been proposed. SIMD architectures exploit synchronous parallelism, while the use of MIMD architectures results from asynchronous parallelism and an access to the shared (global) memory. An architecture combining both features, this is the *Parallel Random Access Machine* (PRAM). Therefore, the aim of the paper is to show that the PRAM is an architecture adequate for parallel image reconstruction. Moreover, the previous solutions for SIMD and MIMD architectures have been extended for a PRAM architecture.

2. SIMD and MIMD architectures for parallel image reconstruction

For parallel image reconstruction, both SIMD and MIMD architectures have been proposed. We will review some of them taking into account the forms of parallelism exploited for parallelization as well as the ways of ensuring a conflict free access to the projection memory (keeping a measurement vector) and image memory (holding a reconstructed image).

Let us consider SIMD architectures for the parallel convolution method. In [7] two synchronous algorithms making use of projection and pixel parallelism have been designed and architectures efficient for them proposed. A conflict free

access to the memory has been achieved by storing projections in separate memory modules, and by restricting the degree of pixel parallelism. The architecture exploiting projection parallelism stores a reconstructed image in the global memory, whereas a measurement vector is distributed among local memories. On the other hand, the architecture making use of pixel parallelism holds in the global memory a measurement vector, while an image is distributed among local memories. The global memory blocks are accessed by a switching network.

Another approach of parallelization of the fan beam convolution method has been presented in [2]. The fan beam convolution method contains two operations: convolution (filtering) and weighted backprojection, both involving the evaluation of inner products. Accordingly, the implementation has been focused on the realization of inner product computers [13]. In this solution, weighted backprojection is performed concurrently exploiting operation parallelism, but pixels of a reconstructed image are processed one after another. This way a simultaneous access to the image memory has been avoided.

Let us review MIMD architectures. A pipelined multiprocessing system for the convolution method has been presented in [1]. It processes projections sequentially, however, every projection is processed concurrently exploiting operation parallelism (three-stage pipelining). The low degree of parallelism allows the system to be used in situations where the reconstruction time constraints are not very severe. In [4] a MIMD architecture destined for industrial applications has been proposed. It is a multiple source, multiple detector system with transputers as processing elements. For reconstruction iterated discrete backprojection has been used (i.e., discrete backprojection which is repeated to obtain an overall error sufficiently small). Discrete backprojection is parallelized by pixel parallelism, i.e., each transputer is assigned to a pixel. However, component processes are not performed synchronously, since data flow and processing are along rays as for ray parallelism. It results in the intensive interprocess communication, as processes assigned to pixels have to transfer data along rays. The solution ensures a conflict free access to the memories, as the image memory is distributed among local memories of transputers, and the measured ray sums are sent directly along rays. However, the main drawback of this solution is a low and weakly balanced workload among transputers, as their utilization varies from 10% to 35%. To avoid these disadvantages, for the parallel implementation of series expansion methods a processor farm model has been proposed [8,11]. The processor farm provides an automatic mechanism for dynamic load balancing and allows a high efficiency to be obtained. In addition, the processor farm may be used for algorithms parallelized by various forms of parallelism, like ray, projection, or operation parallelisms.

3. PRAM architecture for parallel image reconstruction

The forms of parallelism are used to parallelize image reconstruction algorithms. In addition, the forms allow to study the properties of parallel image

reconstruction independently of peculiarities of a given algorithm. Therefore, we may expect the properties of these forms facilitate to determine a general architecture for parallel image reconstruction.

Component processes of each of pixel, projection, ray, and plane parallelisms form an array of similar processes. Moreover, component processes of operation parallelism are in many cases similar, e.g., for continuous, discrete, and weighted backprojections, inner product, or noniterative method with the minimal error. Hence, image reconstruction algorithms parallelized by these forms of parallelism are synchronous and have centralised control. Consequently, these algorithms match SIMD architectures. On the other hand, however, component processes demand an access to the shared (global) memory holding a measurement vector, reconstructed image, or precalculated data. This feature is characteristic of MIMD architectures.

A formal parallel computer model which combines both features, this is the Parallel Random Access Machine (PRAM). The PRAM assumes the use of identical processors the number of which is arbitrary working synchronously in the SIMD processing mode and having an access to the shared (global) unlimited memory, see Fig 1.

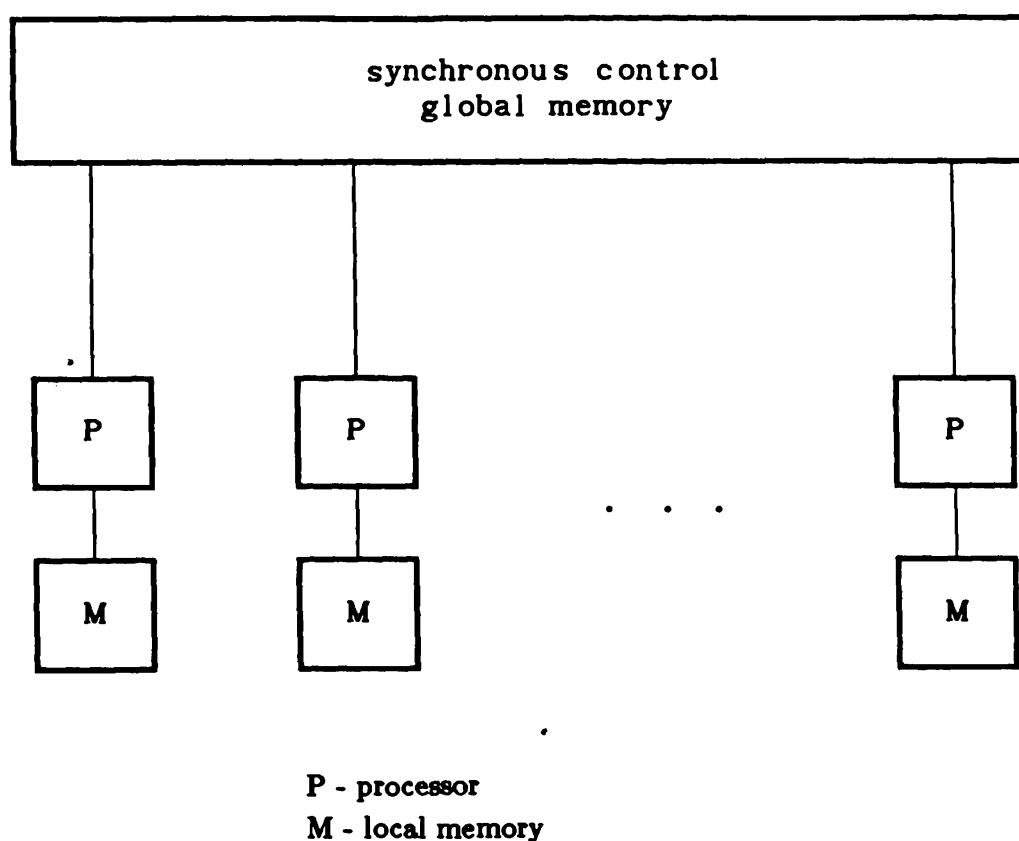


Figure 1. The Parallel Random Access Machine

In the CREW PRAM model, processors are allowed to read concurrently from one memory location but may not write concurrently into it. In the CRCW PRAM model, each of read and write operations can be executed in parallel. In the event of write conflicts in the global memory, several conventions are possible [14].

**It is proposed to consider the PRAM to be
the image reconstruction architecture.**

A vital problem of using a PRAM architecture for parallel image reconstruction is in the distribution of data among the global and local memories. In general, the strategy of data distribution depends on the type of mapping of the projection space into the image space. In [8] two types of mapping have been defined:

- *pixel-driven mapping* (i.e., for a given pixel all rays influent it are determined);
- *ray-driven mapping* (i.e., for a given ray all pixels affected by it are determined).

The type of mapping has a vital influence on the memory utilization. For the pixel-driven mapping any component process accesses just a single location in the image memory, while any location in the projection memory may have to be accessed by several processes. For the ray-driven mapping the situation is opposite. Therefore, for the pixel-driven mapping a measurement vector has to be kept in the global memory, whereas an image distributed among local memories. Conversely for the ray-driven mapping where an image should be stored in the global memory, while a measurement vector distributed among local memories.

Let us discuss some examples. Considered parallel image reconstruction algorithms will be expressed in the *occam* programming language.

The convolution method parallelized by projection parallelism can be expressed as follows (see [7])

```
PROC ConvolutionMethodProjectionParallelism (VAL INT M, J)
  INT m, j :
  SEQ
    PAR m = 0 FOR M      -- M - the number of projections
      ProjectionFiltration(m)
    PAR m = 0 FOR M
      SEQ j = 0 FOR J    -- J - the number of pixels
        BackprojectionSingleProjection(m,j)
  :
```

The `BackprojectionSingleProjection(m,j)` procedure actualizes the density of the j -th pixel contributed by the m -th projection. Note that this formulation is independent of the type of beam. In this case a measurement vector is distributed among local memories (constituting the projection memory), so that each local memory keeps a complete projection. In the first step (i.e., filtration) the m -th processor, $m = 0 \dots M-1$, filtrates the m -th projection and stores it in the m -th local memory. In the second step (i.e., backprojection) the m -th processor calculates the

contribution of the m -th filtered projection to the image density and actualizes the image memory (i.e., the global memory).

Exploiting pixel parallelism in order to parallelize backprojection, the convolution method can be formulated as

```
PROC ConvolutionMethodPixelParallelism (VAL INT M, J)
  INT m, j :
  SEQ
    PAR m = 0 FOR M
      ProjectionFiltration(m)
    PAR j = 0 FOR J
      Backprojection(j)
  :
```

Note that in this example to parallelize the method, two different forms of parallelism have been used. After filtration, filtered projections are in local memories. The next step, however, demands local memories to store a reconstructed image. Therefore, backprojection should be preceded by moving filtered projections from local memories to the global memory

```
PROC ConvolutionMethodPixelParallelismPRAM (VAL INT M, J)
  INT m, j :
  SEQ
    PAR m = 0 FOR M
      ProjectionFiltration(m)
      -- move contents of local memories to the global memory
    PAR j = 0 FOR J
      Backprojection(j)
  :
```

Consider simultaneous algorithms. They process all equations iteratively until a convergence condition has been satisfied. One of the most popular algorithms is iterated discrete backprojection (mentioned in Chapter 2). Recall that in every iteration it performs two steps. In the first step an image vector is projected and a residual calculated. In the second step the residual is backprojected. As each of the steps may be parallelized by ray parallelism, the algorithm is following

```
PROC IteratedDiscreteBackprojection (VAL INT I)
  WHILE NOT ConvergenceCondition
  SEQ
    PAR i = 0 FOR I    -- I - the number of rays
      Projection_Residual(i)
    PAR i = 0 FOR I
      DiscreteBackprojection(i)
  :
```

To perform the projection operation and calculate a residual, a reconstructed image should be kept in the global memory, whereas each local memory has to hold a ray sum. Then, the contents of both memories should be swapped in order to perform discrete backprojection.

4. Summary

In many medical and industrial applications parallel image reconstruction is required. Image reconstruction is open to parallelism, as pixel parallelism, projection parallelism, ray parallelism, operation parallelism, and plane parallelism are inherent in it. Component processes of each of pixel, projection, ray, and plane parallelisms are similar; in addition, component processes of operation parallelism are also similar in many cases. Therefore, image reconstruction algorithms exploiting these forms of parallelism are synchronous and have centralised control. An access to the global memory as well as the synchronous nature of algorithms parallelized by the introduced parallelism forms suggest that the architecture specific to image reconstruction is the Parallel Random Access Machine. This way the PRAM is a general image reconstruction architecture suitable either for transform methods or series expansion methods of image reconstruction. Some examples of implementation of parallel image reconstruction algorithms on a PRAM architecture have been discussed taking into account, especially, the distribution of data among memories.

5. References

1. A. R. Borges; A. M. D. Ferrari de Almeida, *A multiprocessor architecture for image reconstruction in CT*, in *Parallel Processing for Computer Vision and Display*, eds. P. M. Dew; R. A. Earnshaw; T. R. Heywood (Addison-Wesley, Wokingham, 1989) 489-496.
2. B. K. Gilbert; S. K. Kenue; R. A. Robb; A. Chu; A. H. Lent; E. E. Swartzlander, Jr., *Rapid execution of fan beam image reconstruction algorithms using efficient computational techniques and special-purpose processors*, IEEE Trans. Biomed. Eng., **BME-28**, 2 (1981) 98-115.
3. G. T. Herman, *Image Reconstruction from Projections. Fundamentals of Computerized Tomography* (Academic Press, New York, 1980).
4. N. Kingswood; E. L. Dagless; R. M. Belchamber; D. Betteridge; T. Lilley; J. D. M. Roberts, *Image reconstruction using the transputer*, IEE Proceedings, **133**, Pt. E, 3, May (1986) 139-144.
5. F. Natterer, *The Mathematics of Computerized Tomography* (Willey, New York, 1986).
6. W. L. Nowinski, *Image reconstruction methods and a tool for their investigation*, Journal of New Generation Computer Systems, **1**, 4 (1988) 327-352.

7. W. L. Nowinski, *Parallel implementation of the convolution method in image reconstruction*, Lecture Notes in Computer Science, **457** (1990) 355–364.
8. W. L. Nowinski, *Memory access synchronization in series expansion methods of parallel image reconstruction*, in *Real-time Systems with Transputers*, ed. H. Zedan (IOS Press, Amsterdam, 1990) 295–304.
9. W. L. Nowinski, *Parallel implementation of backprojection in image reconstruction*, Volume on Special Technical Contributions of the Joint International Conference on Vector and Parallel Processing CONPAR90/VAPP-IV, 10–13 September 1990, Zurich, Switzerland, C49–C52.
10. W. L. Nowinski, *Certain properties of parallelism in image reconstruction*, Proc. International Conference on Information Technologies for Image Analysis and Pattern Recognition – ITIAPR'90, 22–28 October 1990, Lvov, Soviet Union, Vol II, 199–203.
11. W. L. Nowinski, *Parallel implementation of series expansion methods of image reconstruction using the processor farm*. Transputing'91 World Conference, Sunnyvale, CA, USA, 22–26 April 1991 (accepted for presentation).
12. E. L. Ritman; J. H. Kinsley; R. A. Robb; L. D. Harris; B. K. Gilbert, *Physics and technical considerations in the design of the DSR: a high temporal resolution volume scanner*, AJR, **134**, Feb. (1980) 369–374.
13. E. E. Swartzlander, Jr.; B. K. Gilbert; I. S. Reed, *Inner product computers*, IEEE Trans. Comput., **C-27**, 1 (1978) 21–31.
14. P. G. Spirakis, *Fast parallel algorithms and the complexity of parallelism (Basic issues and recent advances)*, Mathematical Research, **48** (1988), Proceedings of the IV. International Workshop on Parallel Processing by Cellular Automata and Arrays – PARCELLA'88, 17–21 October 1988, Berlin, 177–189.

A COMPARISON OF INTERCONNECTION NETWORKS FOR LARGE SIMD PARALLEL COMPUTERS

MARKUS SCHWEHM

*Institut für Mathematische Maschinen und Datenverarbeitung VII,
Universität Erlangen-Nürnberg, Martensstr. 3, D-8520 Erlangen, Germany*

and

ALFRED STREY

*Institut für Mathematische Maschinen und Datenverarbeitung VII,
Universität Erlangen-Nürnberg, Martensstr. 3, D-8520 Erlangen, Germany*

ABSTRACT

A comparison of some static interconnection networks with different topologies for the use in large SIMD parallel computers is presented. The criterion of the comparison is the efficient realization of communication patterns often used in algorithms that are based on large one- or multidimensional arrays with regular communication structure.

1. Introduction

A lot of interconnection networks for SIMD Parallel Computers has been proposed, some networks are used in today SIMD machines:

- a two-dimensional mesh in the ILLIAC IV³ and in the MPP (Massively Parallel Processor⁵),
- a two-dimensional mesh with busses in each dimension in the AMT DAP (Distributed Array Processor) also installed at Erlangen University, or
- a 16-dimensional hypercube in the Connection Machine CM-2¹².

The performance of all these machines is not sufficient for solving very large scientific problems. Typical algorithms used here have one- or multidimensional arrays as basic data structure and can further be characterized by a regular communication structure. So the question arises which interconnection network is well suited for an SIMD machine with a large number of processing elements if the number of array dimensions and the number of elements per dimension in the algorithms are variable.

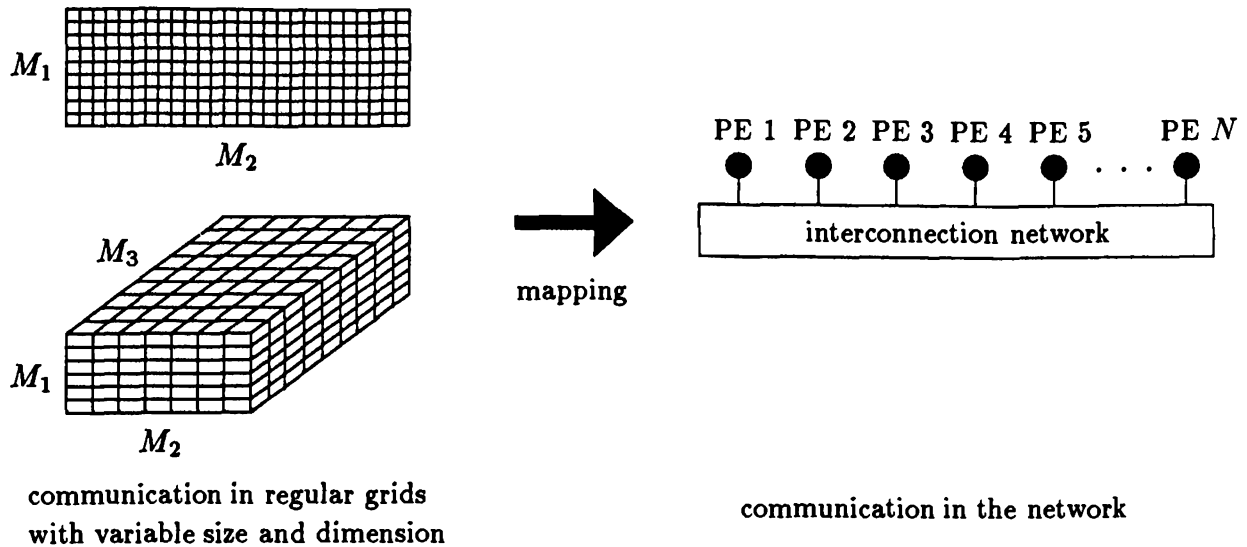


Figure 1: central problem

2. SIMD Computer Models

To compare interconnection networks, we have to select a set of networks for large SIMD computer models. An SIMD computer consists of a number N of processing elements (PEs), indexed 0 through $N - 1$ and denoted as $PE(i)$. The PE's can be enabled/disabled by a mask, and all enabled PEs perform the same instruction synchronously. In particular during a communication step only one type of connection can be used. So the interconnection pattern of an SIMD computer model is decisive for the efficiency of the realisation of communication patterns required by an algorithm. We consider the following interconnection networks:

Mesh Connected Computers (MCC k): The PEs are arranged as in a k -dimensional array (or mesh) $A(N_1, \dots, N_k)$, where N_i is the size of the i -th dimension. The total number of PEs is $N = \prod_{i=1}^k N_i$. The PE at location $A(i_1, \dots, i_k)$ is connected to the PEs at locations $A(i_1, \dots, i_j \pm 1 \bmod N_j, \dots, i_k)$ for $1 \leq j \leq k$. So each PE is connected to its nearest neighbors for each dimension, with wrap around connections for the PEs on the edges of the mesh, thus forming a torus topology. Alternatively, the PEs can be connected in a spiral topology, so that $PE(i)$ is connected to $PE(i \pm 1 \bmod N)$ for the first dimension and to $PE(i \pm \prod_{l=1}^{j-1} N_l)$ for dimension $j = 2, \dots, k$. On some existing machines this alternative is realized in software.

An example for MCC2 is the MPP with a 2-dimensional array of 128×128 PEs⁵. For comparison we will use the MCC2 as well as a proposed MCC3 model to see the effect of increasing the dimensionality of MCC models.

Mesh Connected Computers with Busses (BusMCC k): Additional to the MCC interconnection network, a bus system connects all the PEs in each dimension. During a communication step, only one bus system can be used. For each bus of the used bus system, one (selected) PE can send one data item to the bus, which thus is broadcast to any (nonmasked) other PE along this bus.

An example for BusMCC2 is the AMT DAP with a 2-dimensional array of 32×32 (or 64×64) PEs and two bus systems⁸. As for the MCC model we will use the BusMCC2 as well as a proposed BusMCC3 in the following.

Cube Connected Computer (CCC): The CCC model is essentially a MCC with only two PEs per dimension. If the number of PEs is $N = 2^k$, the PEs are arranged as in a k -dimensional array (or cube) $A(2,2,\dots,2)$. Let $i_{k-1}, \dots, i_0$ be the binary representation of i for $i \in \{0, \dots, N-1\}$ and let $i^{(b)}$ be the number obtained, if the b -th bit of i is complemented (thus $i_{k-1}, \dots, \bar{i}_b, \dots, i_0$) for $0 \leq b < k$. Then $PE(i)$ is connected to $PE(i^{(b)})$ for $0 \leq b < k$ and during one communication step data can only be transmitted along one dimension.

An example for this model is the Connection Machine with $N = 65536$ and $k = 16^{12}$.

Perfect Shuffle Computer (PSC): The PSC model is derived from the notion of shuffling a deck of cards. If the deck is divided into two exact halves and shuffled perfectly, the cards are moved inside the deck in the same way as data items would be moved inside an array of PEs by a shuffle communication step of the PSC model. Using k , i and $i^{(b)}$ as in the cube model, $PE(i)$ is connected to $PE(i^{(b)})$, $PE(i_{k-2}, \dots, i_0, i_{k-1})$ and $PE(i_0, i_{k-1}, \dots, i_1)$. These three connections are called exchange, shuffle and unshuffle.

Although there exists no realisation of a PSC, this is a very promising network as we will see in the following. Another network, closely related to the PSC model is the binary de Bruijn network. It has four connections: shuffle and unshuffle as defined for the PSC model, but instead of the exchange connection it has two combined connections shuffle-exchange and exchange-unshuffle which realize the serial execution of the two PSC connection steps in one cycle. As the de Bruijn network can easily be emulated on the PSC model, we introduce only the PSC model for further investigation.

The following table lists some hardware characteristics of the selected networks. In the following, the technological constraints will be ignored. Instead we concentrate on the communication costs implied by the networks, because they are a limiting factor for parallel processing on SIMD type computers.

3. Classification of Communication Patterns

The comparison of interconnection networks can only be done in relation to a certain class of algorithms. Our focus lies on very large problems as they arise in numerical

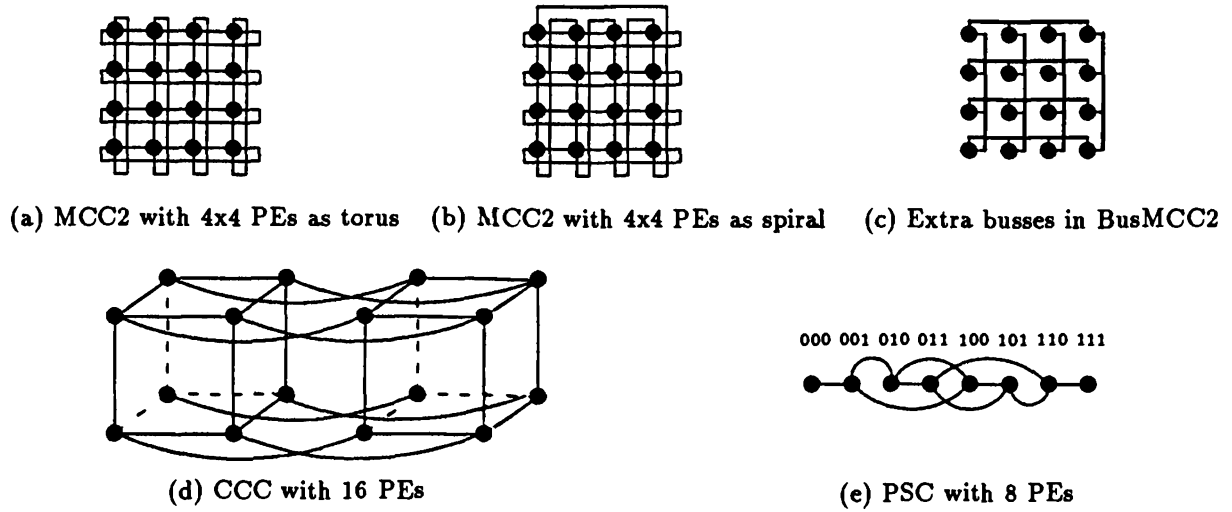


Figure 2: The selected SIMD computer models

Table 1: Characteristics of some static networks

	Number of connections per node	Total number of connections per network	Diameter
MCC1	2	$2 \cdot N$	$N/2$
MCC2	4	$4 \cdot N$	$\sqrt{N}$
BusMCC2	6	$4 \cdot N + 2 \cdot \sqrt{N}$	2
MCC3	6	$6 \cdot N$	$\frac{3}{2} \cdot \sqrt[3]{N}$
BusMCC3	9	$6 \cdot N + 3 \cdot \sqrt[3]{N}$	3
CCC	$\lg N$	$N \cdot \lg N$	$\lg N$
PSC	3	$3 \cdot N$	$2 \cdot \lg N - 1$

analysis, namely in finite element analysis (e.g. for fluid dynamics). For these problems a large amount of data is organized in a regular fashion, and each data element will be processed in a similar way. Thus we concentrate on data parallel algorithms, which are well suited to SIMD computers.

Algorithms for parallel processors can be divided into computational and communicational parts. As we want to compare interconnection networks, we are mainly interested in the communication effort of an algorithm. Communication of a data parallel algorithm is defined as any dependency of a value $x(i)$ from another value $y(j)$, where $i = (i_1, \dots, i_k)$ and $j = (j_1, \dots, j_k)$ are k -dimensional indexes and $j = f(i, \dots)$ describes the dependency of these indexes.

The communication structure of an algorithm can be classified as follows:

- **local/global:** A communication is called local, if $j_l = i_l \pm 1$ for any dimension $l \in \{1, \dots, k\}$ and else global.
- **static/dynamic:** A communication is called static, if the communication pattern f is fixed at compile-time, i.e. f is only dependent from i , else dynamic. In this paper we confine on static communication patterns.
- **regular/irregular:** A communication is called regular, if the communication pattern f can be described by easy arithmetic expressions so that the computation time for f can be neglected, else irregular.

In this paper we define communication patterns to be regular, if they can be described as affine index function as follows:

$$(j_1, \dots, j_k) = f(i_1, \dots, i_k) = (f_1(i_1, \dots, i_k), \dots, f_k(i_1, \dots, i_k)) \quad (1)$$

with

$$f_l(i_1, \dots, i_k) = \sum_{l=1}^k a_l i_l + b, \quad 1 \leq l \leq k \quad (2)$$

Most algorithms in numerical analysis can be expressed by using this kind of communication exclusively. The following classes of communication patterns can be found in such algorithms:

Regular communication patterns:

- **translation:** $y(\dots, i_l, \dots) = x(\dots, i_l + d, \dots)$, $i_{\min} \leq i_l \leq i_{\max}$, $d \in \mathbb{N}$
- **linear compression:** $y(\dots, i_l, \dots) = x(\dots, a i_l, \dots)$, $i_{\min} \leq i_l \leq i_{\max}$, $a \in \mathbb{N}$
- **linear expansion:** $y(\dots, a i_l, \dots) = x(\dots, i_l, \dots)$, $i_{\min} \leq i_l \leq i_{\max}$, $a \in \mathbb{N}$
- **reversion:** $y(\dots, i_l, \dots) = x(\dots, a - i_l, \dots)$, $i_{\min} \leq i_l \leq i_{\max}$, $a \in \mathbb{N}$
- **broadcast:** $y(\dots, i_l, \dots) = x(\dots, a, \dots)$, $i_{\min} \leq i_l \leq i_{\max}$, $a \in \mathbb{N}$

Regular communication patterns combined with associative operator:

- **reduction:** $y(\dots, 1, \dots) = \sum_{i=i_{\min}}^{i_{\max}} x(\dots, i, \dots)$
- **prefix:** $y(\dots, i_l, \dots) = \sum_{i=i_{\min}}^{i_l} x(\dots, i, \dots)$, $i_{\min} \leq i_l \leq i_{\max}$

Irregular communication patterns (both can be static or dynamic):

- general permutation: $y(\text{dest}(i)) = x(i)$
- general broadcast: $y(i) = x(\text{source}(i))$

These patterns are called communication primitives. In the next section it will be described how these primitives can be realized on the selected set of SIMD computer models.

4. Realization of Communication Patterns on the Networks

The basic idea for the realization of the communication patterns is the *emulation* of important dynamic multistage networks on the static networks of the selected computer models. These networks consist of $O(\log N)$ stages with binary or ternary switches (see figure 3) that are simulated by the PEs with precomputed masks. Some

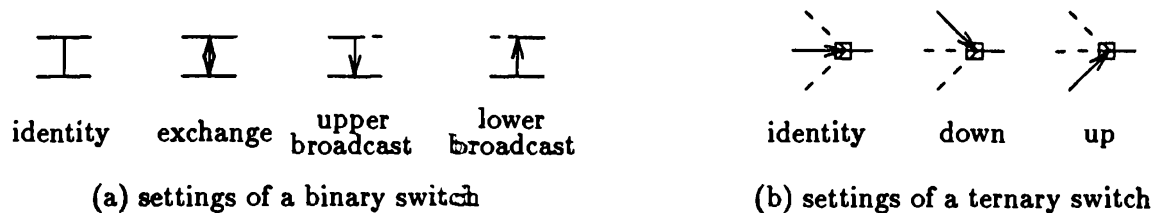


Figure 3: Switches in dynamic multistage networks

of these networks are shown in figure 4 as line diagramms with the inputs on the left and the outputs to the right side.

- The **Omega** network suggested by Lawrie⁷ is a versatile network for the realization of *affine* communication patterns. It has N inputs/outputs and $\lg N$ switching stages, each stage consisting of $\frac{N}{2}$ binary switches. There are two equivalent representations for the omega network, based either on the (un)shuffle or the cube interconnection structure, see figure 4(a). The allowed settings are *identity*, *exchange*, *upper broadcast* and *lower broadcast*. Lawrie proved that the Omega network Ω_N with N inputs x and N outputs y can realize all communication patterns $y(ci + d) = x(ai + b)$ in one pass, if $\text{ggT}(a, N) \leq \text{ggT}(c, N)$. The inverse Omega network can realize all affine communication patterns in one pass, if $\text{ggT}(a, N) \geq \text{ggT}(c, N)$.
- The **ADM** network first proposed by Feng⁴ and later modified by Siegel et al.⁹ uses ternary switches, each switch is set in one of the positions *identity*, *down* or *up*. It includes the functionality of the Omega network but has some additional communication capabilities. It can be emulated efficiently on the MCC models only.

- The **Benes network**¹ allows any (irregular) permutation in one pass. It consists of an inverse Omega network followed by an Omega Network. The last stage of the inverse Omega Network and the first stage of the Omega Network are identical, so one of them can be omitted.
- The **Generalized Connection Network (GCN)** can realize any (irregular) broadcast pattern in one pass. We use the GCN proposed by Thompson¹¹. It represents a concatenation of Omega-, inverse Omega- and Benes network.
- The **Ascend Network** is an operation network which has the structure of an inverse Omega network but performs in each stage a binary operation on two inputs.

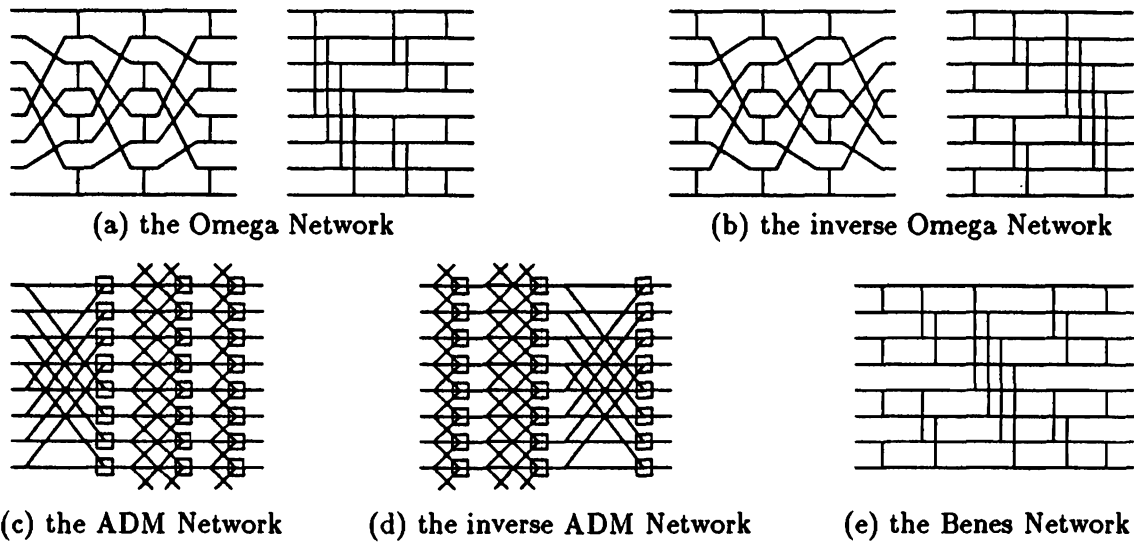


Figure 4: Dynamic networks with 8 inputs and 8 outputs

By referring either to the shuffle or to the cube representation of the dynamic network all dynamic networks can be emulated on the static networks of our models. Table 2 shows how to realize the communication primitives on all models, each entry represents the name of the dynamic network that must be emulated. The entry *direct* indicates that the communication pattern can be realized by a direct use of one of the interconnection functions of the model, the entry *special* refers to an implementation described in the literature.

Figure 5 shows the number of communication steps for the realization of reduction and of general permutation when varying the machine size.

Table 2: Realization of all communication patterns

	MCC	BusMCC	PSC	CCC
translation	direct		Ω	Ω
compression	iADM		$i\Omega$	$i\Omega$
expansion	ADM		Ω	Ω
1-dim. affine communication	$\Omega / i\Omega$			
broadcast	global bus			
reduction	special ²	special ⁶	Ascend	Ascend
prefix	special ¹⁰	special ¹⁰	Ascend	Ascend
general permutation	Benes			
general broadcast	GCN			

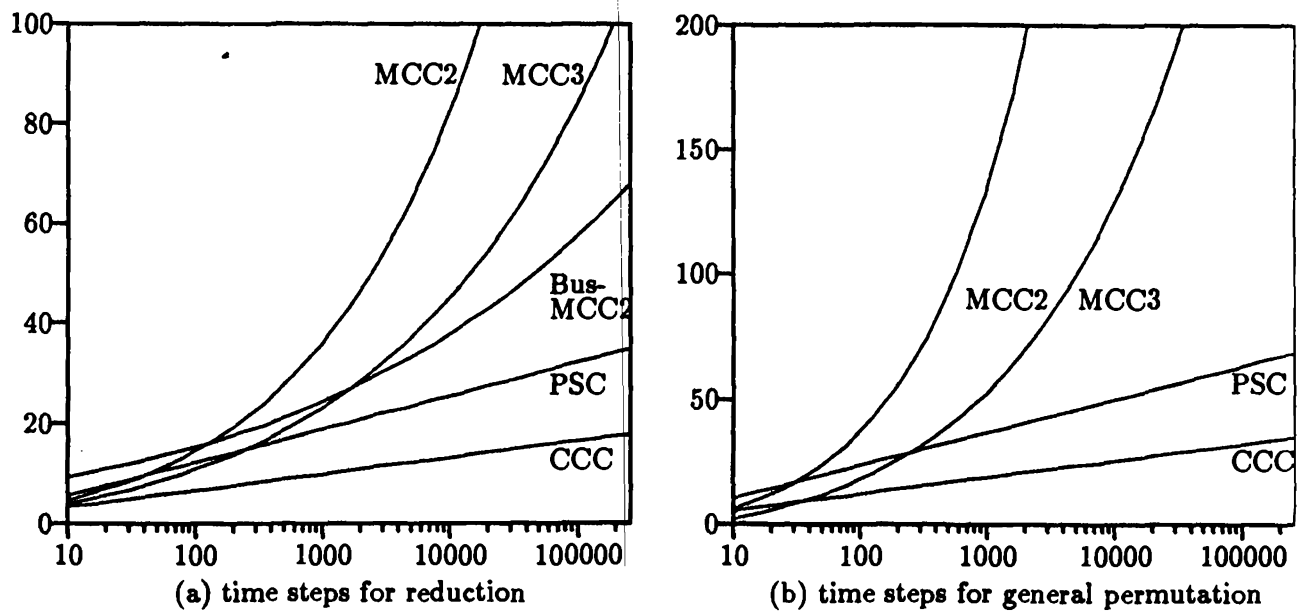


Figure 5: Number of communication steps needed when varying the machine size

5. Partitioning and Mapping

Each PE has a onedimensional index $i \in \{0, \dots, N-1\}$. This index has the binary representation $i = (i_{n-1} i_{n-2} \dots i_1 i_0)$. If a multidimensional computation requires a communication in only one of k array dimensions, the SIMD computer model must be partitioned in k dimensions. There are three different partitioning methods:

- The natural partitioning of the MCC models uses in each dimension the number of PEs of the underlying grid. The models MCC2 and BusMCC2 can be partitioned in two dimensions with $\sqrt{N}$ PEs each, the models MCC3 and BusMCC3 can be naturally partitioned either in three dimensions with $N^{\frac{1}{3}}$ PEs or in two dimensions with $N^{\frac{1}{2}}$ and $N^{\frac{1}{2}}$ PEs. In each dimension the resulting

Table 3: Number of communication steps needed for the (1-dimensional) realization of all communication patterns on the different networks

	MCC1	BUS-MCC2	MCC2	BUS-MCC3	MCC3	PSC	CCC
translation by d	d	$d \operatorname{div} \sqrt{N} + d \bmod \sqrt{N} = O(\sqrt{N})$		$d \operatorname{div} N^{\frac{2}{3}} + (d \bmod N^{\frac{2}{3}}) \operatorname{div} N^{\frac{1}{3}} + d \bmod N^{\frac{1}{3}} = O(\sqrt[3]{N})$		$2n$	n
compression/expansion	$\leq N - 1$	$\leq 2 \cdot (\sqrt{N} - 1)$		$\leq 3 \cdot (\sqrt[3]{N} - 1)$		$2n$	n
1-dim. affine communication	$\leq \frac{3}{2}N - 2$	$3 \cdot \sqrt{N} - 4$		$\frac{9}{2} \cdot \sqrt[3]{N} - 6$		$2n$	n
reduction	$3\sqrt{N} - 3$	$4\sqrt[3]{N} - 4$	$9\sqrt[6]{N} - 4$	$5\sqrt[4]{N} - 5$	$30\sqrt[12]{N} - 9$	$2n - 1$	n
prefix	$3\sqrt{N} - 3$	$4\sqrt[3]{N} - 4$	$4\sqrt[4]{N} - 3$	$5\sqrt[5]{N} - 5$	$6\sqrt[6]{N} - 4$	$3n - 1$	n
general permutation	$4N - 3$	$\frac{9}{2} \cdot \sqrt{N} - 8$		$\frac{13}{2} \cdot \sqrt[3]{N} - 12$		$4n - 3$	$2n - 1$
general broadcast	$6N - 8$	$\frac{17}{2} \cdot \sqrt{N} - 12$		$\frac{25}{2} \cdot \sqrt[3]{N} - 18$		$8n - 7$	$4n - 3$

network has the functionality of an MCC model with lower dimensionality.

- The binary partitioning of all models is based on the binary representation of the PE index i with $n = \lg N$ Bits. Groups of m_i contiguous bits form a dimension with $M_i = 2^{m_i}$ PEs, so the model is partitioned in k dimensions with $2^{m_1} \times 2^{m_2} \times \dots \times 2^{m_k}$ PEs, $\sum_{j=1}^k m_j \leq n$.

- The **linear partitioning** is formed by first counting up the index j_1 in dimension 1 (beginning with PE 0), then increasing the index j_2 of dimension 2 and again counting up the PEs in dimension 1, and so on. When using the linear partitioning for generating k model dimensions with $M_1 \times M_2 \times \dots \times M_k$ PEs, the onedimensional PE index i can be identified by the formula

$$i = j_1 + A_2 \cdot j_2 + A_3 \cdot j_3 + \dots + A_k \cdot j_k \text{ with } A_p = \prod_{q=1}^{p-1} M_q. \quad (3)$$

In the rest of the paper we concentrate on the important binary partitioning of the computer models PSC and CCC, for the MCC models we consider only the natural partitioning. A binary partitioning of the MCC is possible but in this case the time complexity for the realization of a communication pattern in only one model dimension equals the time complexity for the realization on the complete MCC model.

Every binary partitioning of the CCC model is ideal because each partition with $P = 2^p$ PE has the full functionality of a p -dimensional CCC model. On the PSC model with $N = 2^n$ PEs most communication patterns can be realized in a binary partition with $P = 2^p$ PEs by shuffling or unshuffling n times and using the exchange interconnection function only in those p steps where one of the p index bits of the corresponding partition is located in the least significant index bit.

Our approach to *mapping* is rather simple. First the model is partitioned by one of the methods described above, then the j th element of each array dimension is mapped on the j th element of the corresponding dimension of the partitioned computer model. A special communication pattern only relevant in a partitioned model is the *regular multicast*. This is a broadcast in each dimension of the model, where many elements (e.g. all elements with index c in one array dimension) have to be broadcasted in one step. Table 4 shows the number of communication steps needed for the realization of the communication patterns in one dimension of a binary partitioned PSC or CCC model with $P = 2^p$ PEs and in one dimension of a natural partitioned MCC k or BusMCC k model with $\sqrt[p]{N}$ PEs.

6. Results

Considering tables 3 and 4 the following conclusions can be drawn:

- The best model is surely the CCC model on which the least number of communication steps is necessary for the realization of all communication patterns.
- Except for model partitioning, the PSC model needs in most cases only twice the number of communication steps as the CCC model. It needs only three connections per PE compared to $\lg N$ connections per PE in the CCC model.

Table 4: Number of communication steps for the realization of some communication patterns in one dimension of a partitioned model

	MCC2	BusMCC2	MCC3	BusMCC3	PSC	CCC
translation by d	d		d		$n + p$	p
compression/ expansion	$\leq \sqrt{N} - 1$		$\leq \sqrt[3]{N} - 1$		$n + p$	p
1-dim. affine communication	$\leq \frac{3}{2}\sqrt{N} - 2$		$\leq \frac{3}{2}\sqrt[3]{N} - 2$		$n + p$	p
regular multicast	$\frac{3}{2}\sqrt{N} - 2$	1	$\frac{3}{2}\sqrt[3]{N} - 2$	1	$n + p$	p
reduction	$\sqrt{N} - 1$	$3\sqrt[3]{N} - 3$	$\sqrt[3]{N} - 1$	$3\sqrt[3]{N} - 3$	$n + p$	p
general permutation	$4\sqrt{N} - 3$		$4\sqrt[3]{N} - 3$		$2n + 2p - 1$	$2p - 1$

Also in a partitioned PSC model every regular and irregular communication pattern can be realized in $O(\log N)$ steps. Considering simple PEs with a word length of 1 bit, the PSC model is a good model for large SIMD Parallel Computers. Here the communication time for a typical data word length of 32 Bits is smaller than the time to access a data word in the memory of the PEs.

- In contrast to the PSC and CCC models the MCC models do not have an efficient binary partitioning. Regarding large MCC models it is difficult to map an array with a different number of elements in each dimension on the PEs without using data partitioning.

- On the MCC2 model used in most today SIMD parallel computers the complexity for the realization of all patterns is $O(\sqrt{N})$, not to mention the high proportionality constants. So the number of communication steps in a model with a large number of PEs is enormous compared to the corresponding number of steps for the PSC model.
- The upgrade from MCC2 to MCC3 model increases performance from $O(\sqrt{N})$ to $O(\sqrt[3]{N})$ at the cost of two additional connections per PE.

References

1. V.E. Beneš. On Rearrangeable Three-Stage Connecting Networks. *The Bell System Technical Journal*, September 1962.
2. S.H. Bokhari. Finding Maximum on an Array Processor with a Global Bus. *IEEE Transactions on Computers*, 33(2):133-139, 1984.
3. W.J. Bouknight, S.A. Denenberg, D.E. McIntyre, J.M. Randall, A.H. Sameh, and D.L. Slotnick. The Illiac IV System. In *Computer Structures: Principles and Examples*. McGraw-Hill, New York, 1982.
4. T.-Y. Feng. Data Manipulating Functions in Parallel Processors and Their Implementations. *IEEE Transactions on Computers*, 23(3):309-318, 1974.
5. P.A. Gilmore. The Massively Parallel Processor. In S. Fernbach, editor, *Supercomputers*. North-Holland, 1986.
6. V.K.P. Kumar and C.S. Raghavendra. Array Processor with Multiple Broadcasting. *Journal of Parallel and Distributed Computing*, 4:173-190, 1987.
7. D. Lawrie. Access and Alignment of Data in an Array Processor. *IEEE Transactions on Computers*, 24(12):1145-1155, 1975.
8. D. Parkinson and J. Litt, editors. *Massively Parallel Computing with the DAP*. Pitman, London, 1990.
9. H.J. Siegel and D. Smith. Study of Multistage SIMD Interconnection Networks. In *5th Int. Symp. on Computer Architecture*, 1978.
10. A. Strey. *Ein Vergleich von Verbindungsnetzwerken für große SIMD-Parallelrechner*. PhD thesis, University Erlangen-Nürnberg, 1991. (in preparation).
11. C.D. Thompson. Generalized Connection Network for Parallel Processor Intercommunication. *IEEE Transactions on Computers*, 27(12):1119-1125, 1978.
12. L.W. Tucker and G.G. Robertson. Architecture and Application of the Connection Machine. *IEEE Computer*, pages 26-38, 1988.

Space-Time Transformation of Cellular Computations

O.L.Bandman

*Computing Center, Siberian Division of Academy
of Science, 630090, Novosibirsk, USSR.*

Abstract

A formalism for changing the depth of cellular computation parallelism at the expense of its slow-down is presented. A method is created to get from the computation within a k -dimensional cellular array of finite size to result-equivalent computation within another array of smaller size. The transformation is given in terms of parallel substitution algorithm theory.

1.Introduction

The architecture design of cellular array computers comprises the problem of an appropriate trade-off between hardware cost and computation speed. To achieve this, we need some formalism for changing the depth of computation parallelism at the expense of its slow-down. Formal methods of equivalent transformations of algorithmic descriptions used to reduce the number of processor elements at the cost of longer computation time are called *space-time transformations* (ST-transformations). ST-transformations were first investigated in Iterative Systems[1]. Some useful results were also obtained for systolic algorithms [2].

The purpose of this paper is to derive ST-Transformation for cellular computations given in the form of parallel microprograms [3,4]. The following way of solving this problem is proposed. A procedure is created to get from the computation within a k -dimensional cellular array of finite size to result-equivalent computation within another array of smaller size. Necessary and sufficient conditions are defined an initial cellular microprogram is to meet, so that the resulting one be correct (determinate and terminating). ST-Transformation at work is shown for a well known cellular array - the content-addressable memory [5].

2. Parallel Microprogram and its Interpretation

Parallel microprograms constitute an important class of parallel substitution algorithms [3,4]. They provide a concise mapping of distributed computation processes in cellular arrays. The *cellular array* is given by the set $W = \{(a_i, m_i)\}$, where $(a, m) \in A \times M$ is a *cell*, A is the alphabet of states, M is a set of names, each cell of W bearing a distinct name. A *parallel microprogram* $\Phi(A, M)$ is an unordered set of *parallel microinstructions* of the form

$$\Theta: S'(m) * S''(m) \Rightarrow S'''(m), \quad (1)$$

where $S'(m) = \{(a_i, \phi_i(m))\}$, $S''(m) = \{(b_j, \psi_j(m))\}$, $S'''(m) = \{(c_i, \phi_i(m))\}$, $i = 1, \dots, p$, $j = 1, \dots, q$; for any $i \neq j$ $\phi_i(m) \neq \phi_j(m)$; $a_i, b_j, c_i \in A$; $m, \phi_i(m), \psi_j(m) \in M$. Expressions like S' , S'' and S''' are called *configurations*. $S'(m) * S''(m) = S^*(m)$ is referred to as the *left-hand side* of the microinstruction, $S'(m)$ being the *base*, $S''(m)$ - the *context*. The functions $\phi_i(m)$ and $\psi_j(m)$ are called *naming functions*. The operation $*$ in (1) means that for any $m \in M$

$S^*(m) = S'(m) * S''(m) = S'(m) \cup S''(m)$. We will later consider parallel microprograms in which coordinate vectors of k -dimensional integer lattice of finite size are the names, and linear shifts along the lattice axes are the naming functions.

The parallel microinstruction $\Theta \in \Phi(A, M)$ is applicable to $W \in A \times M$ if there exists such $m \in M$ that its left-hand side is included in W , i.e.

$S^*(m) \in W$. The application of Θ to W in a cell named $m \in M$ is called a *microoperation* $\Theta(m)$ and is in substituting the states c_i of the cells $(c_i, \phi_i(m)) \in S'''(m)$ for the states a_i of the cells $(a_i, \phi_i(m)) \in S'(m)$. Cells from the context $S''(m)$ are not changed by the microinstruction $\Theta(m)$.

To represent more complicated transformation than just a state substitution, variable symbols $x, y, \dots$ and functions of the form $f(x, y)$ with the value domain and range from A are used. Notation like

$$\{(x_i, \phi_i(m))\} * \{(y_j, \psi_j(m))\} \Rightarrow \{(f_i(x_i, y_j), \phi_i(m))\}, \quad (2)$$

$$i = 1, \dots, p, j = 1, \dots, q,$$

is called a *functional microinstruction*. It is applicable to W , if states of cells in W belong to the domain of variables of the left-hand side. Its application replaces cell states from the base by the function values of the equally named cells of the right-hand side.

Synchronous [3] and asynchronous [6] modes of computations in cellular arrays have been examined. *Synchronous computation* $C_S(W_0)$ is an iterative procedure such that at each i -th step all the applicable

to W_{i-1} microoperations are executed. *Asynchronous computation* $C_A(W_0)$ implies that every microoperation is eligible for execution when it proves applicable, with only one microoperation executed at every step. In both cases the computation terminates after a certain W_F is achieved with no microinstruction applicable, $W_F = \Phi(W_0)$ being called the *result* of the computation $C_S(W_0)$. A computation $C_S(W_0)$ (or $C_A(W_0)$) may be associated with a *computation graph* $G_S(W_0)$ (or $G_A(W_0)$) whose vertices correspond to reachable cellular arrays $\{W_0, \dots, W_F\}$, an arc from W_i to W_j exists if W_j is one-step result of applying the microprogram to W_i . A computation is *determinate*, iff the associated computation graph has only one termination vertex. A computation is *terminating*, if the associated graph has no circuits.

A set of synchronous (asynchronous) computations for all $W_0 \in A^*M$ is called a *synchronous (asynchronous) interpretation* of a parallel microprogram $\Phi(A, M)$ and is referred to as Ψ_S (Ψ_A). A synchronous (asynchronous) interpretation is *correct*, if the computations $C_S(W_0)$ ($C_A(W_0)$) are determinate and terminating for any initial cellular array $W_0 \in A^*M$ [7].

Example. The parallel microprogram for adding many binary numbers has the alphabet $A = \{0, 1\}$, the set of names $M = \{ \langle i, j \rangle \}$ is a two-dimensional rectangular table whose cells are numbered in accordance with the left-coordinate system. Binary numbers are written into the rows of the table, less significant bits being in the column $i=1$.

$$\begin{aligned} \Theta_1: & \{ (1, \langle i, j \rangle) (1, \langle i, j+1 \rangle) (0, \langle i+1, j \rangle) \} * \{ (0, \langle i+1, j-1 \rangle) (0, \langle i, j-1 \rangle) \} \Rightarrow \\ & \{ (0, \langle i, j \rangle) (0, \langle i, j+1 \rangle) (1, \langle i+1, j \rangle) \}; \\ \Theta_2: & \{ (1, \langle i, j \rangle) (0, \langle i, j+1 \rangle) \} * \{ (0, \langle i, j-1 \rangle) \} \Rightarrow \{ (0, \langle i, j \rangle) (1, \langle i, j+1 \rangle) \}; \end{aligned}$$

Fig.1 shows the geometric patterns of Θ_1 and Θ_2 as well as the fragments of synchronous and asynchronous computation graphs in computing the sum of the integers 2, 3 and 5. It is seen that both computations are determinate and terminating.

The property of determinacy is related to the concept of *dangerous intersections* of microinstructions. The two microinstructions $\Theta_1, \Theta_g \in \Phi(A, M)$ have a dangerous intersection, if there is a vector d , such that

$$S(m) = S'_1(m) \cup S_g^*(m+d) \quad (3)$$

If the condition (3) holds for any pair of $\Theta_1, \Theta_g \in \Phi(A, M)$ then the cellular array $W = S(m)$ is called *dangerous*.

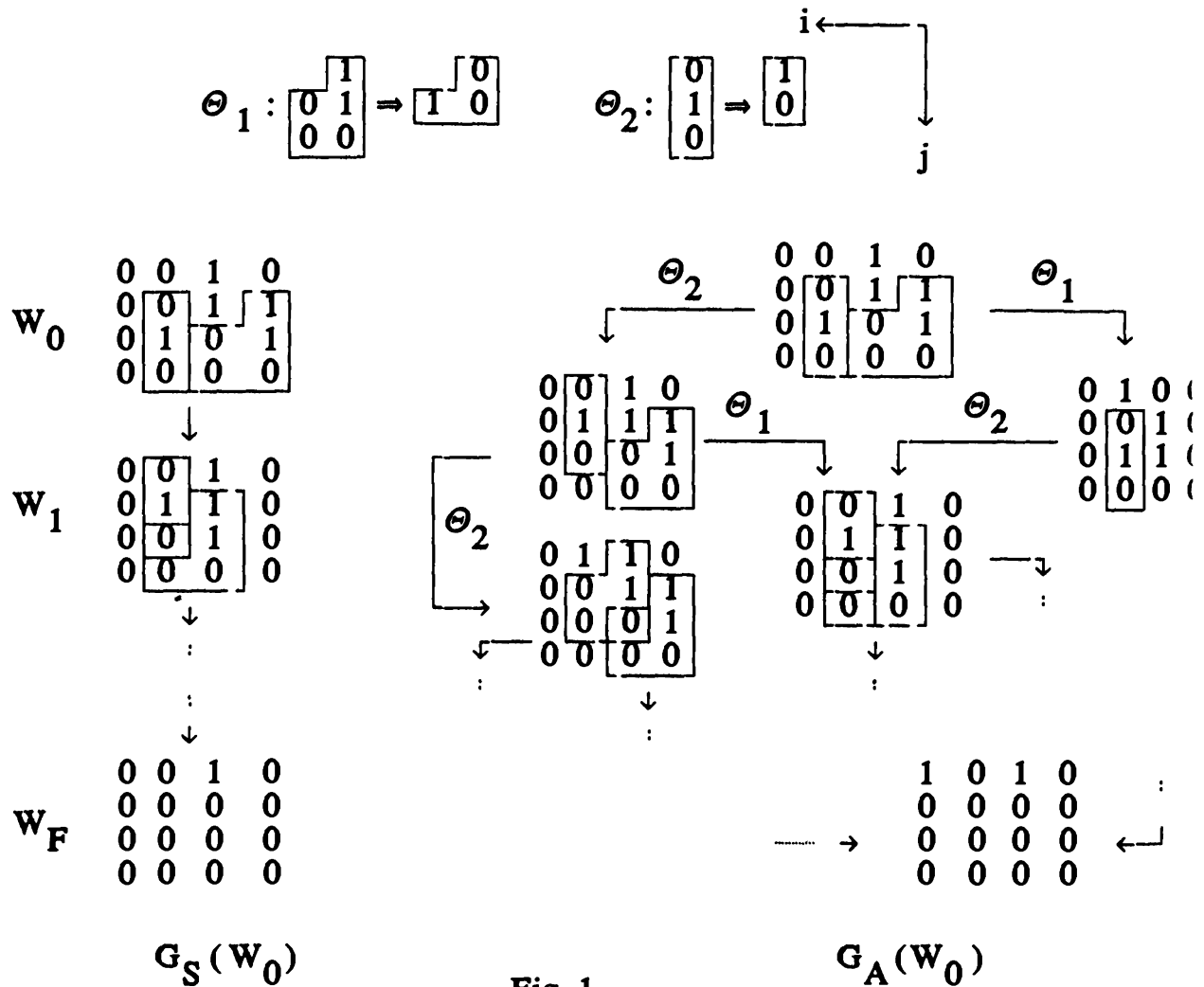


Fig. 1.

A synchronous interpretation is determinate, iff cellular arrays resulting from one-step application of Φ to all dangerous $W=S(m)$ contain not a single pair of cells (c_i, m_i) and (c_j, m_j) , such that $m_i=m_j$, $c_i \subseteq c_j$ [7]. Asynchronous interpretation is determinate, iff asynchronous computations $C_A(W_0)$ are determinate for all dangerous $W_0=S(m)$ [7].

Termination conditions for two interpretations are formulated through the concept of a *provocation graph* as follows. A synchronous (asynchronous) interpretation of a parallel microprogram is terminating, iff no provocation graph $\Gamma_S(\Theta_i)$ for synchronous ($\Gamma_A(\Theta_i)$ for asynchronous) interpretation with an $\Theta_i \in \Phi$ contains a circuit. The methods of constructing these graphs are described in [7] in details. Here we give only the main idea. Construction of both graphs $\Gamma_A(\Theta)$ and $\Gamma_S(\Theta)$ is an iterative procedure which starts with a vertex

assigned $W_1(m) = S_i^*(m)$. Let $\{W_1(m), \dots, W_j(m)\}$ be a set of vertices of a constructed part of $\Gamma_S(\Theta_i)$ ($\Gamma_A(\Theta_i)$). Then the j -th step of the iterative procedure is as follows.

1. If there exists $m \in M$ such that Φ is applicable to $W_j(m)$, then $W_{j+1}(m)$ which is the one-step result of the synchronous $C_S(W_j(m))$ (asynchronous $C_A(W_j(m))$) computation is added to the graph together with the arc going from W_j to W_{j+1} .

2. If Φ is not applicable to $W(m)$ for any $m \in M$, then $W_j(m)$ is extended in order to provoke its application. This is done in different ways for synchronous and asynchronous interpretations. In the case of $\Gamma_S(\Theta_i)$ the concept of *configurations chain* is used. A configuration chain, generated by $W(m)$, is the union of configurations $W(m), S_{k1}^*, \dots, S_{kl}^*$, where $S_{k1}^*, \dots, S_{kl}^*$ are left-hand sides of a maximal set of microinstructions which has a connected graph of intersections, and $W(m)$ has nonempty intersection with at least one of them. So, if Φ is not applicable to $W_j(m)$, then new vertices equal to all configuration chains generated by $W_j(m)$, are added to the graph together with arcs going from $W_j(m)$ to them. In the case of $\Gamma_A(\Theta_i)$ each new vertex is equal to the union of $W_j(m)$ with the left-hand side of a microinstruction, if they have a nonempty intersection. These vertices are added to the graph with arcs going from $W_j(m)$ to them.

3. If the conditions from 1 and 2 do not hold, or the size of $W_j(m)$ exceeds the size of M , then $W_j(m)$ is a terminating vertex of the graph.

4. If $W_j(m)$ is contained in or equal to any previously obtained vertex, then an arc from $W_j(m)$ to this vertex completes a circuit, which indicates that the interpretation Ψ_S (Ψ_A) is nonterminating.

3. Formal Definition of ST-Transformation

Let a parallel microprogram $\Phi(A, M)$ be given. In the name set M , three subsets may be distinguished: M_0 is a subset of names of cells storing initial data; M_P is a subset of names of cells responsible for processing data referred to as *processor names* set; M_F is a subset of names, whose states are computation results. The intersection of M_0 , M_P and M_F is generally nonempty. It is assumed that cells named M_0 are driven into original state in advance, using a special microprogram. Similarly, the result is read out from cells named M_F after the computation by Φ is completed.

Definition 1. Synchronous interpretations Φ_S and Φ'_S of parallel microprograms $\Phi(A, M)$ and $\Phi'(A, M)$ are called *equivalent*, if for any pair of synchronous computations $C_S(W_0)$ and $C'_S(W'_0)$ the following is valid

$$\text{if } W_0(M_0) = W'_0(M'_0) \text{ then } W_F(M_F) = W'_F(M'_F), \quad (4)$$

where W_F and W'_F are results of computations $\Phi(W_0)$ and $\Phi'(W'_0)$ respectively, and $W_0 = W'_0 \cap A \times M$.

Example The parallel microprogram $\Phi(A, M) = \{\Theta\}$, A is a set of numbers, x, y, z are variables with values from A , $M = \{0, 1, \dots, n\}$,

$$\Theta: \{(x, i)\} * (y, 0) \Rightarrow \{(x \cdot y, i)\}, i = 1, \dots, n,$$

multiplies the vector $(x_1, x_2, \dots, x_n)$ by the scalar y in one step. If the initial cellular array is $W_0 = \{(c, 0)(a_1, 1) \dots (a_n, n)\}$, then the result equals $W_F = \{(a_1 \cdot c, 1) \dots (a_n \cdot c, n)\}$. Here $M_0 = M$, $M_P = M_F = \{1, \dots, n\}$. The microprogram $\Phi'(A', M') = \{\Theta', \delta, \gamma\}$ with $A' = A$, $M' = M \cup \{n+1\}$, where

$$\begin{aligned} \Theta' &: (x, 1) * (y, 0)(1, n+1) \Rightarrow (x \cdot y, 1), \\ \delta &: \{(x, i)(0, n+1)\} * \{(z, (i-1) \bmod n)\} \Rightarrow \{(z, i)(1, n+1)\}, \\ \gamma &: (1, n+1) \Rightarrow (0, n+1), \end{aligned}$$

is equivalent to Θ . It also multiplies $(x_1, \dots, x_n)$ by y , but does it in $2n$ steps. Here the cell named 1 is the only processor element. It executes the microinstruction Θ' . The microinstruction δ is executed by the cells named $1, \dots, n$. They form a shift register which serves as an input and output device. A control microinstruction γ is responsible for alternating Θ' and δ applicability.

We will then create the parallel microprogram transformation reducing the cardinality of the processor names set $M_P = \{ \langle i_1, \dots, i_k \rangle \}$, $i_j = 1, \dots, n_j$, through replacing it by the set $M' = \{ \langle i_1, \dots, i_{k-1} \rangle \}$. For the sake of definiteness we are excluding the last coordinate from the corresponding to M_P array assuming M'_P to be interpreted like a hyperplane of M_P . It will be shown further that all ST-transformations of parallel microprograms, related to decreasing the size of cellular array, are reduced to this case.

Definition 2. A parallel microprogram $\Phi'(A', M')$ is called an *ST-transform* of $\Phi(A, M)$, if

- a) Φ' is equivalent to Φ ,
- b) $M'_P \subset M_P$.

We construct ST-transformations relying on the assumptions to follow.

1. One-step cell state changes performed by the original microprogram $\Phi(A, M)$ in the cells named $\langle i_1, \dots, i_k \rangle \in M$, should be

performed by the transformed microprogram $\Phi'(A', M')$ as n_k sequential steps applied to the cells named $\langle i_k, \dots, i_{k-1}, 1 \rangle \in M'_p$, $1 \in \{1, \dots, n_k\}$.

2. Cells named from $M''_p = \{\langle i_1, \dots, i_k \rangle : i_k \neq 1\}$ perform only the transport function by shifting data along the i_k -axis.

Definition 3. A cellular subarray $W(i_k=1) \subset W$ is called the l -th i_k -layer of the cellular array $W = \{(a, m) : m = \langle i_1, \dots, i_k \rangle\}$, $i_j = 1, \dots, n_j$, if $W(i_k=1) = \{(a, m) : m = \langle i_1, \dots, i_{k-1}, 1 \rangle\}$, $1 \in \{1, \dots, n_k\}$. The set of i_k -layer cell names is denoted by $M(i_k=1)$.

Definition 4. The parallel microprogram $\Phi''(A'', M'') = \{\Theta''_1, \dots, \Theta''_v\}$ is called a *projection* of $\Phi(A, M) = \{\Theta_1, \dots, \Theta_v\}$ on the l -th i_k -layer $W(i_k=1)$, if every Θ''_i has resulted from Θ_i by substituting the constant 1 for i_k and possibly by adding the cells named $m' \in M$ into its context.

Definition 5. The set of microinstructions

$$\Delta(A', M') = \{\delta_1, \dots, \delta_p, \eta_1, \dots, \eta_q\} \quad (5)$$

is called the *data shift microprogram*, if

a) δ_j are of the form

$$\delta_j = \{(a_j, \langle i_1, \dots, i_k \rangle) * \{(b_j, \langle i_1, \dots, i'_k \rangle)(a_j, \mu_j)\} \Rightarrow \{(b_j, \langle i_1, \dots, i_k \rangle)\}, \quad (6)$$

where $i'_k = (i_k + 1) \bmod n'_k$, $n'_k > n_k$, $a_j, b_j \in A'$, $\mu_j = M'$;

b) η_k are microinstructions controlling shift applicability by changing the context cell states $(\alpha_j, \mu_j) \notin A \times M$.

Definition 6. The microprogram $\Delta(A', M')$ of data shift along i_k -axis is called *compatible* with $\Phi(A, M)$, if $A \times M \subseteq A' \times M'$ and the following condition is met for every $\Theta_i \in \Phi$: if $\Theta_i(\langle i_1, \dots, i_{k-1}, j \rangle)$ is applicable to $W \in A \times M$, then $\Theta_i(\langle i_1, \dots, i_{k-1}, 1 \rangle)$, $1 > j$, is applicable to $W'_{1-j} \in A' \times M'$, where W'_{1-j} is the result of $(l-j)$ -iteration step of synchronous computation of Δ over W'_0 .

Compatibility Δ with Φ specifies the necessary condition a data shift microprogram is to ensure ST-transformation equivalence. Constructing a Φ -compatible data shift microprogram is trivial, if all naming functions are of the form $f(m) = m + d$, where d is an integer vector $(d_1, \dots, d_k)$ such that $d_k = 0$. In this case Δ contains but one microinstruction δ of the form (6) with $n'_k = n_k$, $A' = A$, $M' = M$. In a general case, when Φ contains $\phi(m) = m + d$ such that $d_k \neq 0$, compatibility

is achieved by taking $n'_k = n_k + d_k$ in (6) and also inserting control microinstructions $\{\eta_1, \dots, \eta_q\}$.

4. Correctness conditions for ST-Transformation

The main problem in constructing ST-transform of a parallel microprogram is to preserve the determinacy property of its synchronous interpretation. The first question is whether for any microprogram Φ with the determinate interpretation Φ_S there exists an ST-transform Φ' such that Φ'_S is also determinate. Since the answer is negative the next question is what conditions the microprogram is to meet for its ST-transform has the determinate synchronous interpretation. To specify these conditions we need the following definitions.

Definition 7. The computation $C(W_0)$ of $\Phi(A, M)$, $M_P = \{ \langle i_1, \dots, i_k \rangle \}$ is called i_k -ordinary, if at every step Φ is applied to only one arbitrary i_k -layer. The set of i_k -ordinary computations over all $W_0 \in A \times M$ is called i_k -ordinary interpretation of the microprogram Φ .

Definition 8. For the parallel microprogram $\Phi(A, M) = \{\Theta_1, \dots, \Theta_v\}$, $M_P = \{ \langle i_1, \dots, i_k \rangle \}$, $i_j = 1, \dots, n_j$, a configuration of the form (4) is called

a) i_k -asynchronously dangerous, if there exists a vector $d = (d_1, \dots, d_k)$, $|d_j| < n_j$, $d_k \neq 0$ and a pair $\Theta_i, \Theta_j \in \Phi$, such that

$$S'_i(m) \cap S_j^*(m+d) \neq \emptyset \quad (7)$$

b) i_k -synchronously dangerous, if there exists a vector $d = (d_1, \dots, d_k)$, $|d_j| < n_j$, $d_k = 0$, and a pair $\Theta_i, \Theta_j \in \Phi$ such that

$$S'_i(m) \cap S'_j(m+d) \neq \emptyset \quad (8)$$

Theorem 1. An i_k -ordinary interpretation of a parallel microprogram is determinate iff the two following conditions are satisfied:

a) i_k -ordinary computations $C(W_0(m))$ are determinate for all $W_0(m)$ equal to i_k -asynchronously dangerous configurations;

b) synchronous computations $C_S(W_0(m))$ are determinate for all $W_0(m)$ equal to i_k -synchronously dangerous configurations.

The proof is based on the representation of i_k -ordinary computations in the form of a composition of synchronous computations within i_k -layers and asynchronous computations between them.

Corollary. An i_k -ordinary interpretation of a parallel microprogram is determinate, if its synchronous and asynchronous interpretations are determinate.

Definition 9. A configuration $S(m)$ is called i_k -chain generated by $W(m)$, if it is the union of configurations $W(m), S_{k1}^*, \dots, S_{kl}^*$, where $S_{k1}^*, \dots, S_{kl}^*$ are left-hand sides of a maximal microinstruction subset, which has a connected graph of i_k -synchronously dangerous intersections and the intersection of $W(m)$ with at least one of S_{kj}^* is not empty.

Let $\Gamma'_S(\Theta_i)$ be a provocation graph for synchronous interpretation Ψ'_S of $\Phi(A, M(i_k=1))$ acting within an i_k -layer. It is constructed in the same way as the graph $\Gamma_S(\Theta_i)$, with the only difference that $\Gamma'_S(\Theta_i)$ vertices are i_k -chains. Let $G'_A(\Psi'_i)$ be a provocation graph for i_k -ordinary interpretation Ψ'_A of Φ . It is constructed similarly to the graph $\Gamma_A(\Theta_i)$, but the provocation of the application of a microoperation is done by forming the union of $W_j(m)$ with a left-hand side of a microinstruction which has a nonempty intersection of the form (7). Every microoperation applied to a vertex $W_j(m)$ corresponds to a step of a synchronous computation $C_S(W_j(m))$, executed over one i_k -layer.

Theorem 2. An i_k -ordinary interpretation of the parallel microprogram $\Phi = \{\Theta_1, \dots, \Theta_\nu\}$ is terminating, iff provocation graphs $\Gamma'_S(\Theta_i)$ and $\Gamma'_A(\Theta_i)$ for all $\Theta_i \in \Phi$ contain no circuits.

The proof is similar to that of Theorem 1.

Corollary. An i_k -ordinary interpretation of a parallel microprogram is terminating, if its synchronous and asynchronous interpretations are terminating.

Of practical use are also the sufficient correctness conditions for an i_k -ordinary interpretation, expressed by the

Theorem 3. An i_k -ordinary interpretation of a parallel microprogram is correct, if its synchronous interpretation is correct and asynchronous computation graphs for all i_k -dangerous words contain no circuits.

The proof is based on Theorem 2 from [7] and above Theorems.

Theorem 4. A parallel microprogram $\Phi' = \Phi \cup \Delta$, where Φ' is a projection of Φ to an i_k -layer and Δ is a microprogram of data shift along i_k -axis, is the ST-transform of a parallel microprogram Φ , iff the i_k -ordinary interpretation of Φ is correct.

The proof is based on the comparative analysis of graphs of synchronous computations of the transform Φ' and i_k -ordinary computations of the initial microprogram Φ .

It follows from Theorem 4 that only a parallel microprogram with a correct i_k -ordinary interpretation admits an ST-transformation along the i_k -axis. If this condition is not met, the microprogram should first undergo a synchronous-asynchronous transformation [6]. Clearly, whether these efforts are worth taking, should be decided in each individual case.

ST-transformation can be easily extended to the cases when the set of processor elements of the transform is assumed equal to several i_k -layers. Then an appropriate choice of these layers may help to improve the space (hardware) and time properties of a system designed.

5. Example.

The associative memory stores a data array of m words of the form $a_i = a_{i1}, \dots, a_{in}$. The word needed is selected by the key $b = b_1, \dots, b_n$, among the words stored in array rows in which the state of a cell named $\langle i, 0 \rangle$ $s_{i0} = 1$, $i = 1, \dots, m$. In the rows where the word stored coincides with the key, the cell named $\langle i, n \rangle$ takes the value $s_{in} = 1$. A parallel microprogram describes this algorithm as follows: $\Phi(A, M) = \{\Theta\}$; $A = \{[x, y]\} \cup \{z\}$, $x, y, z \in \{0, 1, -\}$, where "-" is don't care symbol, $[x, y]$ is a cell state assigned by a pair of symbols

$$\Theta: \{([-, x], \langle i, j \rangle) \} * \{(y, \langle 0, j \rangle) ([z, -], \langle i, j-1 \rangle) \} \Rightarrow \{([z * (x \cdot y \vee \bar{x} \cdot \bar{y})], x], \langle i, j \rangle) \}.$$

The parallel microprogram Φ is correct in any interpretation, hence, it allows an ST-transformation. Let us eliminate the coordinate j , having chosen $W(j=1)$, i.e. $M'_P = \{\langle i, 1 \rangle\}$, $i = 1, \dots, n$. The projection of Θ on $W(j=1)$ is

$$\Theta': \{([-, x], \langle i, 1 \rangle) \} * \{(y, \langle 0, 1 \rangle) ([z, -], \langle i, 0 \rangle) (0, m_0) \} \Rightarrow \{([z * (x \cdot y \vee \bar{x} \cdot \bar{y})], x], \langle i, 1 \rangle) \}.$$

Here m_0 is the name of a control cell introduced into the shift microprogram $\Delta(A, M')$, $M' = M \cup \{m_0\}$, to provide compatibility of Δ with Φ . $\Delta = \{\delta, \eta_1, \eta_2\}$, where

$$\begin{aligned} \delta: (1, m \{([-, -], \langle i, j \rangle) \} * \{([z, x], \langle i, j+1 \rangle) (1, m_0) \} \Rightarrow \\ \{([y, x], \langle i, j \rangle) \}; \\ \eta_1: (1, m_0) \Rightarrow (0, m_0); \\ \eta_2: (0, m_0) \Rightarrow (1, m_0). \end{aligned}$$

The resulting ST-transform $\Phi' = \Phi'' \Delta$ is a parallel microprogram of a dynamic associative memory [5] (fig 2). Note, that twice as long computation time arising in Φ' can be easily shortened by equivalently replacing the microoperation, executing the shift and the computation simultaneously.

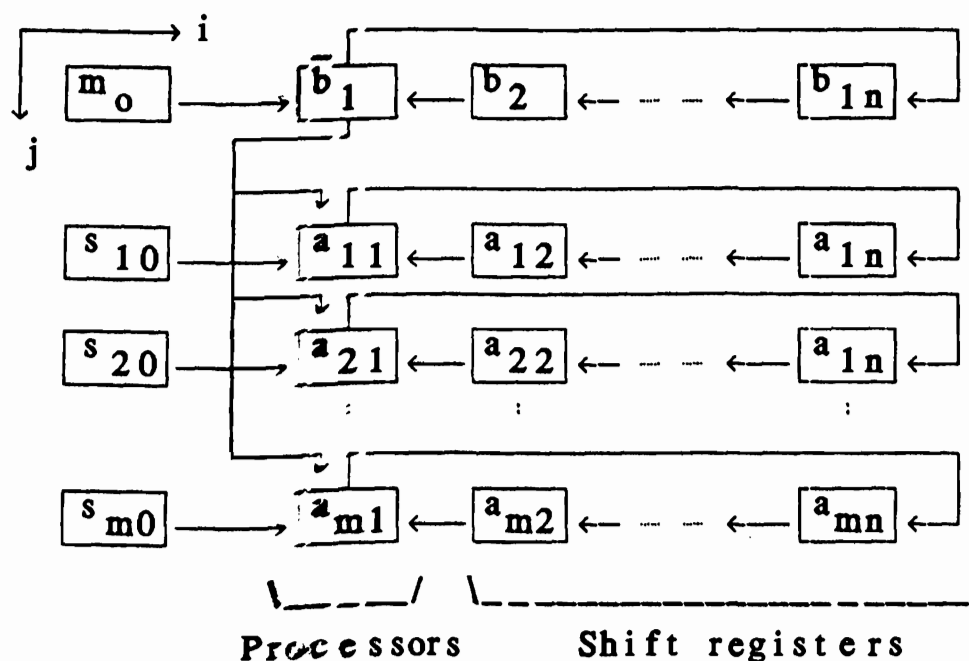


Fig. 2.

References

1. F.C.Hennie, *Arrays of Logical Circuits* (Willey, New York, 1961).
2. P.Quinton *The systematic Design of Systolic Arrays* (IRISA, Int. Rep., N 216. INRIA, 1983).
3. P.A.Anishev, S.M.Achasova, O.L.Bandman et al., *Parallel Microprogramming Methods* (Novosibirsk, Nauka, 1981).
4. O.L.Bandman, S.V.Piskunov, in: *PARCELLA'88* (Academie-Verlag, Berlin, 1988) p.57-72.
5. Ya. I. Fet, *Izvestia Akad. Nauk SSSR, Technicheskaya Kibernetika*, N 5 (1985) p.139-152.
6. O.L.Bandman, *Kibernetika*, N 1 (1980) p.42-47.
7. S.M.Achasova, O.L.Bandman, *Correctness of Parallel Computation Processes* (Novosibirsk, Nauka, 1990) 230 p

HIGH-PERFORMANCE COMPUTING SYSTEMS OF COMBINED ARCHITECTURE

A.P. Vazhenin, S.G. Sedukhin, Ya.I. Fet

Computing Center, Siberian Division, USSR Academy of Sciences,
Novosibirsk, 630090, USSR

ABSTRACT

A new type of parallel computing system architecture with a very large number of processing elements is described. The principal idea is to combine mutually supplementing useful features of two high-performance subsystems. The processing subsystem is based on a set of systolic/wavefront arrays, which provides superhigh performance in a broad range of important problems. It is suggested to implement a finegrain SIMD-system (like Goodyear's STARAN) as a preprocessing subsystem which has some inherent properties suitable for effective support of the processing subsystem. The combination is called Mixed Associative Systolic Computer (MASC). Basic features of MASC-architecture are listed. Several examples of data transformation for typical systolic algorithms are described.

1. Introduction

One of the main directions in the development of high-performance computing systems is based on the concurrent functioning of a very large number of processors. The design objective is to assure a linear (or nearly linear) rise of system performance through involving a great number of concurrently working computers. As far as possible this desired dependence should be retained for a broad set of problems.

In the rapidly growing field of design and implementation of powerful computing systems with a large number of processing elements (PE), two approaches have been recently dominating. These approaches are SIMD architectures with bit-serial processing¹ and parallel pipeline systolic/wavefront arrays of MIMD-type.² The SIMD systems are oriented mainly on effective implementation of nonnumerical problems, such as data retrieval, pattern matching and various manipulations on

data structures. On the other hand, the systolic/wavefront arrays probably comprise the fastest of the known computing means intended to solve numerical problems involving huge amounts of data. It should be noted that the effective use of systolic/wavefront arrays is strongly influenced by appropriate preparation of the form of incoming data and by the corresponding organization of the input and output of these data.

The principal idea of the present project is to combine mutually supplementing useful features of the SIMD system and systolic/wavefront MIMD system. This is a new conception of *combined architecture* consisting of two interacting subsystems: a high-speed processing subsystem presented by a set of systolic/wavefront arrays, and an intelligent preprocessing subsystem which provides preparation of all formats of needed data as well as necessary high-bandwidth input/output. An Associative Array Processor (AAP) is considered a convenient tool for the implementation of the preprocessing subsystem.

We call our combined architecture the *Mixed Associative-Systolic Computer* (MASC). Now let us describe in detail the basic features of the MASC architecture.

2. Basic features of MASC architecture

The emergence of the MASC conception was initiated by rapid and successful developments in the systolic/wavefront approach which provides a real possibility to implement MIMD architectures with extremely high performance. Obvious advantages of systolic/wavefront computations justify the suggestion of such MIMD architecture as a basis for the processing subsystem of MASC. Thus, the processing subsystem can consist of a single systolic array which may be reconfigured for solving various specific problems. Another version of the processing subsystem is a set of problem-oriented systolic arrays with relatively simple specialized hardware. In general, we propose a processing subsystem of an open type which can be expanded by such options as, for example, scalar, vector, symbol and other processors.

A good balance between the input/output operations and the processing itself is decisive for the effective functioning of the

MASC system. This balance implies input and output of data in specified formats and in proper time. For example, in systolic processing the coefficients of incoming matrices should be sent (rowwise or columnwise) to certain PE inputs of a respective array in the form of a polygon of the given shape. The sequences of coefficients must be sometimes interleaved by zeros.

If the conventional memory devices and input/output hardware are used, the realization of such elaborate data manipulations gets into serious difficulties and reduces inevitably the expected performance of the processing subsystem. The disbalance mentioned above is one of the main reasons restricting the broad spreading of systolic arrays. For instance, the designers of new systolic computers must reduce their projects to one-dimensional arrays, in which case the problems become much simpler and may be achieved by a conventional computer with serial memory access.³

In the MASC approach the necessary balance can be realized since the preprocessing subsystem is a SIMD computer with bit-serial processing possessing such inherent properties as flexible highly-parallel memory access, speed manipulation on data structures and large bandwidth of data exchange channels. The most appropriate architecture for the preprocessing subsystem is supposed to be a STARAN-like computer,^{4,5} because it possesses a set of individual items very useful for the implementation of the functions pertinent to that subsystem. These items are:

- high-level parallelism of SIMD processing;
- availability of multidimensional access memory (MDA);
- using a FLIP interconnection network, providing a large set of permutations and shifts;
- elaborate control system including a rich set of instructions for manipulations with various data types;
- high bandwidth of input/output.

A version of MASC-architecture using a STARAN-like computer as a preprocessing subsystem is shown in Fig. 1. The data communications links of Fig. 1 take into account a possible case of the direct connection between memory and the preprocessing subsystem.

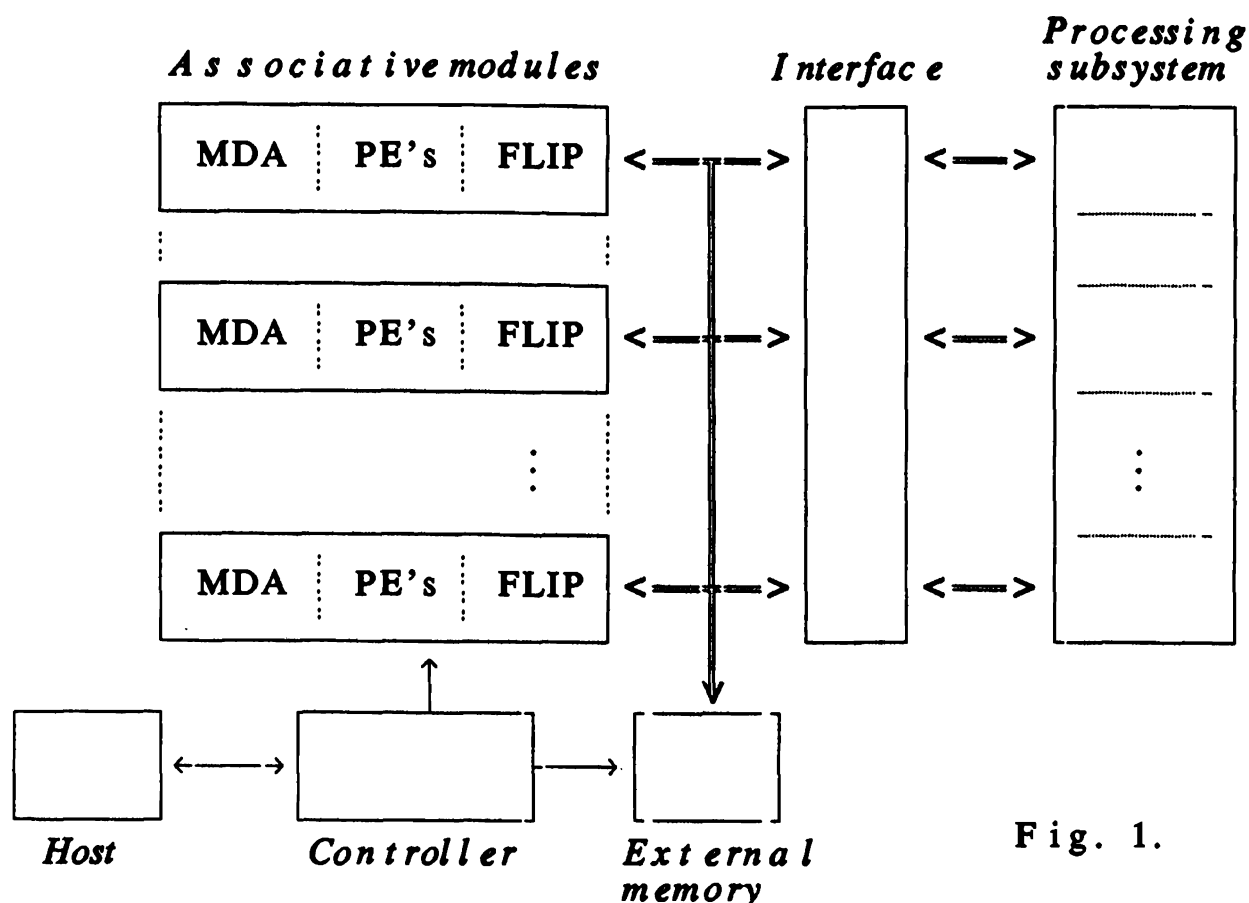


Fig. 1.

The MASC architecture removes the main complications appearing in the design of contemporary parallel computing systems with a large number of PE's.

First, recently effective specialized systolic arrays have been designed for many important practical classes of problems. For instance,⁶ contains a comprehensive list of problems with references to corresponding systolic processors. When the processing subsystem of MASC contains a single reconfigurable systolic array, various specific processors may be presented as a MASC standard systolic structures library. During the operation, the MASC supervisor executes the necessary "structure calls", and performs the customization of a reconfigurable array. At the same time the AAP enforces the procedures of data input/output, adjustment and formatting. These procedures are fulfilled in accordance with the requests of library calls for specified systolic processes.

Second, the systolic/wavefront principle gives the green light to formalized synthesis of particular MIMD structures from the initial

specifications of problems.^{7,8} Hence, the user would be discharged from the demand to describe the explicit way of partitioning the job, to program all separate parts and to point out the intertask communications. The libraries (or databases) of predesigned systolic/wavefront structures would support entirely new style of parallel program execution, based on computer-aided programming of the architecture of processing structures.

One of the complicated problems of MIMD architectures is preparing and loading of programs into a large number of the PE's. The corresponding losses can rise proportionally to the number of PE's which results in essential degradation of the overall system performance. In the MASC architecture this problem can be efficiently solved by means of :

- the relative simplicity of the functions of the PE's assuring small program size;
- the possibility of rapid duplication of these programs in the AM of the preprocessing subsystem;
- the parallel loading via high-bandwidth channels;
- the possibility of dynamic control of the PE's through information streams containing not only the data but also instructions for handling these data.

All above-mentioned possibilities are consistent with the features of the preprocessing subsystem suggested.

3. Transformation of data structures

As an example illustrating the functioning of the suggested system we consider the preparation of input data for solving the algebraic path problem, for which a set of systolic arrays has been developed by means of a systematic method.¹⁰

Suppose that the initial matrix of the order n with k -bit coefficients is stored rowwise in the channels of the memory of STARAN-like SIMD-system (see Fig. 2, where $n = 4$).

One of systolic algorithms solving the mentioned problem requires the incoming data in the form shown in Fig. 3. The corresponding

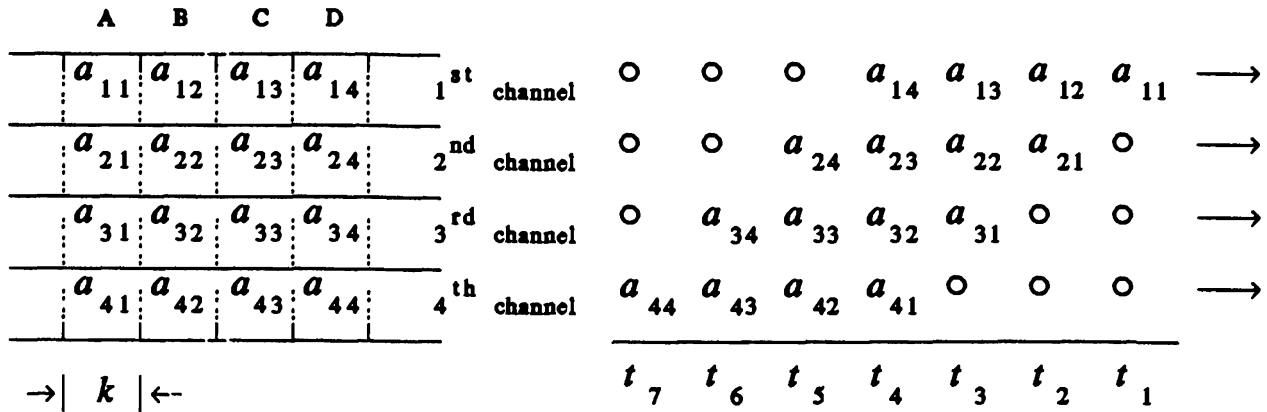


Fig. 2.

Fig. 3.

preprocessing is made in the memory of the SIMD-system by means of regular row shifts of the initial matrix: the i -th row ($i = 1, \dots, n$) is shifted in the i -th channel on $k(i-1)$ bits to the right relative to its primary position. The implementation of this procedure in STARAN requires only n read-write cycles or $2n$ time steps (in the presented mode of the matrix storage $n \leq 256$ for one associative module). The result of preprocessing is shown in Fig. 4. As seen from figure, the necessary volume of memory is $nk(2n-1)$ bits.

After the described preprocessing we can proceed to read the data from the memory and transmit them to the processing subsystem. The reading is accomplished by means of standard SIMD-system's instructions using necessary masks. The set of masks for the present example is given in Fig. 5. The time of transmission is negligible because the data readout and the processing in the systolic array make up a single pipeline.

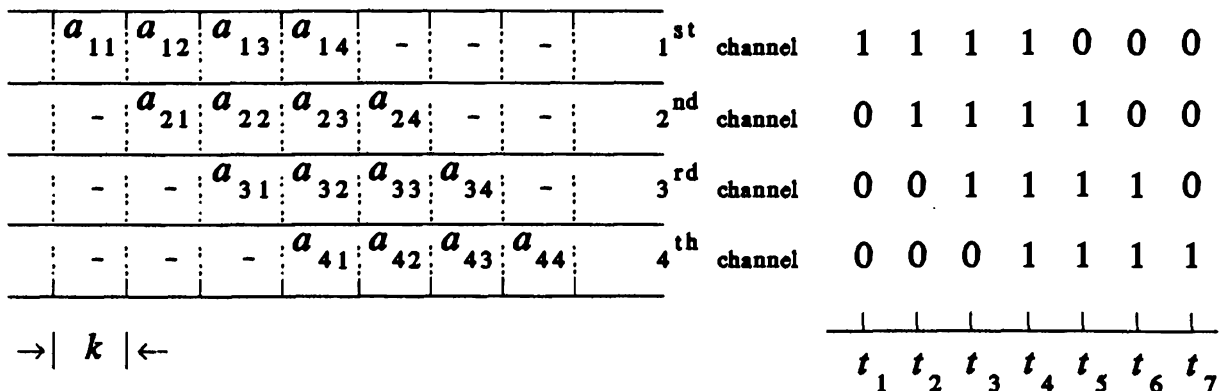


Fig. 4.

Fig. 5.

Now consider another example. For the same algebraic path problem a different systolic array requires incoming data in the form of Fig. 6 (as before a matrix of order 4 is considered). The strategy used in the foregoing example is to form the needed data structure in the working space of the memory in its final shape (in the first example it was a dense parallelogram). This way may be followed again, but in the second example the needed structure is more complicated (a sparse rhombus). As can be shown, forming of such a structure while using STARAN's standard instructions requires an order of n^2 time steps.

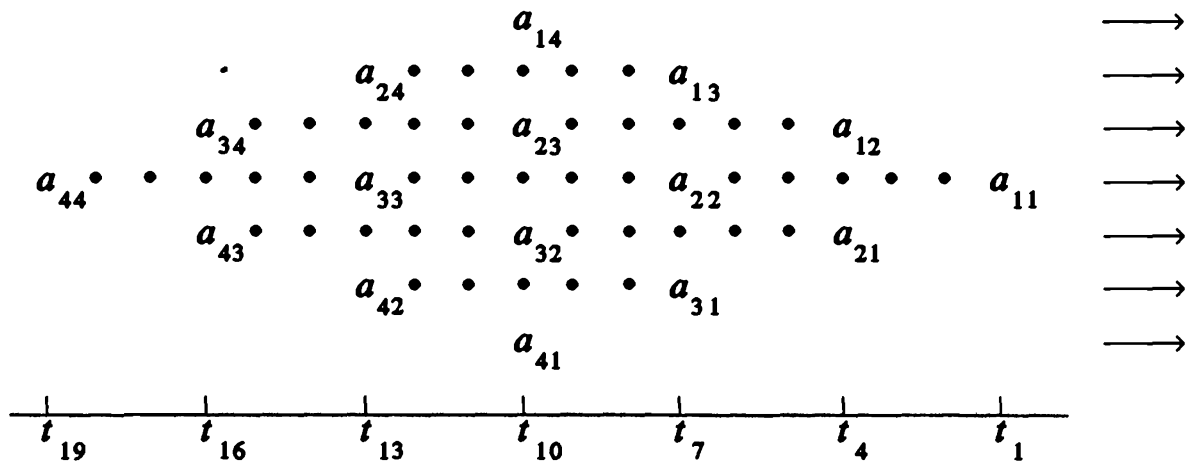


Fig. 6.

To speed up the preprocessing we take advantage of the fact that the "columns" of the needed rhombus are regularly thinned and therefore can be formed and stored in memory in a merged manner. For example, the columns $(a_{12} \cdot a_{21})^T$ and $(a_{13} \cdot a_{22} \cdot a_{31})^T$ of the Fig. 6 can be stored as $(a_{13} a_{12} a_{22} a_{21} a_{31})^T$ and fetched from the same address but with different masks. In the present case the masks $(01010)^T$ and $(10101)^T$ should be used. The data preparation in this approach can be made for the same example in three phases:

1). Regular column shifts of the initial matrix (Fig. 7a): the j -th column ($j = 1, \dots, n$) is shifted on $(j-1)$ channels upwards relative to its primary position (thus forming the 1-st intermediate structure - parallelogram of Fig. 7b). The first phase requires an order of $2n$ time steps and an order of $2kn^2$ bits of memory.

2). Regular shifts of the upper rows of the 1-st intermediate structure: the i -th row ($i = 1, \dots, n-1$) is shifted on $k(n-1)$ bits to the left (2-d intermediate structure - triangle of Fig. 7c). The expense of time and memory the same as in the first phase.

3). "Stepped" shift of the 2-d intermediate structure, i.e. forming of the resulting rhombus with merged columns: i -th row of the triangular intermediate structure is shifted on kt bits to the right, where

$$t = \begin{cases} s/2, & \text{if } s \text{ is even,} \\ (s+1)/2, & \text{if } s \text{ is odd;} \end{cases} \quad s = |i-n|.$$

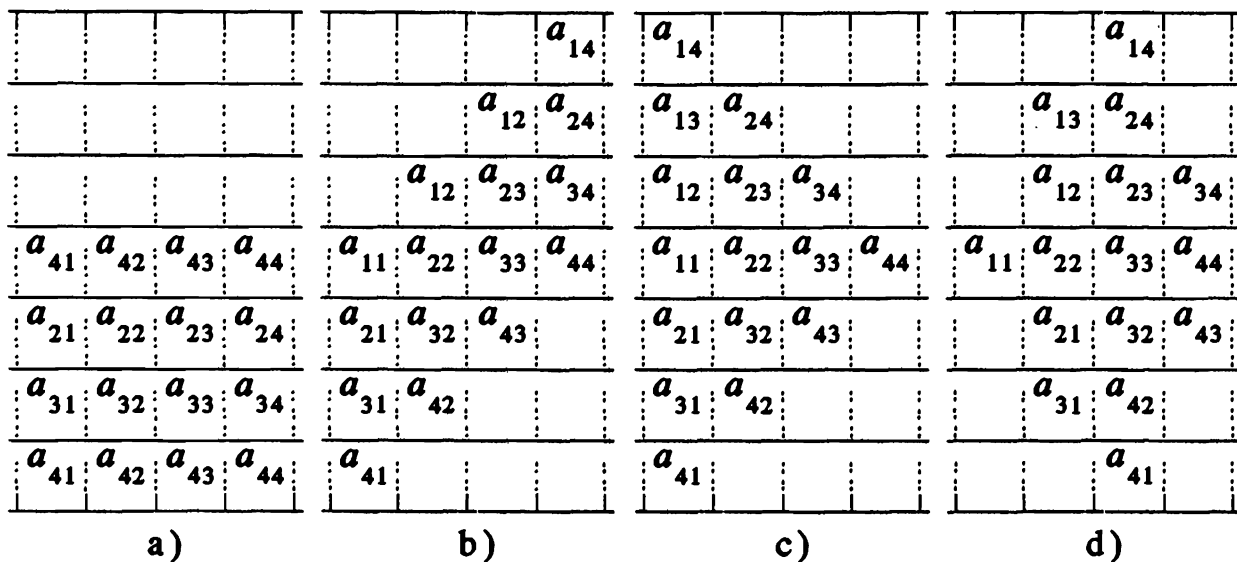


Fig. 7.

This phase requires an order of $4n$ time steps, memory as previous. The resulting structure for the present example is shown in Fig. 7d. The total time for all three phases is an order of $8n$ time steps. The subsequent reading is accomplished by standard instructions using the set of masks given in Fig. 8.

The performance of the system can be improved if some additional hardware is introduced.

One important improvement of finegrain SIMD-systems is the local address modification,¹¹ which ensures independent addressing of the local memory of each PE instead of the common global address used in most known systems. In presence of local addressing, implementation of

the described data preparation procedures does not require row shifts. This function can be replaced by corresponding addressing carried out simultaneously with the reading.

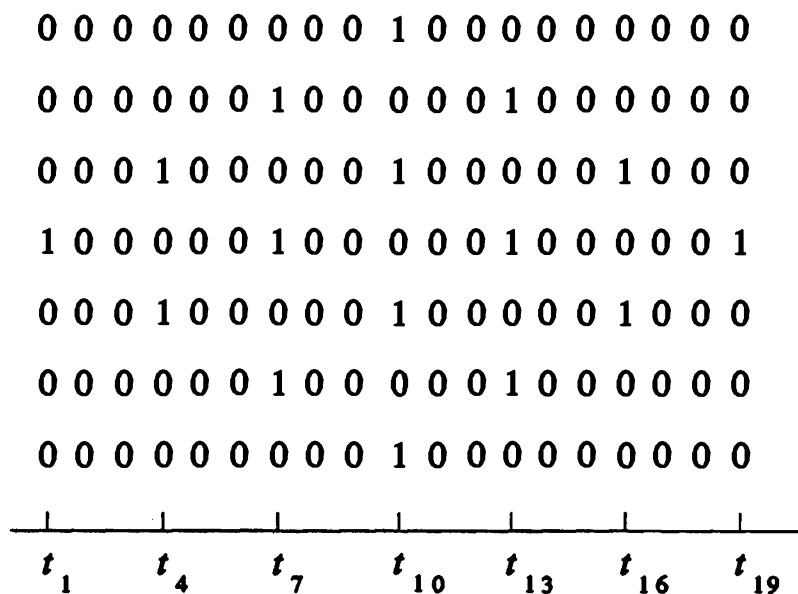


Fig. 8.

For the first of described examples our procedure looks like this. Before reading each successive field of initial matrix (A,B,C,D in Fig. 2) the corresponding column of addresses from the auxiliary address table (Fig. 9) is placed into the local address registers, which ensures "skewed" reading. At the same time the corresponding mask (see Fig. 5) is placed into the mask register, which enables selection of the necessary channels. So, preparing of the given data format is fulfilled "on-the-fly", without additional time expense.

A	B	C	D	-	-	-
-	A	B	C	D	-	-
-	-	A	B	C	D	-
-	-	-	A	B	C	D
t_1	t_2	t_3	t_4	t_5	t_6	t_7

Fig. 9.

Another possible improvement of the system is associated with the implementation of specialized interconnection networks. Thus, it is usefull to introduce a compressing/expanding unit similar to the λ -network.¹²

Let us call *compressing permutation* a permutation which corresponds to the connection of arbitrary ordered M -tuple of input channels (from the total number N) with M adjacent output channels; and *expanding permutation* a permutation which corresponds to the connection of M adjacent input channels with an arbitrary ordered M -tuple of output channels (from the total number N); in both cases the relative order of channels is retained. The λ -network is an (N,N) - pole implementing compressing and expanding permutations when the corresponding binary control vectors are fed to its control inputs.

If the system is equipped with such a network, most transformations executed by means of shifting columns and rows in the described implementation of our second example, can be replaced by simple channel permutations while reading. In the considered example four inputs (according to the number of rows of the input matrix) and seven outputs (according to the number of rows of the required rhombus) of the λ -network are used. The λ -inputs are connected to the corresponding outputs of SIMD channels, and the λ -outputs with the corresponding inputs of the processing subsystem.

Now for the data preparation it is sufficient to make a regular row shift resulting in a "skewed" matrix (Fig. 4). Then one can start reading. Before reading each successive column of the "skewed" matrix, corresponding binary control vector from an auxiliary control table (which coincides in this case with the mask table Fig. 8) should be fed to control inputs of the λ -network. This expanding ensures the required vertical "stretching" of matrix coefficients. At the same time the corresponding masks from the mask table Fig. 10 should be put into the mask register.

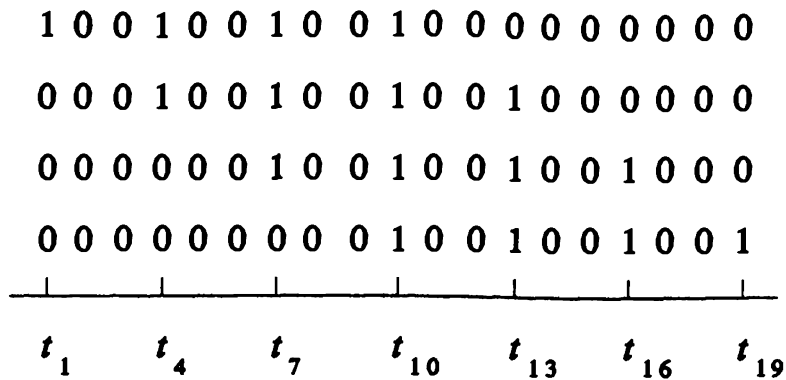


Fig. 10.

It is clear that the joint use of both mentioned mechanisms guarantees fetching of columns in accordance with the necessary rhombus structure.

4. Conclusion

The proposed MASC architecture possesses many useful features assuring efficient work of a computing system with a large number of PE's. These features are:

- solution of any problem is considered as a communication of two processes: data preparing and processing itself. In the MASC architecture the realization of these processes is delegated to two specialized subsystems, each being oriented to the effective implementation of the corresponding process;
- each PE of the processing subsystem comprises a relatively simple computing unit which simplifies the control task, executed by the preprocessing subsystem. Moreover, this subsystem can realize some diagnostics functions for the processing subsystem, and organize fault-tolerant and high-accuracy computations;⁹
- the MASC architecture enhances the possibilities of using high-level problem-oriented programming languages by means of direct mapping of algorithms onto the structures of processing units.

The authors hope that the MASC architecture which combines the unique features of AAP with exclusive possibilities of systolic/wavefront arrays can assure superhigh performance in solving a sufficiently broad range of important problems.

References

1. *Batcher K.E.* Bit-serial Parallel Processing Systems // IEEE Trans. on Comput. - 1982. - Vol. C-31, N 5. - P. 377-384.
2. *Kung H.T.* Why Systolic Architectures? // Computer. - 1983. - Vol. 15, N 1. - P. 37-46
3. *Annaratone M. et al.* Warp Architecture and Implementation // Proc. 13th Int. Symp. Computer Architecture, June 1986. - Computer Society Press, Silver Spring, MD. - 1986. - P. 346-356.
4. *Batcher K.E.* The Multidimensional Access Memory in STARAN // IEEE Trans. on Comput. - 1977. - Vol. C-26, N 2. - P. 174-177.
5. *Batcher K.E.* STARAN Series E // Proc. Int. Conf. Parallel Processing. New York: IEEE. - 1977. - P. 140-143.
6. *Fortes J.A.B., Wah B.W.* Systolic Arrays - from Concept to Implementation // Computer. - 1987. - Vol. 20, N 7. - P. 12-17.
7. *Lengauer C.* On the Projection Problem in Systolic Design. // Carnegie Mellon Univ., Computer Science Department, Preprint CMU-CS-88-102, 1988.
8. *Sedukhin S.G., Trishina E.V.* Automated Procedure for Synthesis of Systolic and Wavefront Arrays // CONPAR 88, Eds. C.R. Jesshope and K.D.Reinartz. Cambridge University Press, British Informatics Society Ltd. - 1989. - P. 735-742.
9. *Vazhenin A.P.* Programming System of High Accuracy Computations for Associative Array Processor // CONPAR 90 - VAPP IV, Volume of Special Technical Contributions, Zürich. - 1990. - P. C69-C72.
10. *Sedukhin S.G.* Design and Analysis of Systolic Algorithms for Solving the Algebraic Path Problem. - Novosibirsk, 1987. - 44 p. - (Preprint USSR Acad. of Sciences, Sib. Div., Comp. Center; N 744). (In Russian)
11. *Blevins D.W. et al.* BLITZEN: A Highly Integrated Massively Parallel Machine // Journ. of Paral. and Distrib. Comp. - 1990. V. 8, N 2, P. 150-160.
12. *Fet Ya.I.* Parallel Processors in Control Systems. - Moscow: Energoizdat. - 1981. (In Russian)

LANGUAGE STAR FOR ASSOCIATIVE AND PARALLEL COMPUTATION WITH VERTICAL DATA PROCESSING

A. Sh. NEPOMNIASCHAYA

*Computing Center, Siberian Division,
USSR Academy of Sciences,
Novosibirsk 630090, pr. Akad. Lavrentieva,6*

ABSTRACT

This paper describes the language STAR for algorithms oriented towards associative (or content-addressable) bit-serial parallel processors. STAR is based on Pascal and includes new data types and operations which allow the easy writing of algorithms for an associative parallel processor like STARAN.

1. Introduction

The architecture of the associative parallel processor (APP) is well suited for different applications, such as relational data bases, matrix computations, solution of graph theoretic problems and problems which use set operations ¹⁻³.

Most high-level languages which have been proposed for APPs (Glypnir, Parallel Pascal, Pascal/L) are directed towards a specific machine and take into account its architecture.

STAR uses the STARAN-based model ^{2,4} which consists of a control unit (CU) and of one to thirty-two associative processor modules (APM). The CU decodes instructions and causes all APMs simultaneously to execute them. An array consists of 256 words by 256 bits. APM may involve one or several arrays. The memory words are 256 bits long (may be extended up to 9 kbits). Outside the memory there are three registers of the same size as memory words. Notice that a 256 bit word or a 256 bit slice can be accessed equally easy.

In order to describe the parallel run of different APMs, STAR

uses some constructions from Pascal/L ⁴.

On the one hand, the language STAR includes a set of constructions for writing algorithms which will be realized on APP with vertical processing. On the other hand, its operations, functions and predicates are simpler than operations of a real machine. STAR can be used as a language for publications of different algorithms for associative parallel processors with vertical data processing.

2. Kernel STAR

In this section we shall describe STAR constructions for the case when APP has one module. Variables of following data types will be used: *integer*, *boolean*, *word*, *slice*, *table* and *array*. Data type will be introduced as in Pascal. We shall use an ordered sequence of symbols zero and one enclosed within single quotation marks (apostrophes) as constants for data types *word* or *slice* (for example, '011001001'). Let T be a variable of the type *table*. Then T is a structure of a fixed number of components which all are of the same type *slice*. Let M be a variable of the type *array*. Then M is a structure of a fixed number of components which all are of the same type *integer*. Concerning the types *table* and *slice* it should be noted that variable T is associated with a matrix T of k columns and r rows, where $k \leq 256$. If $r > 256$, then the matrix T is divided into parts of 256 rows each. If $r < 256$ or the last part of T has less than 256 rows, we write down new $256 - r$ rows of symbols ω after the last row of T. That is why we can assume that any variable of the type *slice* consists of 256 components which belong to $\{ 0, 1, \omega \}$.

2.1. Operations with Slices

The description of APP is taken from ²⁻⁶. Let Y be a variable of the type *slice*. Consider that ONE-component, ZERO-component, ω -component of Y denotes a component of Y with the value ONE, ZERO and ω , respectively. Define the following operations.

CLR(Y) sets all components of Y to ZERO.

SET(Y) sets all components of Y to ONE.

MASK(Y,i..j) sets i-th through j-th components inclusive to ONE, others sets to ZERO ($i < j$).

Y(i) selects the i-th component of Y.

STEP(Y) resets the first ONE-component of Y. If slice Y has no ONE-components it does not change.

NUMB(Y) yields the number k of all ONE-components, $k \geq 0$.

FND(Y) yields the ordinal number k of the first ONE-component of Y, $k \geq 0$.

DIS(Y) yields the disjunction of all components of Y different from ω -component.

Let X and Y be variables of the type *slice*. We define the following operation.

PRESS(X,Y) erases from X all its components which are matched with the corresponding ZERO-components of Y, and then compresses the contents of X. The contents of X does not change if slice Y has no components with value ZERO.

Let us explain the semantics of this operation. Let Y have k ZERO-components ($k \geq 1$) and X have r ω -components ($r \geq 0$). Then after the execution of PRESS(X,Y) the last $k+r$ components of X will consist of symbols ' ω '.

Before defining bit-serial operations for slices X and Y, we consider that $\omega\sigma\gamma = \gamma\sigma\omega = \omega$, where $\gamma \in \{0,1,\omega\}$, σ is any operation which will be defined below.

The following bit-serial operations are executed simultaneously by all corresponding components of X and Y. These logical operations are introduced in the obvious way.

$X \wedge Y$ is conjunction, $X \vee Y$ is disjunction, $\neg X$ is negation. Other operations are constructed from these operations by means of superposition: $X \Rightarrow Y$ is implication, $X \sim Y$ is equivalence, $((\neg X) \wedge Y) \vee (X \wedge (\neg Y))$ is exclusive 'OR' (which will be written as $X \oplus Y$), $X \wedge (\neg Y)$, $X \vee (\neg Y)$, $(\neg X) \wedge Y$, $(\neg X) \wedge (\neg Y)$, $(\neg X) \vee (\neg Y)$.

2.2. Predicates and Standard Functions for Slices

Let us define the following three predicates.

ONE(Y) yields *true* if and only if slice Y has no components with value ZERO.

ZERO(Y) yields *true* if and only if slice Y has no components with value ONE.

SOME(Y) yields *true* if and only if slice Y has at least one component with value ONE.

Let X be a variable of the type *slice* and j,k,r be variables of the type *integer*. We shall define the following functions which transform the contents of X or any group A from r components of X, where $r=2^j$, $1 \leq j \leq 8$.

Shift(X,k,r) moves the contents in each group A of the slice X in the following way. The initial part of r-k components of A is moved on k positions, ZERO-components are set from the first through its k-th positions, inclusive, and the last k components of the group A are shifted out at its edge.

Rotate(X,k,r) performs the circular shift of the contents in each group A. This function is similar to the function shift(X,k,r) except that the components which are shifted out at one edge of the group A are shifted in at the opposite edge of the same group.

Shuffle(X,r) performs the transpose of contents in each group A dividing all its components into two equal parts and placing components of the lower part upon the components of the upper part.

Mirror(X,r) performs the reverse of contents in each group A.

Notice that all considered above functions except Shift(X,k,r) are only used when the group from r components of X has no ω -components.

2.3. Operations with Strings

Let w be a variable of the type *word*. The following operations are defined.

w yields the length of w (# w $\leq$ 256).

w(i) selects the i-th component of w .

Mirror(w) yields the reverse of w .

The operations $FND(w)$, $NUMB(w)$, $STEP(w)$ and predicates $ONE(w)$, $ZERO(w)$, $SOME(w)$ are introduced in the same way as they are for slices.

Let w and v be variables of the type *word* and have the same length. We can introduce bit-serial logical operations for them analogous to the operations for slices discussed above.

2.4. Operations with Matrices

Let T be a variable of the type *table*, Y be a variable of the type *slice* and i, j, k be variables of the type *integer*. The matrix operations are the following:

$T(i)$ yields the i -th row in the matrix T .

$Col(j, T)$ yields the j -th column in the matrix T .

$T[k]$ yields the k -th part of matrix T . This operation is used if the matrix T consists of more than 256 rows.

Y with T (respectively T with Y) adds to T left (respectively right) the contents of the slice Y . Let us explain the semantics of this operation. If the last r rows ($r \geq 0$) of T consist of symbols ' ω ', then the last r components of Y are ω -components.

$PRESS(T, Y)$ deletes from matrix T all rows which are matched with the corresponding $ZERO$ -components of Y . Then the matrix T is compressed. This operation does not change T if the slice Y has no $ZERO$ -components.

$PRESS(T, w)$ deletes from T all columns which are matched with the corresponding positions of symbols ' 0 ' in w . For example, if $w = '1011101'$, then the second and sixth columns will be deleted from T .

2.5. Standard Functions for Matrices

Let T be a variable of the type *table* and i be a variable of the type *integer*. The matrix functions are the following.

Function $size(T)$ yields the number of columns in T .

Function $part(T)$ yields the number of parts in the partition of T .

Function $row(T[i])$ yields the number of rows in the i -th part of T which do not consist of symbols ' ω '. If $part(T) = 1$, then suppose

that $T[1] = T$ and $\text{row}(T[1]) = \text{row}(T)$.

Function $\text{incr}(T)$ adds a new part to the end of the matrix T .

Function $\text{dis}(T)$ yields a slice Z in which each component $Z(i)$ is a disjunction of all components of $T(i)$. It is obvious that $Z(i) = '1'$ if $T(i)$ has at least one symbol '1' and $Z(i) = '\omega'$ if $T(i)$ consists of symbols ' ω '.

Note that operation $\text{PRESS}(T, Y)$ and the function $\text{incr}(T)$ will change values of the functions $\text{row}(T)$ and $\text{part}(T)$, respectively. The operations T with Y and $\text{PRESS}(T, w)$ will change the value of $\text{size}(T)$.

2.6. Statements

Each program has two parts: header and block. The program header specifies its identifier name and the formal parameter identifiers. The block contains the declaration part and the statement part. The declaration part comprises the following parts, any of which can be omitted: the label part, the constant definition part, the type definition part, the variable declaration part, the procedure and function part. Note that statements resemble these of Pascal⁷.

Let $\langle I \rangle$ be a variable identifier, $\langle BE \rangle$ be a boolean expression, $\langle E \rangle$ be an expression of the integer type, $\langle S \rangle$ be a statement or a compound statement.

Language STAR uses the following statements:

- assignment statement;
- goto statement: *goto* $\langle \text{label} \rangle$;
- conditional statements of forms: *if* $\langle BE \rangle$ *then* $\langle S \rangle$;
if $\langle BE \rangle$ *then* $\langle S_1 \rangle$ *else* $\langle S_2 \rangle$;
- repeat statements: *repeat* $\langle S_1 \rangle \{ ; \langle S_2 \rangle \}$ *until* $\langle BE \rangle$;
- for* $\langle I \rangle := \langle E_1 \rangle$ *to* $\langle E_2 \rangle$ *do* $\langle S \rangle$;
- for* $\langle I \rangle := \langle E_1 \rangle$ *downto* $\langle E_2 \rangle$ *do* $\langle S \rangle$;
- procedure statement;
- compound statement.

It should be noted that the current value of a variable in the left-hand side of the assignment statement may be of the type *slice* or *word*. But the variable (or the function) from the left-hand side of this statement and its right-hand side must be of the same type.

3. Examples

Let us consider the following procedures, which are written in STAR.

Example 1.

```

proc MATCH(T: table; w: word; var Z: slice);
* Testing w for membership in the matrix T *
var i,k: integer; X,Y: slice;
begin k:= size(T); SET(Y);
  for i:= 1 to k do
    begin X:= col(i,T); if w(i)='1' then Y:=Y^X
      else Y:=Y^(¬X)
    end;
  Z:=Y
end;
```

A short explanation may be appropriate here.

This procedure yields a slice Z in which $Z(i)='1'$ if and only if $T(i)=w$. For this purpose we use a slice Y in which the initial values of all components are set to ONE. At the i -th step of the algorithm we remember the rows of T which have the first i symbols of w as their initial part. We can do it by means of ONE-components of Y .

Example 2.

```

proc CHANGE(P: table; var K: table);
* Elimination of repeated rows of T *
label 1,2; var i,j,r: integer; w: word; X,Y,Z: slice;
begin i:=1; j:=1; CLR(Z);
1: w:= P(i); K(j):= w; MATCH(P,w,Y); Z:= Z^Y;
  X:= Z; i:= FND(X); if i = 0 then goto 2 else
    begin j:= j+1; goto 1
  end;
2: begin size(K):= size(P); row(K):= j; part(K):= part(P)
  end;
end;
```


This procedure uses the previous procedure MATCH. To receive an effective algorithm, we accumulate all positions of repeated rows in the slice Z each time the procedure MATCH has been used.

4. References

1. T.Kohonen, *Content-Addressable Memories*. (Berlin: Springer-Verlag, 1980).
2. C.C.Foster, *Content-Addressable Parallel Processors*. (Van Nostrand Reinhold Company, New York, 1976).
3. Ya.I.Fet, *Parallel Processors in Control Systems*. (Moscow: Energoizdat, 1981).
4. C.Fernstrom, J.Kruzela, B.Svensson, *LUCAS Associative Array Processor. Design, Programming and Application Studies*. (Lecture Notes in Computer Science, Berlin: Springer-Verlag, vol. 216, 1986).
5. A.D.Falkoff, in *Journ. of the ACM*, vol.9, no.10 (1962) p.488.
6. K.Richter, in *Intern. Conf.: Control Systems of Robots* (Elsevier Science Publishers B.V., North-Holland, 1984) p.309.
7. K.Jensen, N.Wirth, *PASCAL User Manual and Report*. (Berlin: Springer-Verlag, 1978).

The Kernel of a Graphic Environment for Analyzing Distributed Systems

Pierpaolo Degano,[§] Roberto Gorrieri, Luca Zamboni and Pierluigi Zanotti

Dipartimento di Informatica, Università di Pisa, Corso Italia 40 — I-56125 Pisa

e_mail address (uucp): {degano,gorrieri}@dipisa.di.unipi.it

[§] also Dipartimento di Matematica, Università di Parma, Via d'Azeglio 85 — I-43100 Parma

ABSTRACT

The first results of an ongoing project are presented, about the systematic development of environments and automated tools for the analysis and the specification of concurrent systems in the framework of true concurrency. These first outcomes consist of the kernel of an integrated environment for a specification metalanguage and of a tool for extending it to cope with process description languages such as CCS³⁵. The used graphic metalanguage is *Concurrent Histories*²², a formalism similar to Petri Nets, which explicitly describes the causal relation among the events performed by processes, and among processes themselves. The state of a concurrent system is described as the set of the states of its sequential processes, and the dynamics of the whole system is specified by a set of so-called *atomic histories* representing the elementary steps of the sequential processes; then, a graph replacement operation is used as a computation step. This metalanguage is provided with a syntax-directed graphic editor which supports the user in editing the set of atomic histories, and with a graphic interpreter for running specifications, equipped with some helps for debugging. An interface has been developed, which, given a process description language and its operational semantics in a variant of Plotkin's SOS style¹⁶, produces its abstract interpreter in terms of the metalanguage.

1. Introduction

The main research topic in software engineering is the definition of new techniques, new higher level languages, new programming environments in order to improve both production and quality of software products. This result can be considered partly achieved in the case of sequential systems, where the definition of new high level languages based upon a well-founded mathematical theory has led, and is leading to the realization, and sometimes to the mechanical generation of automatic tools supporting system development. These achievements are supposed to increase by an order of magnitude the efficiency and effectiveness of programmers and designers of sequential systems. The situation in the case of concurrent distributed systems is very different, mainly because the present understanding of their nature is still rather limited. Certainly, practical experience of such systems is not very deep, since truly parallel systems have not been available until recently. But, it is more crucial the lack of a conceptual and linguistic framework for investigating, understanding and assimilating the wide variety of phenomena typical of concurrent systems, such as deadlock, starvation, race conditions, etc. As a consequence, most of the software engineering activities taking place in this area are conducted on empirical grounds — often without clean mathematical definitions and constructions corresponding to the basic concepts. While the theory of automata and formal languages is behind syntax analyzers,

Research partially supported by ESPRIT Basic Research Action 3011, CEDISYS, and by Progetto Finalizzato Informatica e Calcolo Parallelo, obiettivo LAMBRUSCO.

and the theory of λ -calculus describes sequential programming languages, no comprehensive and agreed upon theory is known for concurrent distributed systems.

To our opinion, the basic primitive concepts characterizing any concurrent language should include, at least, sequentiality, nondeterminism, synchronisation and concurrency. However, there is no general agreement about this set of intuitive semantic operations: in most of the proposed theories, concurrency is not considered as primitive, but expressible in terms of nondeterminism and sequentiality. This issue is at the basis of the debate between the advocates of the so-called *interleaving* approach^{5,6,35,36,33} and of the *true concurrent* one^{2,7,22,14,28,37, 45,46}. In the interleaving approach, a system run is represented as a total ordering among the possibly spatially separated and causally independent events which are performed, and concurrency is expressed by the fact that concurrent events can occur in any order; implicitly, a global clock is assumed: thus the total ordering represents also the temporal ordering. In the true concurrent approach, a partial ordering of events represents a run of a system, and concurrency is expressed by the fact that concurrent events are not related. Consequently, there is no global clock and time can be seen flowing only when considering causally related events. Absence of ordering means that concurrent events may occur or in either temporal ordering or *simultaneously*.

We firmly stay on the partial ordering side because, even if its theory is not completely satisfactory yet, it offers a definitely better, always closer, and often simpler description of reality than the other approach; see⁴¹ for an interesting discussion about this issue. Another advantage of the true concurrent approach is that a formalism in it has naturally associated a much more expressive graphic representation, since partial orderings are depicted in a two-dimensional space. Indeed, a graphic formulation is usually preferred to a linear one, for its immediacy. Indeed, the success of Petri Nets²⁸, defined as a set of places, a set of transitions and a flow relation connecting places and transitions, is probably due also to the nice pictorial representation as a two-sorted graph; similar arguments stand behind the design and implementation of Statecharts³². Thus, a visual interface is needed in order to fully exploit the expressiveness of the truly concurrent models. As a matter of fact, visual tools give a more intuitive representation of distributed programs, and thus their development is made easier. Many proposals on visual formalism and tools for the design and simulation of distributed systems can be found in the literature, and we refer the interested reader to³⁴, which however are mainly driven by heuristic considerations.

Although not definite, the theory in the interleaving approach is rich enough to provide the formal basis to a first design and implementation of tools supporting specification and verification of concurrent systems, (e.g.^{9,29,30}). This is not yet the case for truly concurrent approaches (at least for tools supporting concurrent languages) and only remarkable but restricted exceptions can be found in the literature, particularly in the framework of Petri Nets and Statecharts.

The tools developed in the Petri Net area mainly consist of editors supporting drawing nets, of interpreters for playing the token game, and of some specialised analysers (e.g.,^{24,3,8}); no attention however is paid to any linguistic level. Statecharts³², a state/event formalism devised to specification and design of real-time systems, are a good example of successful synergy of theory and practice: from the very beginning they have been equipped with visual tools for defining and running specifications; again little attention is paid to supporting concurrent languages.

We aim at systematically developing computer aided tools, with a formal basis, that support the specification, design and verification of concurrent systems for process description languages within the truly concurrent approach.

1.1. The Project Leonardo

The ongoing project *Leonardo* at the University of Pisa is concerned with the generation of interactive environments that support the specification, design and verification of concurrent systems. As for programming environments for sequential languages, the basic idea of *Leonardo* is that each process description language could be mechanically equipped with an integrated environment offering common functionalities and user interfaces, starting from a formal description of both its syntax and semantics. Our goal is then moving towards a *semantic-driven generation* of programming environments. To this aim, we think useful introducing a basic metalanguage consisting of (hopefully) all the elementary concepts common to concurrent languages, designing an integrated environment for the metalanguage and then building a tool for describing a process description language in terms of the metalanguage. Another key feature of *Leonardo* comes from having been conceived within an entirely visual framework, thus providing its users with expressive and powerful graphic representation of both the statics and the dynamics of system specifications — the future is image.

The tools supplied for specifying concurrent systems are developed within a formally well-founded framework. First, an interactive graphic editor for the metalanguage creates metalanguage specifications; then a graphic interpreter executes them. In particular, the interpreter transforms a specification in a prototype, hence it permits to gain all the advantages of early prototyping, such as experimenting on the specifications themselves. Indeed, late detection of errors in this crucial phase would cause high costs due to a fall of modifications onto all the other life-cycle phases. Also, early prototyping permits experimentation on a process description language in design, enabling a finer tuning of its operators.

Another feature of the proposed environment is top-down development, where abstract specification components can be detailed stepwise towards the “actual” prototype. Other essential features are parameterisation, abstraction and modularity which would enable the user to compose portions of the specification. Finally, some semantic-based verification techniques are planned to be included, such as various forms of equivalence and model checking.

The paper describes the first outcomes of the *Leonardo* project, i.e. the graphic interpreter of the chosen metalanguage and the tools for mapping process description languages, such as CCS, CSP, Occam and the similar, into the metalanguage.

1.2. The metalanguage: Concurrent Histories

The graphic metalanguage used here is *Concurrent Histories*^{20,21,22} (CH for short), by which it is possible to describe the causal dependencies among the processes composing a system and the events they generate. Intuitively, a concurrent history is a partial ordering of labelled events and (states of) processes, which have a suggestive graphic shape. Processes can be only minimal or maximal in the partial ordering, with the intuition that the minimals, called *heads*, represent the initial state of the whole system, the maximals, called *tails*, its present state.

Concurrent Histories can be grown by concatenating them through a nondeterministic operation which respects causality. The concatenation operation *_before_gives_* acts between two concurrent histories, and considers the set of tail processes of the former as the starting point of the head processes of the latter. Thus, these two sets play the rôle of gluing points. In drawing a CH, the partial ordering is represented by its Hasse diagram, growing downwards and processes (events) are depicted as rectangles (circles). The CH in Fig. 1.a describes a system whose initial state is composed of an instance of process A and an instance of process B; the events performed are labelled by x and y, where x represents a synchronization of the two processes, and y an action performed by process B in isolation, concurrently with x. Then, the present state consists of two instances of process A, one of which is caused by x, the other by y which also causes the instance of process B. The CH in Fig. 1.b describes a system where the synchronization of two processes, one in state A and the other in state B, originates the event labelled x without changing the state of the processes (the order of the process states in the graphic representation is irrelevant). The CH in Fig. 1.c is obtained through the *_before_gives_* operation by matching the two heads of Fig. 1.b with the two leftmost tails of Fig. 1.a. Note that the concatenation does not involve the rightmost A-process, which is then allowed to proceed on its own asynchronously. Note that another matching is possible, by considering the two rightmost tails of Fig. 1.a. Fig. 1.d shows the concatenation of the CH of Fig. 1.a to that of Fig. 1.c.

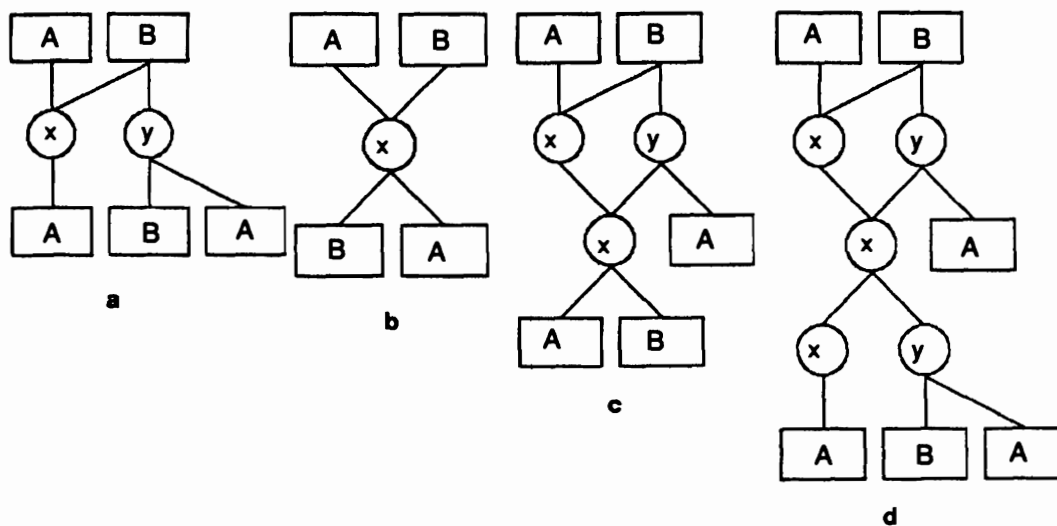


Figure 1

Four histories, h_1 in a, h_2 in b, h_3 in c, h_4 in d, such that h_1 before h_2 gives h_3 , and h_3 before h_1 gives h_4

A concurrent distributed system is specified by means of an abstract machine completely characterized by a set $Z=\{r_i\}$ of *atomic* concurrent histories, each representing an elementary evolutive step of part of the system, starting from a set I of initial processes composing the system. Roughly, an atomic history is a CH where there is only one event which is greater than all the tails and less than all the heads, as the one in Fig. 1.b. The computations are sequences $\{h_0, \dots, h_n\}$ of concurrent histories, where h_0 is the *initial* history (consisting of the set) I of initial processes, and history h_{i+1} is such that h_i before r_j gives h_{i+1} holds. Note that even if a computation is defined as a sequential concatenation of atomic histories, the application of the concatenating operation originates a partial ordering of events.

Actually, the abstract interpreter is a *rewriting system* where the rules are the atomic histories, the processes to be rewritten are a sub-set of the tail processes in the concurrent history representing the ongoing computation, and the rewriting step is performed via the *_before_gives_* operation. This kind of rewriting system is a generalization of the transition system, obtained by relaxing the assumptions that the state is monolithic and the transitions apply to the whole current state. The asynchrony of the model is enforced by the nature of the *_before_gives_* operation that leaves untouched all the tails not involved in the actual rewriting.

1.3. Graphic Interface: Editor and Interpreter

Any programming language needs tools for supporting development, modification and analysis of programs, and this need is even more demanding when the language is visual and devoted to the specification of large systems. CH aims at being the basis for such graphic specification languages. Any elaboration with CH results in a graph transformation; thus, besides the high expressivity due to its graphic nature, by providing CH with such tools the user can obtain:

- **better drawings:** the quality of computer aided drawing is superior to the average human ability;
- **quicker results:** a set of sophisticated functionalities are embodied that make it is much easier to modify a part of a drawing without sketching the other portions;
- **interactive presentation:** when the drawing is created incrementally, the user can test the possible different solutions and take the right decision;
- **less errors:** enforcing the syntactical graphic rules of CH directly by the tools drives the user to automatically follow the correct line, making the production of correct results easier. This helps the expert in avoiding careless mistakes, and the beginner in obtaining results which are, at least, coherent to the theory.

Since graphics and, more generally, ergonomical presentation are inevitable, in order to implement the kernel of the environment for CH, we were looking for a machine which offers these features for interacting with the user. Moreover, we desired the product to run also onto personal computer. Therefore, we have decided to choose Macintosh™ of Apple Computer as a host machine since it better fits with our requirements.

The kernel of the CH environment is essentially composed of two tools:

- **syntax-driven graphic Editor:** it creates files of atomic histories. Some basic syntactical rules are mandatory, while others are detected and reported to the user only when he/she tries to save the present format. Moreover, some typical functionalities as duplication of elements, “undo” of commands, saving of copies, etc., are available and presented in the typical user-friendly modalities of Macintosh;
- **graphic Interpreter:** it incrementally grows computations *via* the *_before_gives_* rule of concatenation between the computation performed so far and one of the atomic histories in the file previously created by the editor. Relevant features offered by the interpreter are some debugging functions, such as undoing the last concatenation step or setting backpoints to save certain points of the computation while exploring some portions of its future, and the possibility of modifying the format of the elements in the graph (shape, dimension, position, but not the links!) to obtain a better graphics. Also in this case the Mac philosophy is respected.

1.4. Modelling Languages in Terms of Concurrent Histories

A well-known approach to define the operational semantics of a programming language is Plotkin's Structural Operational Semantics, SOS for short³⁹. According to it, it is possible to define a transition system, i.e., the abstract machine of a programming language, in a sort of logical theory, by inducing on the syntactical structure of its programs. More in detail, the transition system has the formulae of the theory (or programs of the language) as states, and its theorems as transitions. Such a theory is defined by a set of axioms and inference rules, and verifying whether a transition exists corresponds to proving that the transition is a theorem of the logical theory. Hence, instead of giving an infinite set of transitions, a finite presentation is supplied. At each computation step, the finite set of transitions which can be applied is computed on a by-need basis, and one of them can be selected for application. Plotkin's SOS has been applied to a large variety of programming languages^{35,40,42} in the interleaving approach showing its flexibility. Degano, De Nicola and Montanari^{22,14,15,16,18,23} were able to extend this technique to the partial ordering approach, thus giving a more faithful representation of concurrent languages (see also^{38,4,44}). The resulting abstract machine is a *rewriting system*, rather than a (centralised) transition system; we will refer to this definition style as the DDM one.

A definition of the semantics of a language in Plotkin's and DDM styles can be quite naturally implemented by a set of Horn clauses, and thus by a Prolog program. When divergence and infinitely branching are not present in such a definition, the Prolog programs behave well in the sense that any goal has a finite set of solutions which can be computed in a finite interval of time. Thus, it is easy to develop a generator of abstract machines, which accepts the syntax of a process description language and an SOS or DDM specification of its semantics and produces such Prolog programs. We have implemented it in the framework of true concurrency.

In *Leonardo*, the generator of abstract machines is coupled with an interface, which, given a term denoting a state of a concurrent system, first runs the abstract machine for determining the transitions from that state, and afterwards transforms and displays them as atomic histories. Thus, we have got a *semantic-driven generator of interpreters* of process description languages in the truly concurrent approach. As a case study, we have considered CCS, and we refer the reader to ¹⁹ for a description of how its truly concurrent interpreter has been generated.

1.5. Top-down Specifications

Concurrent Histories, as well as all state-transition based models, describe a system at a definite level of abstraction. However, history decomposition into hierarchical layers should be a basic facility when considering the top-down development of a specification. Following such a methodology, the components of an abstract specification can be detailed stepwise, till the "actual" prototype is completely defined. In our setting, this is reflected in Fig. 2 where the dark grey event can be detailed to the partial ordering composed of light gray events (refinement).

The environment also supports the graphic facilities for observing a history at different levels of abstraction, where an event can be expanded into a new window containing in turn a hierarchical history. Presently, only a simple refinement mechanism is provided, that permits to represent by a single event a complex finite computation. Also the events in the refining computation may in turn represent more concrete activities. Extending the environment with a tool

which fully supports refinement and abstraction is not easy at all, mainly because these notions are not well understood yet from a semantic point of view; theoretical investigation has only recently started on this issue ^{11,27,26,25,31,1,17,12}.

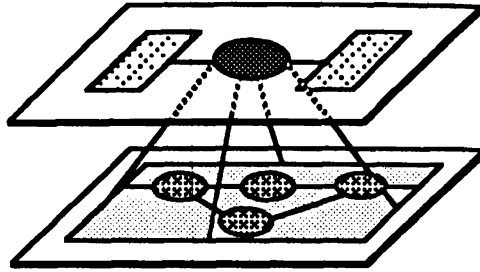


Figure 2. A two-level hierarchical representation of a Concurrent History

1.6. Ongoing Developments

A crucial extension to the proposed kernel is concerned with the support provided to the user for observing the nondeterminism of concurrent systems. In the classical solution, the runs of a system are factorised in a labelled tree (see, e.g., synchronisation trees, labelled on arcs³⁵ in the interleaving approach, or NMS with labels on nodes¹³ and Causal Trees with labels on arcs¹⁰ in the true concurrent one). Presently, our system is able to derive a NMS on a demand-driven basis, level by level, and allows the user to inspect its labels.

Further and theoretical deeper extensions will take care of reasoning about concurrent distributed systems. The development and implementation of proof systems is particularly important since the confidence in a specification is higher if it has been formally verified, and not only tested. We plan to incorporate in the basic environment checkers of bisimulation equivalences that preserve true concurrency and causality, or model checkers in classical or modal or temporal logics.

Finally, there is almost no limit to the number and kinds of features that one might want from a computer program to support ...

The paper is organised as follows. Section 2 surveys Concurrent Histories. The graphic editor and interpreter are presented in Section 3. Then, Section 4 presents the technique for describing a language and generating its truly concurrent interpreter in terms of Concurrent Histories.

2. Concurrent Histories

Let be A a countable set of observable actions and let E be a countable set of process types. Sets A and E are disjoint. A *concurrent history* h in H is a triple $h = (S, l, \leq)$, where S is a finite set of *sub-systems*, l is a *labelling function* $l : S \rightarrow A \cup E$ and $\leq$ is a *partial ordering* relation on S . The sub-systems with labels in A are called *events*, those with labels in E are called *process states* or simply *processes*. It is required that processes always be minimal or maximal (or both) elements of $\leq$, while events can never be maximal or minimal. Processes which are minimal are called *heads*, the processes which are maximal are called *tails*.

A history h is *a-atomic* if either

- i) there are no events and every head is smaller in the partial order than all the tails; or
- ii) there is exactly one event greater than all the heads and less than all the tails.

A history h is *s-atomic* if there is exactly one event greater than all the heads and some tails are not greater than the event, which are called *specials*. Later on, *atomic* will be used for both a-atomic and s-atomic.

The history in Fig. 1.b is a-atomic; the two atomic histories in Fig. 3 are s-atomic. Let us give an informal explanation of atomic histories by comparing them with transitions of Place/Transition Petri Net. A net transition corresponds tightly to an a-atomic history: the heads are the preconditions, the tails are the postconditions and the event corresponds to the transition; if the net transition is unobservable, then the corresponding history has no events and therefore every head is smaller than all the tails. A special process, instead, has no counterpart in this analogy. It represents a process which is enabled by the transition but which does not causally depend on the event performed. In Section 4 an example concerning CCS will clarify the advantage of having these unusual s-atomic histories, too.

Given two histories h_1, h_2 , and a history h , we write

$$h_1 \text{ before } h_2 \text{ gives } h$$

if and only if h can be obtained by the following procedure. The set S^{h_2} of the head processes of h_2 is matched against a subset S^{t_1} of the tail processes of h_1 , and corresponding processes are identified, the set S^{t_1} is called *rewritable*; nondeterminism is present if there is more than one rewritable set. Of course, two processes can match only if their labels are identical. The partial ordering $\leq_1 \cup \leq_2$ is then made transitively closed and the processes in the matching set $S^{h_2} = S^{t_1}$ are erased, i.e. the new set of events and processes is $(S_1 - S^{t_1}) \cup (S_2 - S^{h_2})$.

Given a set $Z = \{r_i\}$ of atomic histories and a set $I = \{p_1, \dots, p_m\}$ of isolated processes, h is a *computation* (on Z and I) if there exists a sequence of concurrent histories $\{h_0, h_1, \dots, h_n\}$ such that h_0 is the initial history (with no event, and with the heads coincident with the tails and) equal to I , and for each $i = 0, \dots, n-1$, we have that $h_i \text{ before } r_i \text{ gives } h_{i+1}$.

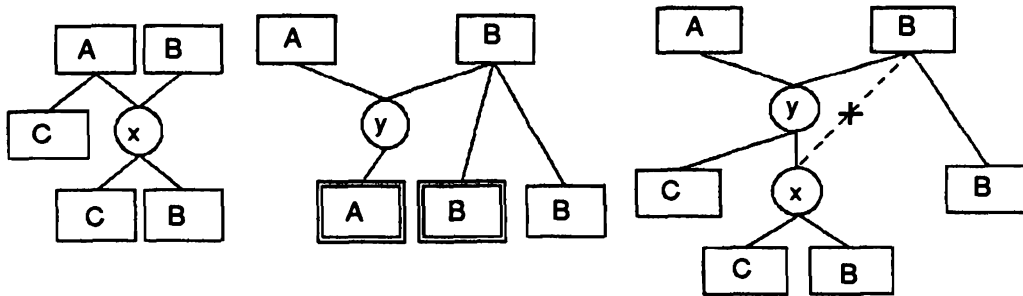


Figure 3

Application of the *_before_gives_* nondeterministic operation. The framed processes of the second history are selected to be matched against the heads of the first one. Notice that the dotted link is erased for transitivity.

3. The Graphic Tools: Editor and Interpreter

3.1. The Editor

The *graphic syntax-directed Editor* for Atomic Concurrent Histories (AH for short) supports the user in creating files of Atomic Histories which specify a concurrent system, through a friendly user interface. When the user opens or creates an atomic history file, a window like the one in Fig. 4.a appears, where rectangles are for heads and tails, and the square for the event.

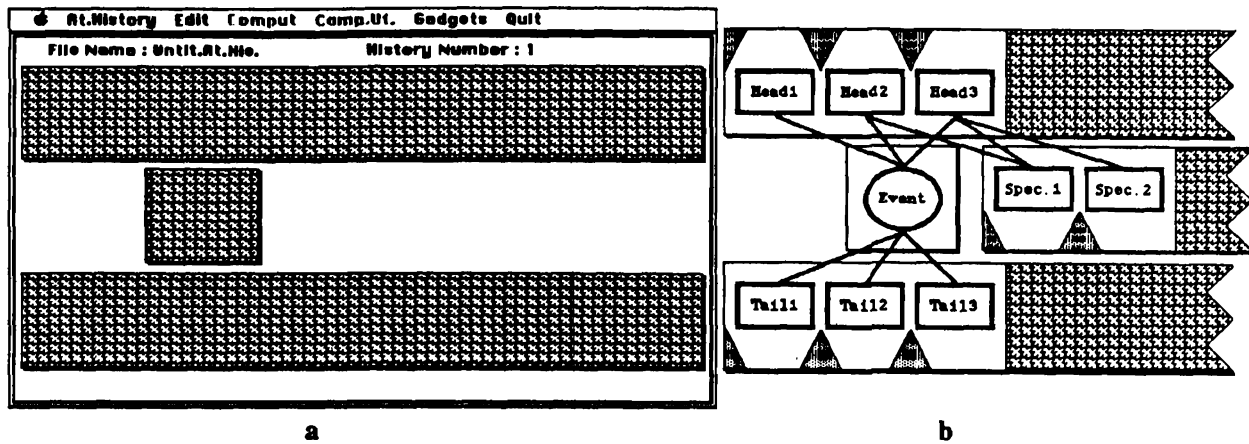


Figure 4. The initial window of the editor, in a, and an editing window, in b.

The Editor enforces the basic syntactical rules:

1. there are at least one head and one tail;
2. if there is an event, every head (tail) has one and only one link and precisely the head-event (event-tail) link;
3. if there is no event, every head has a link to every tail.

Selecting the command **Special** from the **Edit** menu, a rectangle aside of the square appears which creates a special process (see also Fig. 5). The definition of links from heads to specials is left to the user, but the Editor takes care that also the following requirement is fulfilled:

4. any special cannot be isolated and its links has to be tied to some heads.

The processes and the event can be created by clicking on the appropriate box, and then labelled. Triangles in the up-side (down-side) of the head (tail, special) window permit inserting a new process between the heads (tails, specials) already present. The usual Macintosh editing facilities, such as cutting, copying, pasting processes or events (or their labels), are available.

An AH may be closed if well-formed: it is saved in the current file of AH's and the screen is re-initialized to enable the creation of another AH; otherwise an alert box is shown reporting the syntactical error. AH's already created, which appear in a separate window, can be enlarged or shrunk. An AH can be now selected and cut, pasted, copied or loaded, as usual. Also a whole file of AH's can be copied.

3.2. The Interpreter

A computation, i.e. a run of the *Interpreter*, is always associated with a file of atomic histories. Thus, in order to create a new computation, a dialog appears asking for the wanted AH file. After choosing such a file, two windows appear (see also Fig. 6):

- the Computation Window in front;
- the Atomic History Window, containing the file of the associated AH's, behind.

Any CH representing a computation will start from a group of *initial* processes, which can be created with the command **New Initial Process** in the **Comp.Ut.** menu, and which represents the initial history¹ (see Fig. 5.a).

¹ In the analogy between CH and Petri Nets, the initial processes represent the initial marking.

A computation is grown by iterating computation steps, via the *_before_gives_* operation. The objects of a step are:

- an AH;
- a subset of the tails of the history representing the whole computation, i.e. a rewritable set.

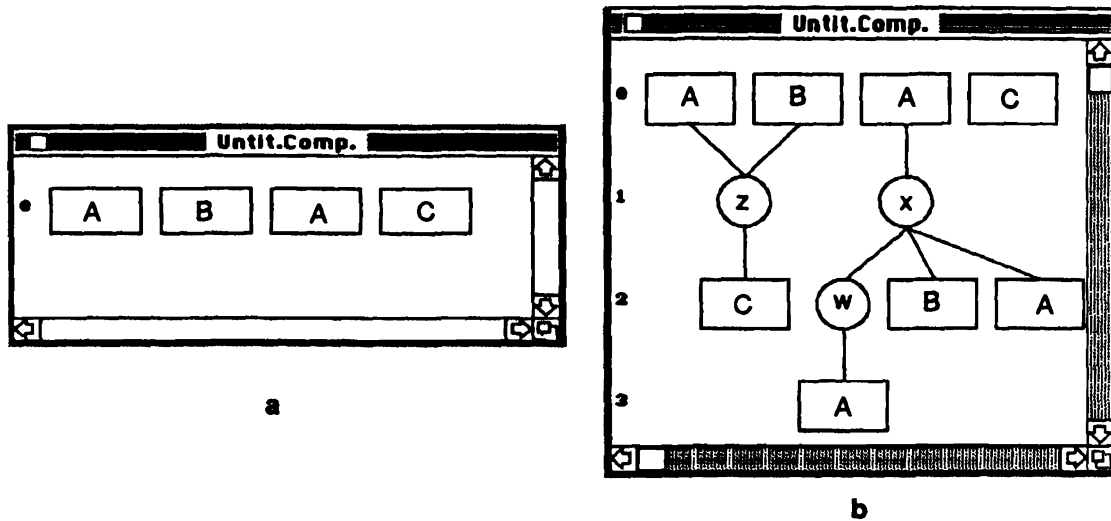


Figure 5

The initial state of a distributed system, in a, and the computation after the step determined by Fig. 6, in b.

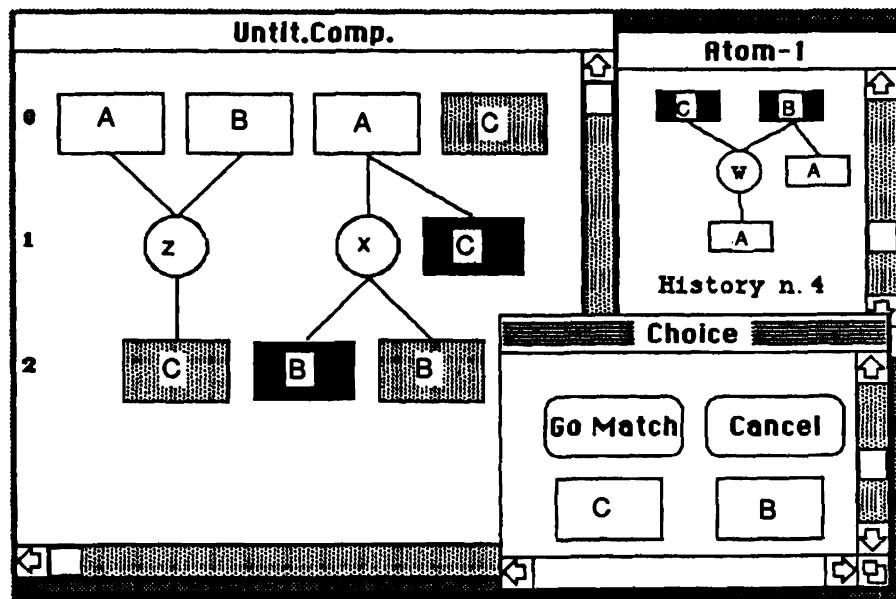


Figure 6. Untit.Comp, Atom-1 and Choice are the Computation window, the Atomic Histories one and the Choice one respectively. The Choice window shows one of the possibilities of matching, i.e. the one involving the black processes. The gray tails are selected but not used in the present matching.

The AH's Window shows only the AH's that match with the set of currently selected tails. Choosing the command Go Match from the Comp.Ut. menu causes one of the following:

- If there is no AH matching, then an alert box appears;
- If there is exactly one AH matching, the computation step is executed (see Figure 5.b);
- In case of nondeterminism, a new window (*Choice*) appears, where all the possible ways of applying an AH to the selected tails are showed (see the rightmost bottom window of Fig. 6).

3.3. Graph Make-up

The usability of the proposed tools largely depend on the quality of the graphic objects displayed by the system, and by its flexibility in customising them. Graphic options may include zooming capabilities, scaling functionalities, line-crossing minimisers, dragging facilities and many other graph make-up features. In the present implementation, some ways for adjusting graph elements are provided, including dragging an element to the desired position. A crucial problem is the minimisation of link crossing in the graph. An automatic two-step algorithm for this purpose is applied to atomic histories, following the idea developed in ⁴³. A version of the algorithm can be applied also to the concurrent histories representing computations.

3.4. Helps for Debugging

Some facilities for *debugging* are available, among which two different kinds of undo operation. By choosing the command **Undo Last Match** from the **Comp.Ut.** menu, the last computation step will be undone; by choosing the command **Undo Adjust** from the **Comp.Ut.** menu the selected elements will be brought back to their old position.

More interesting, backtrack points can be inserted when more than a single computation step is possible, thus controlling nondeterminism. This feature permits the user to explore some future configurations the system will pass through, and to return then to a wanted backtrack point from which the other enabled computation steps can be performed. In this way, it is possible to exhaustively search all the finite computations of the system specification under testing. As a further help in managing nondeterminism, the tree of the computations of a system can be generated, level by level, on a by-need basis, and the user can inspect it by displaying its labels.

3.5. Top-down Specifications

The environment provides a facility for refining an action into a computation. In defining a set of atomic histories, the editor permits to click on events and to define a set of lower level atomic histories using the command **Refine event**. During the interpretation, the refined event can be seen at its own level, or further detailed through the atomic histories that have been introduced before. A new window, **Refined computation window**, will appear, having as heads the heads of the event under refinement, and the computation goes on as usual. It ends, and the refinement succeeds, only if its tails are exactly the tail of the event under refinement. When the user inspects a computation, a simple click on the refined events will display the computation that was used to detail that event.

4. A Generator of Truly Concurrent Interpreters

As noted above, the definition of the semantics of a language in Plotkin's or DDM style can be quite naturally described as a set of Horn clauses, and thus implemented by a Prolog program. As a matter of fact, it is easy to design (and we have implemented) a generator of abstract machines which accepts the syntax of a process description language and a DDM specification of its semantics and produces a Prolog program. A successive phase will take care of calling this Prolog program in order to deduce the transitions involving the state corresponding to a given

set of concurrent processes. In turn, the transitions will be displayed as Atomic Histories and eventually applied to the present state. As a whole, we have got a *semantic-driven generator of interpreters* of process description languages in the truly concurrent approach. In the following section we will consider as a case study CCS, and we will present its truly concurrent interpreter based on ¹⁶. We now give a short account of the technique by Degano, De Nicola and Montanari, which extends Plotkin's SOS to obtain compositional operational semantics in the framework of true concurrency.

The DDM paradigm is as follows.

- Decompose syntactically a term of the used process description language, representing a whole concurrent process, into the set of its sequential components $\{S_1, S_2, \dots, S_n\}$, which can be thought as working concurrently. The set of sequential components denote the state of the system and have a straightforward representation as processes of a CH. The syntax driven decomposition is injective so that well-formed sets of sequential components can be seen again as terms of the language when needed.
- Define the dynamics of those sub-sets of sequential components which can proceed in isolation and describe their evolution as Atomic Histories.

For example, the sequential components of a CCS term $(\alpha.NIL|\beta.NIL)+\gamma.NIL$ is the singleton $\{(\alpha.NIL|\beta.NIL)+\gamma.NIL\}$, since nondeterministic choice is considered as sequential (for a different approach see ¹⁴). The sequential components of $\alpha.NIL|\beta.NIL$ form instead the set $\{\alpha.NIL|id, id|\beta.NIL\}$, where tags *lid* and *idl* record that $\alpha.NIL$ was the left operand of a parallel operator, and $\beta.NIL$ the right one. A transition from the state $\{(\alpha.NIL|\beta.NIL)+\gamma.NIL\}$, deduced in a merely compositional way, is depicted in Fig. 7 as an Atomic History.

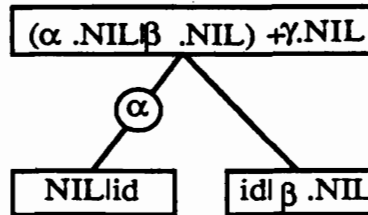


Figure 7. An AH representing an α -transition from state $\{(\alpha.NIL|\beta.NIL)+\gamma.NIL\}$ to state $\{NIL|id, id|\beta.NIL\}$.

Informally, the single process in the head can perform action α and transform into two processes, one of which is a tail, namely $NIL|id$, the other a special, $id|\beta.NIL$. In this way, we describe the fact that component $(\alpha.NIL|\beta.NIL)+\gamma.NIL$ causes both $id|\beta.NIL$ and the event labelled by α which in turn causes component $NIL|id$. Note that the possibility that $id|\beta.NIL$ may have to perform β independently of the occurrence of α is implied by the absence of any causal relation between α and $id|\beta.NIL$. A formal definition of the truly concurrent semantics of CCS sketched above has been given in ¹⁶.

Once given a syntax of a process description language, the decomposition of its terms in sequential components and its semantics in the DDM style, a little interface transforms the latter into a legal Prolog program. Given a set of tails, the inference machine of Prolog will be used to compute the needed transitions. It has a transition as a goal, i.e. a 4-uple consisting of a ground sub-set of the given tails (i.e. the heads of the deducible atomic history), and three variables to be grounded: the first one ranges over actions, the second one over a set of sequential compo-

nents (i.e. the tails and the specials of the atomic history), the third one over the causal relation (actually only its part concerning heads and specials).

Another module now has as input the set of the deduced transitions, transforms them into a graphic form thus producing the file of AH which will be then displayed in the Atomic Histories Window. The user, with the help of the Choice Window, selects one of the AH's just computed and applies it to the present state of the computation *via* the graphic interpreter of the CH. Of course, the initial history from which the computation begins is obtained by decomposing a given term. Note that also the decomposition is implemented by transforming the user's definition into a Prolog program.

5. References

1. L. Aceto, M. Hennessy, Towards Action-Refinement in Process Algebra, in Proc. 4th Annual Symp. on Logic in Comp. Scie. (LICS'89), Asilomar, IEEE Computer Society Press, (1989), 138-145.
2. G. Boudol, I. Castellani, Concurrency and Atomicity, *Theoret. Comput. Sci.* 59, 1-2 (1988), 25-84.
3. G. Bruno, A. Elia, Operational Specification of Process Control Systems: Execution of PROT Nets Using OPS5, in Proc. 10th IFIP Congress, (1986), 35-40.
4. A. Brogi, R. Gorrieri, A Distributed, Net-Oriented Semantics for Delta Prolog, In Proc. TAPSOFT '89, LNCS 351, Springer, (1989), 162-177.
5. S. D. Brookes, C.A.R. Hoare and A.D. Roscoe, A Theory of Communicating Sequential Processes, *Journal of ACM*, 31, 3 (1984), 560-599.
6. J.A. Bergstra, and J.-W. Klop, Process Algebra for Synchronous Communication, *Information and Control* 60, (1984) 109-137.
7. M. Broy, T. Steicher, Views of Distributed Systems, Proc. Advanced School on Mathematical Models for the Semantics of Parallelism, LNCS 280, Springer, (1987), 114-143.
8. J. Billington, G.R. Wheeler, M.C. Wilbur-Hamm, PROTEAN: A High-Level Petri Net Tool for the Specification and Verification of Communication Protocol, *IEEE Trans. on Soft. Eng.*, 14 (3), (1988), 301-316
9. R. Cleaveland, J. Parrow and B. Steffen, The concurrency workbench, in 29, pp. 24-37.
10. Ph. Darondeau, P. Degano, Causal Trees, Proc. 16th ICALP, LNCS 372, Springer, (1989), 234-248.
11. Ph. Darondeau, P. Degano, About Semantic Action Refinement, *Funda. Infor.* XIV, (1991), 221-234.
12. P. Degano, R. Gorrieri, Parallel Procedure Call as Event Structure Substitution, technical report, Dip. di Informatica, Uni. di Pisa, March 1991.
13. P. Degano, R. De Nicola, U. Montanari, Observational Equivalences for Concurrency Models, in *Formal Description of Programming Concepts III* (M. Wirsing, Ed.), North Holland, (1987), 105-132.
14. P. Degano, R. De Nicola, U. Montanari, A Distributed Operational Semantics for CCS based on Condition/ Event Systems, *Acta Informatica* 26, (1988), 59-91.
15. P. Degano, R. De Nicola, U. Montanari, Partial Ordering Description of Nondeterministic Concurrent Systems, in Proc. REX School on Linear time, Branching Time and Partial Order in Logics and Models for Concurrency, LNCS 354, (1989), 438-466.
16. P. Degano, R. De Nicola, U. Montanari, A partial ordering semantics for CCS. *Theoretical Computer Science*, 75, (1990), 223-262.
17. P. Degano, R. Gorrieri, Atomic refinement for process description languages, Proc. MFCS'91, Warsaw, LNCS, (1991), to appear.
18. P. Degano, R. Gorrieri and S. Marchetti, An exercise in concurrency: a CSP process as a Condition/Event system, In *Advances on Petri Nets 1988*, LNCS 340, Springer, (1988) 85-105.
19. P. Degano, R. Gorrieri, L. Zamboni, P. Zanotti, The Kernel of a Graphic Environment for Analyzing Distributed Systems, technical report, Dipartimento di Informatica, Università di Pisa, 1990.

20. P. Degano, U. Montanari, Distributed systems, partial orderings of events and event structures, in: *Control Flow and Data Flow-Concepts of Distributed Programming*, Nato Asi Series F 14, (1985) 7-106.
21. P. Degano, U. Montanari, Specification languages for distributed systems, Proc. TAPSOFT '85, LNCS 185, Springer, (1985), 29-51.
22. P. Degano, U. Montanari, Concurrent Histories - A basis for observing distributed systems, *Journal of Computer and System Sciences*, 34, (1987), 422-461.
23. P. Degano, S. Marchetti, Partial ordering models for concurrency can be defined operationally, *Int. Journal of Parallel Programming*, 16, 6 (1987), 451-478.
24. F. Feldbrugge, K. Jensen, Petri Net tool overview 1986, LNCS 255, Springer, (1986) 20-63.
25. R.J. van Glabbeek, U. Goltz, Equivalence Notions for Concurrent Systems and Refinement of Actions, Proc. MFCS'89, LNCS 379, Springer, (1989) 237-248.
26. R. Gorrieri, U. Montanari, Towards hierarchical description of systems: a proof system for the strong prefixing, *Int. J. of Foundations of Computer Science* 1, (1990), 277-293.
27. R. Gorrieri, S. Marchetti and U. Montanari, A²CCS: Atomic Actions for CCS, *Theoretical Computer Science* 72, (1990), 203-223.
28. U. Goltz, W. Reisig, The Non-sequential Behaviour of Petri Nets, *Info. and Co.* 57, (1983) 125-147.
29. Proc. International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, LNCS 407, Springer, (1989).
30. A. Giacalone, S. Smolka, Integrated Environments for Formally Well-Founded Design and Simulation of Concurrent Systems, *IEEE Trans. on Soft. Eng.*, 14 (6), (1988), 787-802.
31. R.J. van Glabbeek, W.P. Weijland, Refinement in Branching Time Semantics, Proc. Int. Conf. on Algebraic Methodology and Software Technology (AMAST), (1989), 197-201.
32. D. Harel, Statecharts: A Visual Approach to Complex System, *Science of Comp. Prog.*, 8 (3), (1987) 389-449.
33. M. Hennessy, *An Algebraic Theory of Processes*, MIT Press, (1988).
34. *Journal of Parallel and Distributed Computing*, 9, Special Issue on Software Tools for Parallel Programming and Visualization (1990).
35. R. Milner, A Calculus of Communicating Systems, LNCS 92, Springer, (1980).
36. M. Nivat, Behaviours of Processes and Synchronized Systems of Processes, In: *Theoretical Foundations of Programming Methodology* (M. Broy and G. Schmidt Eds), (Reidel, Dordrecht, 1982), 473-550.
37. M. Nielsen, G. Plotkin, G. Winskel, Petri Nets, Event Structures and Domains, Part 1, *Theoretical Computer Science*, 13 (1981), 85-108.
38. E.-R. Olderog, Operational Petri Net Semantics for CCSP. In *Advances in Petri Nets 1987*, LNCS 266, Springer (1987) 196-223.
39. G. Plotkin, A Structural Approach to Operational Semantics, TR DAIMI FN-19, Aarhus Uni. (1981)
40. G. Plotkin, An Operational Semantics for CSP, in *Formal Description of Programming Language Concepts II*, North Holland, (1983) 199-223.
41. V. Pratt, Modelling Concurrency with Partial Orders, *Int. Journal of Paral. Prog.*, (1986) 33-71.
42. V. Saraswat, The Concurrent Language CP: Definition and Operational Semantics, Proc. 14th Symposium on Principles of Programming Languages, ACM, (1987) 49-62.
43. K. Sugiyama, T. Shojiro, T. Mitsuhiro, Methods for Visual Understanding of Hierarchical System Structures, *IEEE Trans. on Sys., Man. and Cyb.*, Vol. SMC-11, 2 Feb. 1981, 109-125.
44. D. Taubner, Finite Representation of CCS and TCSP Programs by Automata and Petri Nets, LNCS 369, Springer, (1989).
45. J. Winkowski, Behaviours of Concurrent Systems, *Theoretical Computer Science* 12, 39-60 (1980).
46. G. Winskel, Event Structures, in *Advances in Petri Nets 1987*, LNCS 266, Springer, (1987) 196-223.

A Design and Specification Environment for Distributed Memory Multiprocessors^{*}

Thomas Bemmerl, Thomas Ludwig, Bernhard Ries

Institut für Informatik, Technische Universität München
Arcisstr. 21, D-8000 München 2, Germany

ABSTRACT

In order to facilitate the development of large and well-structured parallel applications, a multiprocessor programming environment must comprise tools which support the specification phase. The following paper gives an overview of the design and implementation of the SAMTOP environment which has been implemented as a part of the TOPSYS programming environment for distributed memory multiprocessors. The specification method is based on SA/SD, a design method widely used for the development of conventional systems. It has been adapted to meet the requirements of the designer of parallel applications. Key features include a graphical editor for the design of process graphs and code generation facilities which allow for a smooth transition from the specification to the implementation phase.

Introduction

Complex and computation-intensive problems, such as real-time image processing, fluid dynamics and VLSI-layout require computers with computational powers far beyond that of modern single-processor systems. A lot of research has therefore been directed towards the development of multiprocessor-systems. By now some of these systems are readily available on the market.

The development of faster hardware does, however, not automatically imply the solution to the problems mentioned above, since it is significantly harder to develop efficient programs for multiprocessors, than for conventional single-processor machines. The majority of development environments for multiprocessors are still very rudimentary. In most cases, the programmer has to keep in mind the details of the architecture and operating system. This makes it extremely difficult to develop portable and efficient programs.

Up to now, there is a special lack of specification-environments which allow the user to design a parallel application at a high level of abstraction and arrive at the implementation through a number of well-defined and structured steps.

The goal of the project TOPSYS (Tools for Parallel SYStems)² at the Technical University of Munich, is to develop a portable environment which integrates tools that

^{*} Partly funded by the German Science Foundation under contract number SFB 0342, project A1

help programmers cope with every step of the software development-cycle for parallel applications. It is therefore necessary to implement tools which support specification and design, coding and debugging as well as optimizing multiprocessor programs. Consequently, the TOPSYS-environment comprises:

- a CASE-environment for the design and specification of applications (SAMTOP),
- a multiprocessor multitasking operating system kernel (MMK),
- a high-level debugger (DETOP, DEbugging TOol for Parallel systems),
- a visualizer (VISTOP, VISualization TOol for Parallel systems) and
- a performance-analyzer (PATOP, Performance Analysis TOol for Parallel systems).

The tools are based on the MMK operating system³ which has been implemented for Intel's iPSC/2 hypercube. MMK programs consist of a set of tasks which communicate via mailboxes and use semaphores for the purpose of synchronization. These objects form a global object space in which any task can easily access every other object, without having to know on which node the object resides. This paper gives an overview of the rationale and design choices for the specification and mapping tool SAMTOP (Specification and Mapping Tool for Parallel Systems).

1. Criteria for a Design Method for Multiprocessor-Software

When developing an environment for the design and specification of software, the first important step is to postulate requirements which should be met by the tools and to find criteria which can be used to evaluate different approaches. The next paragraph concentrates on the formulation of such a set of requirements. To a certain extent these requirements are similar to those used to evaluate conventional design methods while others are specific to the field of multiprocessor software-design.

The most important consideration is that a design method should be easy to learn, understand and use. In most cases, specification systems are used for the design of large applications, therefore a lot of time is spent working with the specification system. If the design tools are too difficult to use or the resulting specifications are too difficult to understand, the design-environment will most likely not be accepted by the user community.

A lot of research has been invested into the development of specification-methods which are based on complex mathematical formalisms, such as temporal logic¹¹, CSP⁴ or petri-nets^{10,12}. These methods have the advantage that the mathematical formalism can be used to prove the correctness of a specification, and in some cases even to derive implementation parts from the specification. At the actual state of research, however, most of these methods can not be used for the design of large applications, because the formalisms are just too difficult to manage.

It is our belief that a design method which is to be widely accepted has to reduce mathematical formalisms to a minimum in order to be easy to understand and use. A certain degree of formality is, however, necessary in order to make consistency checks possible. These requirements point to the use of a graphical design method, since standardized graphics provide a certain degree of rigidity in a specification, without making mathematical formalisms necessary.

Modern workstations make a considerable amount of computational power readily available to almost every programmer. It is therefore straightforward to use software-tools for the design and specification of parallel programs. In order to facilitate the communication between project members, it is very important to have a powerful database as the central part of the design-environment. This makes it possible to efficiently inspect and modify parts of the specification. Since large software projects normally require a considerable number of project members, a design-environment should run in a network environment and integrate mechanisms for version-control.

One advantage of a computer-based specification method, is that it facilitates automatic consistency checks. The extent to which such checks are possible, depends on the amount of formalism inherent in the specification. In the field of parallel processing, there are a number of important questions which have to be considered when designing an application, e.g. proving that a system is deadlock-free or checking the soundness of the flow of data through the system. A specification system should incorporate as many consistency-checks as possible.

Computer-based design-environments make it possible to automatically generate parts of the implementation directly from the specification. These code-generators also help to ensure the portability of the application programs. The design and specification can take place at a high level of abstraction, without taking into account implementation details, such as the target-architecture or the operating system used. The use of different code-generators for different target-systems greatly simplifies the task of porting a parallel application to a new environment.

An environment for the development of multiprocessor-software should comprise tools which support all the phases of the software development cycle, such as debuggers, visualizers and performance analyzers. User-friendly design- and specification-tools should be compatible with the rest of the tool-environment, in order to facilitate the transition from the specification to the implementation. Once again, this is a strong argument for the use of a graphics-based specification method, since graphics provide a good basis for all of the tools mentioned above. For example, a parallel program can be modeled by a process graph, in which nodes represent the tasks, and edges represent communication links between tasks. Such a representation is also well-suited for the visualization of the dynamic program behavior, and can be used as a user-interface for a debugger or performance-analyzer.

A design method for parallel applications, on concepts which have been widely tested in practice and found to be useful is certain to be more easily accepted by the users, than a method based on a completely new approach. At the moment, however, there is no design methodology for multiprocessor software which is widely used. In the field of specification and design of conventional software the situation is somewhat different. A number of so called CASE-methods such as Structured Analysis/Structured Design (SA/SD)^{5,23}, SADT²⁰ or JSP⁹ are being used more and more for the design and specification of large software systems. It is therefore straightforward to use one of these methodologies as a basis for a new design method for parallel applications. There are, however, a number of points in which the design process for multiprocessor applications differs considerably

from the steps used for the design and specification of conventional software systems. Thus, the new method must contain a certain amount of changes and innovations.

A specification method for parallel applications should enable the user to model systems without taking into account the specific details of the architecture and the operating system used. However, if the specification is to be transformed into the implementation of a system by a series of well-defined steps, these details can not be totally omitted - as the design gets closer to the implementation, the properties of the target environment have to be considered. It is not plausible to search for a methodology which is suited for the design of applications for both tightly coupled and loosely coupled multiprocessors, since the programming models and architectural concepts used in these two classes of machines are very different. In most programming models for loosely coupled multiprocessors, a parallel program consists of a set of active components (tasks, processes etc.) which exchange data through message passing. Design methods which emphasize the flow of data through the components of a software system are particularly well-suited for the design of applications for this class of machines. The most common CASE-methods are based on the principle of dataflow through active components. It is therefore plausible to use one of these methods as a starting point for a design method for applications which are to run on message-based multiprocessors.

As we have seen, a computer based design method for parallel applications should not be overly formal but specifications should contain enough information to allow consistency checks and code generation. It is possible to meet these requirements with graphical methods. In addition to relying strongly on graphics, CASE methods such as SA/SD have the advantage that they have already been widely employed for the design of large conventional software systems. Thus a CASE method centered around a powerful database offers a good platform for the design of multiprocessor applications. For the design of SAMTOP, we therefore chose a set of CASE-tools and tried to enhance them by adding concepts which address issues specific to multiprocessor applications.

2. Overview of Existing Approaches

A number of researchers have tried to tackle the problem of structured specification and design of multiprocessor applications. The following paragraph gives an overview of the most successful design environments for message-based multiprocessors.

One of the first programming-environments for non-shared memory applications was the Poker system developed at the University of Washington^{7,16}. A Poker program is represented by a process graph, with vertices representing processes and edges representing communication paths between processes. A very similar approach is taken in the Muppet system¹³ which is part of the German Suprenum project. The concepts used in the CODE system²¹ which is being developed at the University of Texas, are of a more formal nature. Here, a parallel program is viewed as a set of units of computation, and a number of dependency relations which specify the order of execution of the units of computation. All three systems make extensive use of graphics - graphical editors are used to construct a process graph. Some systems try to make automatic transitions from the specification to

the implementation possible. For example, a CODE-specification can be translated into an Ada program frame that calls the computation units as Ada subprograms while taking into account the specified dependency relations.

The representation of parallel programs as process graphs is very appealing, since such graphs are a good basis for the visualization of the topology and communication scheme of the algorithm. A graphical editor which is to be used for the structured design of applications should, however, enable the designer to partition the process graph into subgraphs or subsystems. This feature is of special importance for the design of large applications. As we shall see in the following, the graphical design of parallel applications can get very cumbersome in some cases. It is therefore helpful, if the design environment offers the possibility of describing parts of the specification in a textual form. Examples for such an approach can be found in Polyolith¹⁸, Pisces¹⁷ and the Design project¹⁴.

In our view, a design environment for parallel applications should combine both approaches. A graphical editor should be used to construct a well-structured program graph, but at the same time the user should be able to resort to a textual representation, whenever a graphical representation is not adequate. In addition, the design environment should support a structured and - as far as possible - automated transition from the specification to the implementation through a series of code-generation steps.

3. Design of Multiprocessor-Applications using SA/SD

The previous paragraphs have shown that conventional CASE-tools offer a good platform for the specification and design of multiprocessor applications. However, the methods used for the design of conventional software systems have to be adapted in a number of ways, in order to meet the requirements of the designer of parallel systems. The following paragraph analyzes the strengths and weaknesses of the CASE-method Structured Analysis/Structured Design which was chosen as a starting point in the design of SAMTOP.

The SA/SD-method developed by Yourdon and DeMarco is one of the most successful methodologies for the design of conventional software systems. According to this approach, a software system is modeled as a set of active components which transform input data into output data. This dataflow-oriented approach is particularly well-suited for the design of applications for message-based multiprocessors. Since the tasks of a multiprocessor program usually need to synchronize from time to time, it is necessary to add constructs which can be used to reflect synchronization statements, as for example operations on semaphores. This can be done by using Ward's extensions to the conventional SA/SD method²² which make it possible to model the flow of control in a system. A parallel program can thus be represented by a dataflow graph, in which vertices represent the tasks or processes of the system, dataflow symbols represent communication links, and control-flow symbols are used to model synchronization operations. In the SA/SD-approach, the dataflow graph can be designed in a well-structured way, since it is possible to partition it into subsystems which reflect the modular structure of an application. The design is done by stepwise refinement of system components using a top-down approach.

The representation of a parallel computation structure by a dataflow graph is particularly useful for the design of applications, whose parallelism is based on the principle of task-partitioning. In this approach, a problem is solved by dividing into a set of distinct sub-problems. Each problem is then solved by a different type of task. This type of application can obviously be easily modeled by a dataflow graph, which shows the different tasks and their communication and synchronization links.

For many important problems the principle of task partitioning is not sufficient, since in many cases it is more natural to partition the data set rather than partition the flow of control of the program⁸. This means that a certain type of task appears in more than one instantiation - many data-parallel algorithms consist of only one task type with every available processor node running a copy of the task which works on a part of the data set. The principle of data-partitioning is of special importance for massively parallel systems, since it is not feasible to program a machine which consists of thousands of processors by creating an equal number of different task types. In many cases both task-partitioning and data-partitioning are used in the design of parallel algorithms.

The design of data-parallel algorithms using dataflow-graphs is somewhat awkward. The reason for this is that the fact that a task is replicated, that is, occurs more than once, cannot be easily expressed. It is of course possible to express replication by simply copying parts of the dataflow graph. This is, however, not satisfactory, because typical data-parallel algorithms have a characteristic interconnection structure, such as for example a ring- or torus-configuration. The size of the configuration is variable, typically it is a function of the number of available processor-nodes.

It follows that the conventional design principle used for SA/SD is not suitable for the design of data-partitioned applications. A design method for multiprocessor software has to contain mechanisms which can be used to express replications of parts of the dataflow graph and to model interconnection structures between replicated components.

One of the drawbacks of using SA/SD as a design method for multiprocessor applications, is that this method makes it difficult to model a system's dynamic behavior. Even though State-Transition Diagrams can be used to specify the dynamic aspects of tasks, the structure of the dataflow graph remains static, since there is no way to model the dynamic creation or destruction of objects. The dataflow graph can therefore only reflect the initial structure or a stable end-configuration of a parallel application.

Conventional CASE-Tools centered around the principles of SA/SD typically allow a number of consistency checks, such as checking the soundness of the flow of data through the dataflow graph. Even though these tests are useful in the design of multiprocessor applications, there are often other requirements which must be met by a multiprocessor application. As an example, it would be very useful if it were possible to check, whether the modeled system is free from deadlocks. As a prerequisite for this kind of tests, methods that allow a detailed specification of dynamic program behavior are necessary.

The previous paragraphs have shown that the conventional SA/SD method needs some changes and extensions, if it is to be used for the design of multiprocessor software. The strength of the SA/SD design principle is that it is possible to design a system in a structured way through stepwise refinement. However, extensions are necessary, to support the

design of data-partitioned applications and consistency checks for typical requirements multiprocessor applications must meet. Many CASE-environments offer code generators which can be used to derive parts of the implementation from the specification. At this point, a design method for multiprocessor applications should offer support for problems typically found in multiprocessor programming, such as the mapping of computational structures to processor nodes.

4. The Samtop CASE-Environment

The Samtop CASE-Environment is based on the CARDtools (Computer Aided Real Time Design Tools)¹⁹, developed and distributed by Ready Systems. The CARDtools offer software design facilities according to Ward's extensions to SA/SD for the design of real-time systems. This CASE-environment includes the following features:

- structured top-down design of applications using structure charts and dataflow diagrams,
- specification of data types,
- specification of algorithms using pseudocode,
- detailed design of process graphs containing operating system primitives.

In the design of SAMTOP, we tried to adapt and extend the functionality offered by the CARDtools, in order to meet the requirements of the designer of multiprocessor software. The following paragraph describes the structure and functionality of the resulting CASE-environment for multiprocessor applications.

In the SA/SD approach, a software system is designed using a top-down approach. This is done using so-called structure charts which show the modular structure of a system, i.e. its decomposition into different subsystems. In our approach, structure charts can be designed using the CARDtools Control Maps Builder. A structure chart can, however, also be seen as a dataflow diagram which emphasizes the flow of data through the system rather than the system's hierarchical structure. Dataflow diagrams can be derived from structure charts or designed from scratch using a graphical editor, the so called Data-Flow Diagram Builder. In a final design step, operating system primitives can be used to specify, how the functionality of a design is to be implemented. For example, active components can be replaced by symbols representing tasks or interrupt-severs, dataflows can be replaced by mailbox-icons and semaphores can be used as a substitute for control-flow symbols. Thus, the design yields a detailed process-graph showing the operating system components and their interconnection (see figure 1). The structure of the data items exchanged by tasks can be specified using a data-definition facility. It is also possible to use a pseudocode-language to specify the exact algorithms used.

As we have seen in the previous paragraph, the graphical design of dataflow diagrams and process graphs is not very helpful for the design of data-partitioned applications. The conventional method has to be enhanced, in order to offer a way of expressing the replication of parts of the process graph and the interconnection structure between replicated parts. One possible approach is the use of graph-grammars as described by Bailey and Cuny¹. But this is once again a very formal concept which requires special skills on part

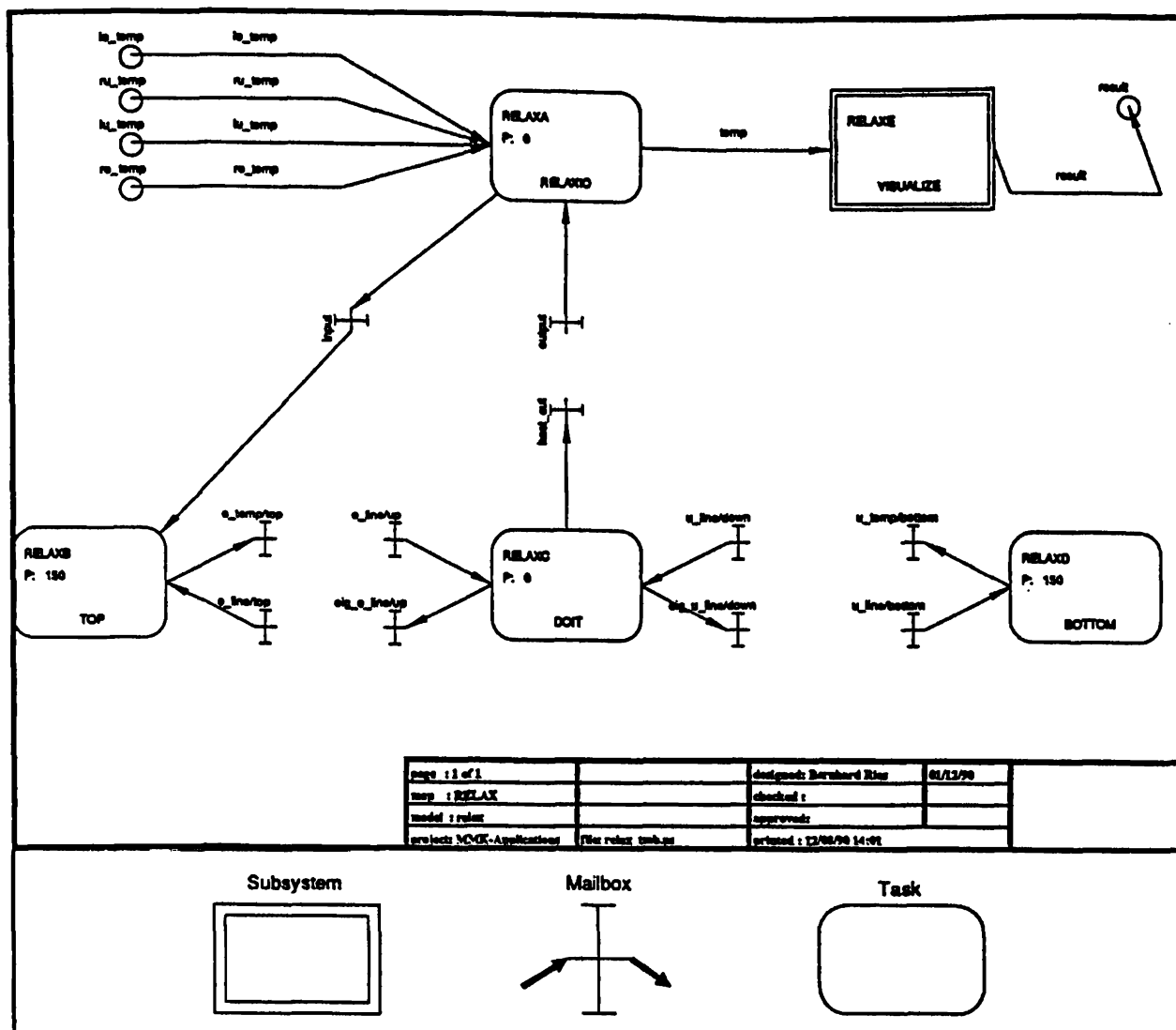


Figure 1: Detailed example process graph.

of the designer. In the design of SAMTOP we therefore developed a simple programming language which offers a natural way to express the structures mentioned above. This language contains conventional C-statements and three special operations: replicate, connect and map. The replicate-operation can be used to create multiple instantiations of parts of the process graph. In the initial design of the process graph, parts that are to be replicated are not connected with the rest of the graph, since they are only templates for the different instantiations. The interconnection structure is created using the connect-operation. Finally, the map-operation can be used to specify the mapping of system objects to processor nodes. Parameters can be used freely in all operations. This makes it easy to express that the exact structure of an application is a function of certain system parameters, e.g. the number of available processor nodes.

Consequently, a complete system model in the SAMTOP-environment consists of a graphically designed specification of the different program objects and a small file

specifying the replication and interconnection structure.

The statements shown in figure 2 model the structure of a producer-consumer system, in which n producers communicate with m consumers via a single mailbox.

In order to facilitate the transition from the specification to the implementation, the SAMTOP-environment incorporates code-generators, document-generators and a mapping-facility.

At the moment, the code-generation facilities offered are rudimentary, it is only possible to derive templates for the task types of a system model. These templates contain task declarations and are used to supply the tasks with parameters. These parameters typically include the identifiers of communication objects, e.g. the mailboxes and semaphores with which a task is connected.

A number of documents can be generated directly from the specification. These include a data types dictionary, data- and procedure cross-reference listings, invocation tree and pseudo-code analyzer reports. In addition, it is possible to create print-ready versions of structure-charts, dataflow-diagrams and process graphs. Figure 3 shows a configuration file generated from a specification.

The mapping of program objects to processor nodes is a crucial step in the design of multiprocessor applications, since a good mapping is essential in order to balance system load and avoid communication bottlenecks. In many cases, this mapping can be thought of as part of the specification. As an example, consider a ring-structured application which is to be mapped onto a hypercube multiprocessor. In a good mapping, neighboring objects in the ring should be mapped onto processor nodes which share a communication link. Such a mapping can be achieved by using a gray-code. However, it is very often not sufficient to consider only the topology of the computational structure and the target architecture when determining a mapping function. Other important factors are the computation load and amount of communication generated by program objects as well as the degree to which different objects interfere with each other through synchronization opera-

```
#define N 10
#define M 20

REPSTART
int i, j;
/*
** producer and consumer are objects, whose exact structure was
** previously defined in the design of the dataflow graph;
** replication yields N consumers and M producers indexed as a vector */
replicate(producer, VECTOR[N]);
replicate(consumer, VECTOR[M]);
/*
** Interconnection structure: The output-mailbox of every producer
** is connected with the input mailbox of every consumer
*/
for (i=0; i; i++)
    for (j=0; j; j++)
        connect(producer[i].out, consumer[j].in);
REPEND
```

Figure 2: Replication and interconnection in a producer-consumer system.


```

GLOBAL BEGIN
IO( 10000 NODE 0 )
MBOX( bottom INIT 0 NODE 7 )
MBOX( down_0 INIT 0 NODE 0 )
MBOX( down_1 INIT 0 NODE 1 )
MBOX( down_2 INIT 0 NODE 2 )
MBOX( down_3 INIT 0 NODE 3 )
MBOX( down_4 INIT 0 NODE 4 )
MBOX( down_5 INIT 0 NODE 5 )
MBOX( down_6 INIT 0 NODE 6 )
MBOX( top INIT 0 NODE 0 )
TASK( BOTTOM INIT bottom_code,32000, 150 ,bottom NODE 7 )
TASK( TOP INIT top_code,32000, 150 ,top ,input NODE 0 )
TASK( DOIT_0 INIT do_it_code,32000, 100 ,top ,down_0 ,output NODE 0 )
TASK( DOIT_1 INIT do_it_code,32000, 100 ,down_0 ,down_1 ,output NODE 1 )
TASK( DOIT_2 INIT do_it_code,32000, 100 ,down_1 ,down_2 ,output NODE 2 )
TASK( DOIT_3 INIT do_it_code,32000, 100 ,down_2 ,down_3 ,output NODE 3 )
TASK( DOIT_4 INIT do_it_code,32000, 100 ,down_3 ,down_4 ,output NODE 4 )
TASK( DOIT_5 INIT do_it_code,32000, 100 ,down_4 ,down_5 ,output NODE 5 )
TASK( DOIT_6 INIT do_it_code,32000, 100 ,down_5 ,down_6 ,output NODE 6 )
TASK( DOIT_7 INIT do_it_code,32000, 100 ,down_6 ,bottom ,output NODE 7 )
GLOBAL_END

```

Figure 3: Generated configuration file.

tions. In most cases, these factors can only be determined by executing the program and analyzing its performance.

The SAMTOP-environment attempts to support all these aspects of the mapping problem. The specification can be used to generate a configuration file which describes the program objects and their mapping to processor nodes. Fixed topological mappings can be generated using the map-operation described above. On the other hand, it is also possible to generate random-mappings and to change the mapping of program objects manually. We are currently exploring the possibility of using the TOPSYS performance-analyzer to measure the performance of applications after such an initial mapping and to use the measurement data to compute an improved mapping function. The idea behind this approach is to use performance data, such as the CPU-time used by a task and the amounts of data transferred between tasks and mailboxes as input for heuristic algorithms which try to balance CPU- and communication loads by calculating an appropriate mapping of objects to processor nodes.

5. The TOPSYS Development Cycle

After having described the SAMTOP design method in detail, let us now take a look at the steps and phases involved in developing a parallel application when using the TOPSYS environment. In our model the development cycle for multiprocessor applications consists of a series of well defined steps which are illustrated in figure 4. The tool environment offers support for each of the phases.

The first step consists of modeling the hardware and software environment of an application and creating a high-level specification of the intended functionality. The high-

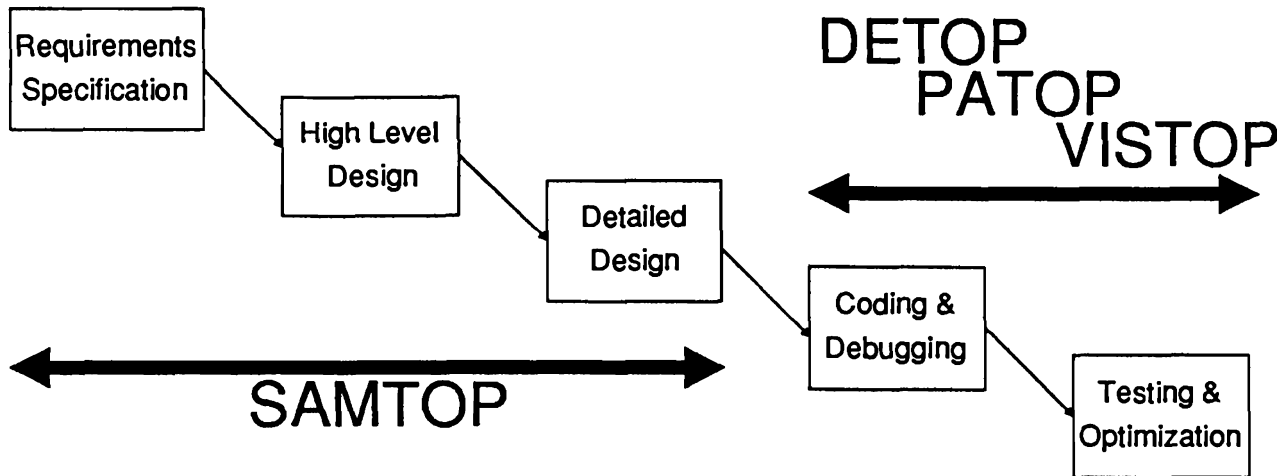


Figure 4: Development cycle in the TOPSYS environment.

level design is then transformed into a detailed design through a series of stepwise refinement steps using the SAMTOP CASE-environment. The facilities for code- and document generation ease the transition to the coding and debugging phase. The TOPSYS environment comprises three interactive tools for use during runtime: a debugger for parallel programs (DETOP), a tool for performance analysis (PATOP) and a visualization tool (VISTOP). All tools use a common monitoring technique based on functionally identical hardware and software monitors on each processing node.

6. Conclusions and Further Work

The SAMTOP CASE-environment offers a new approach to the problem of design and specification of multiprocessor software, in that a standardized and widely used CASE methodology is used as a basis. This method has been adapted and enhanced in order to meet the requirements of the designer of multiprocessor software.

Although our approach supports only multiprocessors with distributed memory its methodology is also applicable to shared memory systems. The CASE tool provides means for specifying data flow by using memory objects instead of mailbox objects. However, with shared memory systems the mapping problem is of minor interest as the scheduling strategies usually lead to a perfect load balance during runtime.

By integrating the design method into our tool-environment we have created a programming environment which offers support for all the steps involved in developing a multiprocessor application. However, the first version of the SAMTOP-environment is only a prototype. The main problem is the lack of a powerful project-database and of version control mechanisms. This makes it very difficult to adapt the design method to our needs in a well-structured way. Current projects include the development of an automatic remapping facility based on performance data acquired by the TOPSYS performance-analyzer.

7. References

1. D.A. Bailey, J.E. Cuny: *Graph Grammar Based Specification of Interconnection Structures for Massively Parallel Computation*, in *Graph Grammars and Their Applications to Computer Science*, Lecture Notes in Computer Science 291, Springer Verlag 1987.
2. T. Bemmerl: *The TOPSYS Architecture*, Proceedings of the CONPAR90-VAPP IV Conference in Zurich, Switzerland, September 1990, Lecture Notes in Computer Science, Nr. 457, H. Burkhardt (ed.), Springer, Berlin, p. 732-743.
3. T. Bemmerl, T. Ludwig: *MMK - A Distributed Operating System Kernel with Integrated Dynamic Loadbalancing*, Proceedings of the CONPAR90-VAPP IV Conference in Zurich, Switzerland, September 1990, Lecture Notes in Computer Science, Nr. 457, H. Burkhardt (ed.), Springer, Berlin, p. 744-755.
4. K. Chen, S. Yan: *An Approach to Verification of Communication in Distributed Computing Systems*, Proceedings of the 9th Inter. Conference on Distributed Computing Systems, June 89, p. 603-610.
5. T. DeMarco: *Structured Analysis and Systems Specification*, Yourdon Press, 1979.
6. S. Eisenbach, L. McLoughlin, C. Sadler: *Data Flow Design as a Visual Programming Language*, ACM Sigsoft Fifth International Workshop on Software Specification and Design, May 1989, p. 281-283.
7. K. Gates, D. Socha: *Programming NCUBES with a Graphical Parallel Programming Environment versus an Extended Sequential Language*, Proceedings of the Second Conference on Hypercube Multiprocessors, 1987, p. 17-27.
8. W.D. Hillis, G.L. Steele: *Data Parallel Algorithms*, Com. of the ACM, December 1986, p. 1170-1183.
9. M.A. Jackson: *Principles of Program Design*, Academic Press, London, 1975.
10. K. Krauss, S. Golden: *A petri-net method for the formal verification of parallel processes*, Proceedings of the IEEE International Conference on Parallel Processing, 1988, p. 157-160.
11. L. Lamport: *Specifying concurrent program modules*, ACM Transactions on Programming Languages and Systems, 5, 1983, p. 190-122.
12. R. Mielke, S. Stoughton, S. Som: *Modeling and performance bounds for concurrent processing*, Proceedings of the 8th Inter. Conference on Distributed Computing Systems, June 88, p. 538-544.
13. H. Muehlenbein, O. Kraemer, F. Limburger, M. Mevenkamp, S. Streitz: *MUPPET - A Programming Environment for Message-Based Multiprocessors*, Parallel Computing 8, 1988, p. 201-221.
14. M. Muehlhaeuser: *Software Engineering for Distributed Applications - The Design Project*, Proceedings of the 10th International Conference on Software Engineering, April 1988, p. 93-101.
15. L.M. Ni, C. King, P. Prins: *Parallel Algorithm Design Considerations for Hypercube Multiprocessors*, Proceedings of the IEEE International Conference on Parallel Processing, 1986, p. 717-720.
16. D. Notkin et al.: *Experiences with Poker*, Proc. of the ACM/SIGPLAN PPEALS, 1988, p. 10-20.
17. T. Pratt: *The Pisces-2 Parallel Programming Environment*, Proceedings of the IEEE International Conference on Parallel Processing, 1987, p. 439-445.
18. J. Purtilo, D.A. Reed, D.C. Grunwald: *Environments for Prototyping Parallel Algorithms*, Proceedings of the IEEE International Conference on Parallel Processing, 1986, p. 431-438.
19. Ready Systems Corp.: *CARDtools for Sun 3.3 User's Guide*, Sunnyvale, 1989.
20. D.T. Ross, K.E. Schomann: *Structured Analysis for Requirements Definition*, IEEE Transactions on Software Engineering 3, no.1, 1977, p. 6-15.
21. S. Sobek, J.C. Browne, M. Azam: *Architecture and Language Independent Parallel Programming - A Feasability Demonstration*, Proc. of the IEEE Inter. Conference on Parallel Processing, 1988, p. 80-83.
22. E. Yourdon, L.L. Constantine: *Structured Design*, Prentice Hall, Englewood Cliffs, N.J. 1976.
23. P. Ward, S. Mellor: *Structured Development for Real Time Systems*, vol. I-III, Yourdon Press 1985.

MULTIPLE FUNCTION MULTIPLE DATA PROCEDURES AND PARALLEL PROGRAMMING

N.N.MIRENKOV, L.A.BULYSHEVA

*Computing Center, Siberian Division
USSR Academy of Sciences,
Novosibirsk 630090, pr. Akad. Lavrentieva, 6*

ABSTRACT

An approach of multifunctional cooperation over multiple data within the framework of one procedure is considered for the development of high efficiency parallel programs. Software tools to support this approach are briefly described. They are special-purpose parallel programming systems which act as sequential program integrators that execute transformations inverse to those of parallelizers. Examples are given of the approach applications to developing library procedures for the VLIW architecture computer.

1. Introduction

A significant amount of research has been done in the area of parallelization, the extraction of parallelism from sequential programs. The classical parallelization approaches are techniques for vector, multiprocessor, and VLIW machines¹⁻⁵. There are many commercial programming systems with automatic extraction of parallelism. Among them especially high-quality code is generated by compaction-based parallelization for synchronous machines of VLIW architectures. Nevertheless, the efficiency analysis of high-performance parallel computers shows that there is a wide gap between their peak performance and their performance at solving real applications. Of course this is attributed to the corresponding systems of parallel programming. Of no less importance, however, is whether the programmer takes into account the peculiarities of application algorithms and computer architecture. But even if he does

take them into account, high efficiency is not always achieved because the inherent parallelism of the application disagrees with the parallelism of the computer. So we suggest making use of integration before parallelization. The gist of the integration is not in making a sequential program parallel but in making it integral with other programs. After integration the new integrated program may be put for a parallelization system. In other words, we suggest a transition to the parallel programming using multiple function multiple data (MFMD) procedures, i.e. the procedures each of which within the framework of one code executes a number of functions over several data sets in an integrated, interrelated mode. Either the functions being computed or the data sets may coincide. In this respect any vector operation can be regarded as a simple and particular case of a MFMD procedure involving identical functions. A more essential MFMD procedure using identical functions is, for instance, execution of a deconvolution algorithm over a few seismotracess. Another particular case of MFMD procedures is computation of a number of the *Bessel* and *Neumann* functions of different orders for a certain x . It uses identical data.

The reasons why MFMD procedure increases the efficiency of parallel computer systems are the following:

- the procedure becomes larger and its inherent parallelism increases, which opens up new possibilities in mapping an algorithm on an architecture;
- intermediate results obtained in computing one function may be used for computing other functions;
- "collective" operations may be employed, i.e. the operations which are executed once but are used for many functions at a time;
- some overheads are reduced due to decreasing both the operating system access and switching from one process to another.

Although these aspects of the MFMD procedure use are rather well-known, we believe that forming a general approach on their basis is very important. To support parallel programming and development of MFMD procedures, programmers need special automatic and interactive tools. First of all, we would like to distinguish the following lines:

- "hand-made" creation of new libraries of highly efficient MFMD

procedures;

- creation of a special-purpose optimization blocks, which would allow detecting in sequential programs not the conventional loop parallelism alone, but also groups of standard procedures and functions, which can be automatically replaced by MFMD procedures;

- development of sequential program integrators which generate a code for executing different sequential programs within the framework of a new integrated program;

- development of integrators which, also within the framework of a new integrated program, generate a code for executing processing of multiple data sets based on the same sequential program.

In this paper the emphasis is on the essence of the MFMD procedure approach and its relation to parallel programming. In addition, software tools which can be considered integrators, are briefly described. Finally, peculiarities of MFMD procedures are illustrated for computing trigonometric and cylindrical functions on the ES-2706 processor with VLIW architecture.

2. MFMD approach

The MFMD approach is based on the following well-known features of the programming process.

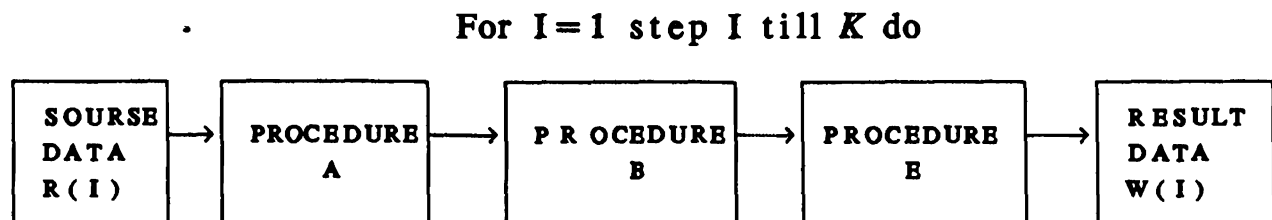


Fig. 1

1. Multi-variant computations are used in numerical experiments, that is the same succession of procedures is employed to execute computation over various data sets. Fig.1 shows a very simple and rather typical computation scheme according to which a portion of source data goes to the input of procedure A, then its output goes to the input of procedure B and so on. After the entire chain of procedures is completed, computation continues with the next portion

of source data.

Making use of the concept of vector processing, here the MFMD approach is implemented by the computation scheme given in Fig.2. Following this scheme, the input of the MFMD procedure A' receives not one but n portions of source data and its output then goes to the input of the MFMD procedure B' and so on. Now there is more inherent parallelism in each of the MFMD procedures. In particular, it becomes easier to use microcode compaction techniques, to vectorize and load various functional units, etc. One time operand fetching may be used n times for computing various functions. "Cooperative" operations of the same kind can also take place in the common loop control, in writing data into the memory and in applying some intermediate results obtained in processing one data set for the remaining sets. Note that generally the program of a MFMD procedure is not a loop repetition of the program of a one-functional procedure nor is it a simple vector version of the course code. It rather represents a new program which executes intercoordinated simultaneous computation of n functions, taking into account peculiar features of algorithms and the architecture of the parallel machine. Operations and microoperations related to processing one kind of data may be shuffled with the operations for processing another kind of data. Our idea is to create tools for automatic generations of procedures A' , B' , ..., E' on the base of A , B , ..., E , respectively.

For $I=1$ step I till K/n do

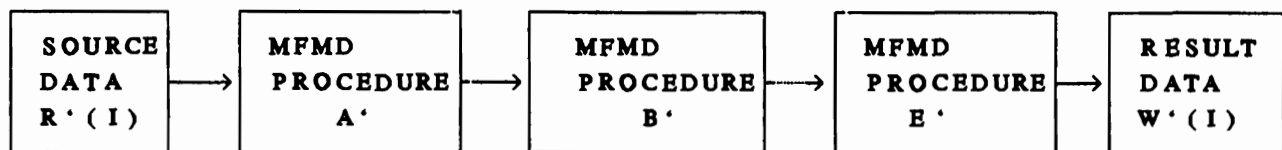


Fig.2

It should be noted in addition that transition to MFMD procedures may provide impetus for using entirely new algorithms such as the algorithms based on table fetching rather than on computations by formulae.

2. Various applications use the intercomplementary and often concurrently executing functions, which brings about an essential change in the total runtime. Fig. 3a displays a computation scheme which

$X := F1(Y, Z)$	$X := F1(X, Z)$
$Y := F2(X)$	$Y := F2(X)$
$A := \text{Cos}(X)$	$R := \text{SinCos}(X, Y)$
$M := F3(A)$	$M := F3(R(1))$
$B := \text{Sin}(X)$	$N := F4(R(2))$
$N := F4(B)$	$P := F5(R(3))$
$C := \text{Cos}(Y)$	
$P := F5(C)$	
$D := \text{Sin}(Y)$	

a)

b)

Fig.3

requires two calls of *sin* and *cos*, respectively. According to the MFMD procedure approach, these computations are automatically converted to the scheme given in Fig.3b, where a four-component vector R : $R(1)=A$, $R(2)=B$, $R(3)=C$, $R(4)=D$, is introduced instead of the variables A, B, C, D , and the MFMD procedure *sincos* of two arguments is applied. The latter computes $\sin(x)$, $\sin(y)$, $\cos(x)$, $\cos(y)$ in the integrated mode. Section 4 describes how to create MFMD procedures of the *sincos* type for the processor ES-2706 and the efficiency of this approach.

3. The use of classical multiprogramming and batch processing provides a better loading of functional units by a pre-selected mix of generally independent tasks. Making use of the batch processing idea, the MFMD procedure approach helps to further increase the efficiency of computations based on closer interrelating and intercoordinating the task mix. This is achieved by transforming the batch programs into the one MFMD procedure program, which takes into account extra knowledge on the integrating programs. Such an integration allows to eliminate (or essentially reduce) the costs of switching from some processes to others, to increase the loading of functional units at

the microlevel, to eliminate repetition of some operations common for different programs, etc.

It is important to note that the MFMD procedure is primarily logical object. It is for realization on any type of parallel computers, not only for MIMD machines. In this respect, parallel programming based on these procedures can be regarded as a general approach which, if provided with appropriate tools, would greatly increase efficiency of parallel data processing. Clearly, different architectures will require additional details and modifications of the approach.

3. Sequential program integration

To support the idea of MFMD procedures we suggest that special-purpose software tools should be developed and used. We call them "sequential program integrators". The gist of the integrator is not in making an "old" sequential program parallel but rather in making it integral with other programs, that is an integrator executes transformations inverse to those of parallelizers. Let $P_1, P_2, \dots, P_N$ be sequential programs and I_i and O_i be input and output data of program P_i , $i=1,2,\dots,N$. Integration of $P_1, P_2, \dots, P_N$ is a process of creation of a new integrated program P , executing required functions over $I=I_1+I_2+\dots+I_N$ and resulting $O=O_1+O_2+\dots+O_N$. We distinguish several types of integration:

- single function/multiple data, with $P_1, P_2, \dots, P_N$ are identical (coincident) and $I_1, I_2, \dots, I_N$ are different;
- multiple functions/single data, with $P_1, P_2, \dots, P_N$ are different and $I_1, I_2, \dots, I_N$ are identical;
- multiple functions/multiple data, with both $P_1, P_2, \dots, P_N$ and $I_1, I_2, \dots, I_N$ are different.

Each of these types in turn can be subdivided into a few variants according to SIMD, MIMD, VLIW or mix architectures of target computers as well as according to their compiler and parallelization systems. So, in one case, program P is a conventional fortran program, in other cases it is a parallel program with explicit vector parallelism, etc.

Here we are going to consider in brief a few integrators developed in the Computing Center of Siberian Division of the USSR Academy of Sciences.

The first example of an integrator which provides execution of sequential program on the parallel SIMD PS-2000 computer is the FORTRAN-PS parallel programming system. This system is a part of the project "High-performance computer system "Siberia" ⁶. Under the programming system FORTRAN-PS a sequential FORTRAN program can simultaneously process a large set of different source data. The system extends the variable dimensions according to the number of processing elements and provides correct execution of if-statements on the basis of the dynamical formation of activation vectors. In other words, these programming facilities support the execution of a system of identical communicating processes in the SIMD mode. In fact, the FORTRAN-PS system generates automatically the vector-parallel program on the base of the integration of identical sequential ones. In addition, this programming system is an efficient aid for taking into account other peculiarities of the PS-2000 architecture.

An example of another type of integrators is the "INYA" parallel programming system, which is also a part of the above-mentioned project. Unlike the first type of integrators, "INYA" requires an additional mastering program, which treats the sequential programs to be integrated as elementary processes. It must be emphasized that a master process can generate various slave processes on the basis of one and the same set of elementary processes. An important feature of created parallel programs is that they may be adapted to the number of processors available at runtime. Using the "INYA" system it is possible to have some influence on the integration level for processes selected to execute on one processor. The reduction of the actions for the process creation as well as the reduction of number of putting and getting the data from main memory of the computer system to local memory of attached processors considerably increase the efficiency. The integrator is oriented to the MIMD system of VLIW processors.

ERA parallel programming system developed under MINIMAX project ⁷ also belongs to the above type of integrators. But the additional

integrating program of this system is a description of information links between elementary processes. The description is used only for the formation of parallel program processes, each of which is executed by an individual processor. Such a new process may integrate a subset of source elementary processes. Division of a set into subsets is aimed at minimizing data exchange among the parallel program processes being created.

Our integrators distinguish between integration for executing the programs within the framework of a single processor parallelism and one for executing the programs under the multiple processor parallelism. The first category involves, as a rule, essential transformations of integrating codes, and the second one comprises not profound but additional integrating or mastering processes. Now we are going to develop software tools to allow programmers to realize the integration before the compiler optimizer work and after it. Matching the results will give the best variant. So, our devise is integration before and after parallelization.

Of course, in some sense Ada, Modula and others compiler systems may be considered integrators. But they support the program integration only on the level of module interface without essential transformations of integrating codes. Besides, these systems have as a rule poor facilities for efficient mapping of the integrated program onto parallelism of different kinds.

4. Examples of developing the MFMD procedures

In order to improve the efficiency of integrators it is important to know various aspects of hand-made development of respective parallel programs. So, in this section we are going to consider some peculiarities of hand-made developing the MFMD procedures for the ES-2706 processor⁸. ES-2706 is FPS AP-190L-like computer with the VLIW architecture. Its instruction word is divided into microoperation fields for different functional units. The maximum number of microoperations which can be simultaneously executed is 10. To attain a higher computation efficiency, one has to overcome the problems of

information dependence of operations, intensive use of a functional unit as well as memory interference and incompatibility of individual microoperations. The MFMD procedure approach simplifies solution of these problems. To support the approach for geophysics applications, we developed the following MFMD procedures:

- *sine* and *cosine* for one argument,
- *sine* and *cosine* for two arguments,
- *tangent* and *cotangent* for one argument,
- *tangent* and *cotangent* for two arguments,
- *complex exponents* for one and two arguments,
- *Bessel* functions of zeroth and first order,
- *Bessel* and *Neumann* functions of zeroth order,
- *Bessel* and *Neumann* functions of the first order,
- *Bessel* and *Neumann* functions of zeroth and first order,
- *Neumann* functions of zeroth and first order.

The peculiarities of implementing these MFMD procedures will be illustrated by integrated computations $\sin(x)$ and $\cos(x)$, *Bessel* functions of zeroth and first order with the analogous *Neumann* functions.

Procedure $\text{sin}\cos(x)$. Algorithms for calculating $\sin(x)$ and $\cos(x)$ usually consist of four steps. When describing them, we will indicate the sources of additional efficiency.

1) Derivation of an argument value to the interval $\{0, \pi/2\}$. Since the functions are periodic, the reduction to a standard interval allows the decrease of the argument and hence, the decrease of the number of terms in the series. The absolute value of the argument is multiplied by the scaling coefficient $2/\pi$ and integer and fractional parts, respectively I and F are distinguished. This reduction is performed only one time for the two functions at once.

2) Calculation of polynomials. After the reduction both functions are calculated by one formula, but for the two arguments F and $1-F$:

$$\sin(F) = AF + F^3(B + CF^2) + F^7(D + EF^2),$$

$$\sin(1-F) = A(1-F) + (1-F)^3(B + C(1-F)^2) + (1-F)^7(D + E(1-F)^2),$$

where A, B, C, D, E are known coefficients. Simultaneous calculation of

two polynomials provides more effective load of functional units of the processor.

3) Identification of functions. If I is even, then the desired $\sin(x)$ is calculated through $\sin(F)$, and $\cos(x)$ through $\sin(1-F)$. If I is odd $\sin(x)$ is calculated through $\sin(1-F)$ and $\cos(x)$ through $\sin(F)$. This verification is made only once in the calculation of two functions.

4) Determination of signs of functions. Signs of functions are determined from the known formulae depending on parity or oddness of the values I , $I/2$ and the sign of x . For example, if $x \geq 0$, I is even and $I/2$ is even as well, then

$$\sin(x) = \sin(F), \cos(x) = \sin(1-F)$$

if $I/2$ is odd, then

$$\sin(x) = -\sin(F), \cos(x) = -\sin(1-F).$$

If $x \geq 0$, but I is odd, then at even $I/2$

$$\sin(x) = \sin(1-F), \cos(x) = -\sin(F)$$

and at odd $I/2$

$$\sin(x) = -\sin(1-F), \cos(x) = \sin(F).$$

The described operations are also executed only once and this reduces computational costs. The given algorithm realized as the MFMD procedure allows 1.83-1.89 speed-up of calculation of the functions as compared to their sequential calculation by microprograms from the ES-2706 standard library. Similarly, the MFMD procedure $\text{sincos}(x,y)$ yields 2.56-2.62 speed-up.

Procedure BBNN(x). For calculation of the *Bessel* and *Neumann* functions of zeroth and first orders, the following algorithm was selected⁹. The interval of variation of x was partitioned into two subintervals $0 \leq x < 8$ and $x \geq 8$. For the subintervals, the following expressions were made use of. Let us show them on the example of zeroth order:

$$\begin{aligned} B_0(x) &\approx \text{const} \times P_n^B(x), \\ N_0(x) &\approx \text{const} \ln(x) B_0(x) + \text{const} \times P_n^N(x), \end{aligned} \tag{1}$$

for $0 \leq x < 8$; $P_n^B(x)$ and $P_n^N(x)$ are polynomials of the order n .

For $x \geq 8$ analogue (1) is of the form:

$$B_0(x) \approx F(Pn(x)\cos(\varphi) - Qn(x)\sin(\varphi)),$$

$$N_0(x) \approx F(Pn(x)\sin(\varphi) + Qn(x)\cos(\varphi)),$$

where $F = \text{const}/\text{SQRT}(x)$, $Pn(x)$ and $Qn(x)$ are polynomials of the order n , $\varphi = x - \text{const}$. Similar representations are written for $B_1(x)$ and $N_1(x)$. The polynomials used here are of the eighth order.

We will now dwell on the main peculiarities of implementation. At $0 < x < 8$ $\text{Ln}(x)$ is computed once for two *Neumann* functions, which provides a four microsec economy. Computation of a few polynomials and expressions of the type (1) provides effective loading the pipeline units. Values of the *Bessel* functions of zeroth and first order are immediately used for computing the *Neumann* functions. $\sin(\varphi)$, $\cos(\varphi)$, $\sin(\varphi')$, $\cos(\varphi')$ are computed by the MFMD procedure $\text{sincos}(\varphi, \varphi')$, which gives a local 2.5-2.6 speed-up. $\text{SQRT}(x)$, $\text{const}/\text{SQRT}(x)$ and other similar terms are computed once for four functions. This provides the economy of 33.4 microsec.

Speed-up due to the use of MFMD procedures for cylindrical functions are shown in Fig.4 with the following notations: BB is MFMD procedure

MFMD Procedures	Speed - up factors
BB	1.36 + 1.6
NN	1.16 + 1.36
BN0	1.67 + 1.99
BN1	1.53 + 1.98
BBNN	1.85 + 2.71

Fig.4

for computing the *Bessel* functions of the zeroth and first orders, NN is MFMD procedure for computing the *Neumann* functions of the same orders, BN0 and BN1 are MFMD procedures computing the *Bessel* and *Neumann* functions of zeroth and first order, respectively, and BBNN is the MFMD procedure already familiar to us which is used for computing

both functions of both orders.

5. Conclusion

Multiple function multiple data procedures and computer-aided systems for sequential program integration open up some new prospects in the development of efficient parallel programs for processors and systems of various parallel architectures. Being in some sense a further extent of assemble programming ideas and compaction compilers, they may help to solve the problem of program portability and to make universal parallel architectures. Naturally, the efficiency of this approach will differ for various applications and architectures. That is why a promising way for developing universal parallel computer systems is the creation of computers based on subsystems specialized as to the types of parallelism and applications.

6. References

1. J.R.Allen and K.Kennedy, Automatic translation of Fortran programs to vector form, in: *Kai Hwang, ed., Supercomputers: Design and Applications* (IEEE Computer Society Press, Silver Spring, MD, 1982) pp. 188-203.
2. J.A.Fisher and J.J.O'Donnell, VLIW machines: multiprocessors we can actually program, in: *Proc.CompCon Spring 84* (IEEE Computer Society, 1984) pp. 299-305.
3. A.Nicolau, Uniform parallelism exploitation in ordinary programs, in: *Proc. 1985 Internat.Conf. on Parallel Processing* (1985) pp. 614-618.
4. R.Cytron, Doacross: beyond vectorization for multiprocessors, in: *Proc.1986 Internat. Conf. on Parallel Processing* (1986) pp. 836-844.
5. A.Aiken, Compaction-based parallelization, *Ph.D. thesis*, Cornell, Department of Computer Science, Technical Report No.88-922, 1988.
6. N.N.Mirenkov. High-performance computer system "Siberia". *Lecture Notes in Computer Science*, 457. CONPAR-90/VAPP-IY. Ed. H.Burkhart. 1990, pp.806-815.
7. N.N.Mirenkov. *Parallel programming for multimodule computer systems*. M, Radio i sviaz, 1989, 320 p. (in Russian).
8. S.Markov and others. High-performance computer complex. Strategy and evolution. *Supercomputer*, 31, YI-3, 1989, pp. 12-23.
9. L.A.Bulysheva, V.D.Elinov, N.N.Mirenkov. *Computation of cylindrical functions by the MFMD procedures on array processors*. Novosibirsk, 1990, Preprint/USSR Academy of Sciences, Siberian Devision, 919, 14p. (in Russian).

A TOOL TO DISPLAY HIERARCHICAL ACYCLIC DATAFLOW GRAPHS

SRDJAN MITROVIC and STEPHAN MURER
Institut für Technische Informatik und Kommunikationsnetze
Eidgenössische Technische Hochschule
CH-8092 Zürich
Switzerland

ABSTRACT

In this paper, the class of hierarchical acyclic dataflow graphs is defined. The data structures and the different phases of an algorithm to lay out this class of graphs are explained. The algorithm proceeds in the following three phases: Initial placement of the graph nodes on topological levels, reordering of the nodes in the levels to minimize edge crossings and finally layout of the connections between the levels in Manhattan Geometry.

One issue while designing the algorithm was the trade-off between layout quality and runtime. It is shown that even large graphs can be laid out well by the presented technique in a time acceptable for interactive environments.

Implemented on an Apple Macintosh computer, a browser tool displaying SISAL sources and the corresponding dataflow graphs is described.

1. Introduction

On one hand dataflow graphs are used in a number of parallel processing research projects as intermediate code between a functional (or single-assignment) language and a parallel machine. Various kinds of operations like partitioning into parallel processes, code optimizations, transformations, resource allocation, code generation, etc. are active research topics on top of the sound theoretical foundations of graphs. On the other hand, in the age of supercomputers, visualization of results is becoming a more and more important step in the research process: "A good picture may tell you more than a thousand words."

In the ADaM-Project^{3, 4, 5} it is our goal to jointly develop software tools and hardware for a coarse grain dataflow machine. As part of that project we are developing an integrated software environment for our machine on a workstation with good graphical capabilities. Because we use single-assignment languages to program our machine and we have no longer been willing to debug the compiler by looking at a textual form of dataflow graphs, it was time to create a graphical display tool for dataflow graphs.

The tool displays interactively dataflow graphs together with the corresponding source text. It keeps links to the source text, in order to synchronize the display of text and graphs. For any position in the source text, the corresponding graph is displayed on a simple mouse click and vice versa. The displayed dataflow graphs are hierarchic, i. e. nodes can contain a number of subgraphs. The tool allows the user to browse through the graph hierarchies. The content of the graph view can easily be copied and pasted into a standard text or graphics document on the workstation for documentation purposes. Of course, the user can scroll through graphs of arbitrary size. Since graphs can get fairly complicated, a visual help is built in, highlighting one edge or all nodes adjacent to the selected one. In addition to these operations, the user can open a specific information window for each node or edge through which additional information can be displayed.

Because the graphs are fairly complex and generated automatically by the compiler, user-guided layout was not considered. Thus, the main problem was to find a good layout algorithm, i. e. an algorithm that places nodes and edges in order to visualize the information of the graph as succinctly as possible. Several factors influence the readability of graphs. For example, the layout algorithm should place adjacent nodes close to each other and should minimize the number of edge crossings and the length of edges.

There exist a number of algorithms to lay out graphs used in VLSI design or PCB routing (see Burstein and Pelavin² for example). However, most of these are very time and resource consuming and some have even exponential time complexity. Since dataflow graphs can have hundreds of nodes and edges in real world applications, these algorithms do not fit our need for quick response in an interactive environment. We had to find an algorithm that produces reasonably looking graphs at reasonable cost. Thus, some layout quality had to be sacrificed for run time performance. We have found an algorithm basically fitting our needs in⁶.

The remainder of this paper defines more precisely the class of graphs under consideration. Then the used layout data structure and algorithm will be described in detail and finally some implementation characteristics will be given.

2. Hierarchical Dataflow Graphs

We can define a hierarchical dataflow graph as the following extension of the common definition of directed acyclic graphs in graph theory:

$$G = (N, E, T) \quad (2.1)$$

In Eq. 2.1, N and E denote the set of *nodes* and *edges* in the graph G , respectively. $T \in N$ denotes an *environment node* representing the environment of the graph. Thus, each edge coming from the outside of the graph comes out of the environment node, and each edge leaving the graph flows into the environment node. The environment node was only introduced to simplify the following definitions. It is not really drawn in dataflow graphs. Each edge connects a certain *outport* of a node with an *inport* of an other node in the same graph:

$$E \supset N \times Z^+ \times N \times Z^+ \quad (2.2)$$

Inports and outputs of a node must be identified with a port number because the order of operands or results matters in arithmetic operators, which the nodes represent in dataflow graphs. In contrast to regular graphs, edges connect specific ports of nodes rather than the nodes themselves. In this definition, ports are numbered by positive integer numbers. There are also special *constant edges* conceptually sharing any output of T as source, but having no connection to an output in the graphical representation.

Another restriction stemming from the fact that dataflow graphs are deterministic is that only one edge is connected to an inport. In other words: The fan-in of ports is always one, while the fan-out can be any number including zero. This fan-in restriction is stated formally in Eq. 2.3:

$$\forall e_1, e_2 \in E: e_1 \neq e_2 \wedge e_1 = (n_1, z_1, n_1', z_1') \wedge e_2 = (n_2, z_2, n_2', z_2') \Rightarrow (n_1', z_1') \neq (n_2', z_2') \quad (2.3)$$

The graphs without the environment node T are acyclic and rooted in T. Each node can be constructed of a number of subgraphs:

$$N = \{G\} \quad (2.4)$$

In this case we are talking about *compound nodes*: The definition shows the recursive nature of hierarchical dataflow graphs. Graphs consist of nodes, nodes consist of graphs and so on. The root structure, however, is a graph. There are special rules concerning the relations between the ports of a compound node and the ports of their subgraphs. But they are beyond the scope of this paper.

As an example for graphs like this, we can take the intermediate form (IF1⁸) produced by the SISAL compiler frontend¹⁰. The frontend consists of the parser and various optimizers. In IF1, nodes represent operations and edges represent data dependencies or data communication between the operations. Every edge represents either an atomic or a structured value. Compound nodes stand for control flow constructs like loops or alternatives. For example in a loop compound node, its four subgraphs represent the four elements of the loop: initialization, test of the loop condition, loop body and returns clause of the loop expression. An example dataflow graph will be shown later in this paper, when details of the layout algorithm are explained. However, a full definition of IF1 and SISAL is well beyond the scope of this paper; the reader is referred to various reports^{7,8,10,11}.

3. Data structures in the graph display algorithm

The data structures of the graph-display are derived from the dataflow graph structure as described in chapter 2. We call this new structure a graph-grid, with all coordinates measured in grid-points. The graph-grid is a two-dimensional structure containing a list of horizontal levels. It corresponds to the environment node of the

dataflow graph. The top level contains the argument ports, i.e. the outputs of the environment node, and the bottom level contains the result ports, i.e. the inputs of the environment node. Every level has a width and a height, the sum of the heights is the height and the maximal level-width is the width of the graph-grid.

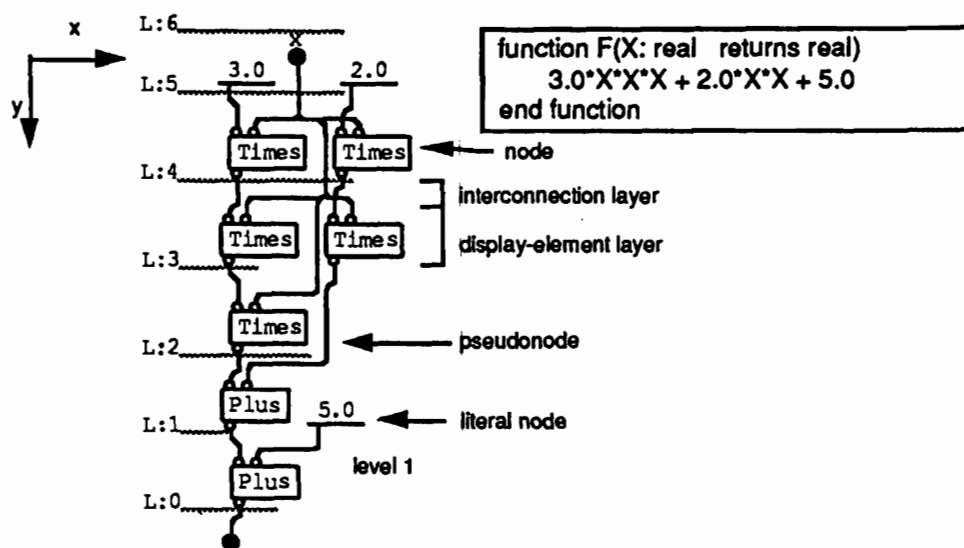


Fig. 1: The structure of the graph-grid

Every level holds a list of display elements and a list of connections. A display element can be a node, a pseudonode or a literal-node. The nodes correspond directly to the dataflow simple- and compound-nodes. Literal nodes are built out of constant edges while pseudonodes do not have a correspondent. The necessity of pseudonodes will be discussed later. The height of a level consists of the constant height of the display-element layer, plus the variable height of the interconnection layer. Usually the source and the destination of an edge are not vertically aligned, therefore we use the interconnection layer for the horizontal shift of the connection lines. More details about this layer can be found in the following chapter about Manhattan Geometry. The connections specify the source, the destination and the floor, i.e. x-coordinate of the horizontal line in the interconnection layer, and connect only display elements in adjacent levels. A dataflow edge across multiple levels is split into multiple connections connecting to pseudonodes instead of crossing the levels directly.

The display elements (DEs) are inserted into the levels. They have the x-coordinate in the level and their y-coordinate is derived from the level position in the grid. Every dataflow node builds a distinct DE which is inserted into a specific level. All result- and argument-ports of a graph are placed as DEs in the bottom respectively in the top level of the graph-grid. Every constant edge to a node at level k will be inserted as a literal node in level $k+1$. A dataflow edge crossing several levels will be split by inserting pseudonodes at crossed levels. An edge between the level k and the level $k+2$ is transformed into a connection from the source element at level k to a new pseudonode at level $k+1$. At level $k+1$, a connection is generated connecting the pseudonode with the destination at level $k+2$. A pseudonode has no height and is used as a space holder for the crossing edge.

4. The algorithm steps

The graph-display algorithm can be split into three phases. The first phase encompasses 2 passes and is responsible for simple conflict-free layout of display elements and connections. The second phase consists of a variable number of passes, some of them optional, improving the readability of the graph display.

```

phase 1: pass 1, laying out nodes
          pass 2, laying out pseudonodes, connections, literals
phase 2: sorting display elements
          optional centering or gap-insertion
          flattening level-crossing edges
phase 3: assigning horizontal lines to connections

```

The last phase assigns horizontal lines in the interconnection layer to the connections. Phase one and two will be discussed in chapter 4.1 and phase 3 is described in chapter 4.2

4.1. Placement of Nodes

Taking a SISAL function, we visualize in two figures the implemented phases; all the pictures are generated by the display-program described in chapter 6. The first pass is the insertion of nodes into levels in the form of display elements (DEs). Into which level a node will be inserted is defined by a topological sort. The sort criteria can either be depth-first (minimal distance from destinations), breadth-first (minimal distance from source elements) or some mixture of both. After pass 1, the next step is to create connections between the display elements and to insert pseudonodes and literals as display-elements. This is the task of pass 2. Because a connection links elements in adjacent levels only, every edge which crosses several levels will be split into connections to and from pseudonodes. The second pass is the last one to insert display-elements and connections into levels; all following passes are only improving the readability by rearranging the horizontal positions.

An optional pass centers the display-elements while the width and the height of the graph-grid remain constant. Note that after pass 2, the pseudonodes and the literals, being the last elements to be inserted, become the rightmost ones. Compared to this order we improve the display quality much by sorting the DEs according to the position of their sources. The second phase starts with the sorting pass. We can either average the x-positions of all destinations of a display-element or left justify the graph in that only the position of the leftmost destination is relevant. Namely, when not centered, the graph has the tendency to build up in a diagonal way from bottom-left point to the top-right point. We can counteract this by having the elements sorted only by the leftmost destination element. A similar approach using barycenters is proposed by Sugiyama et al.⁹.

SISAL Function:

```
function Trap(L,R: real returns real)
  (R-L) * (F(L) + F(R))/2.0
end function
```

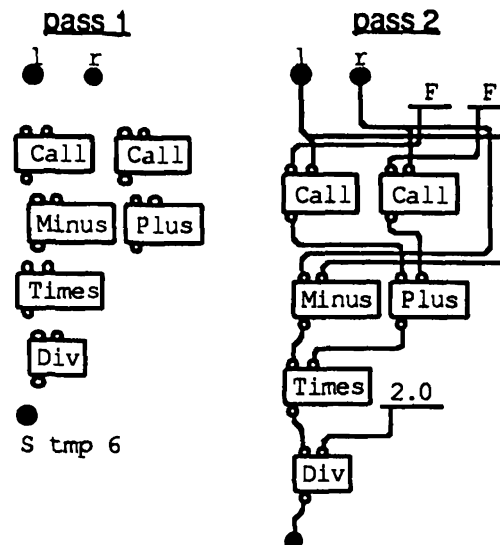


Fig. 2: The first phase

In figure 3, a simple sort using the x-coordinates of the destination elements shows the improvement. Until here the display elements are placed as close as possible to each other; there are no wide gaps between elements. In this way we have a minimal width of the graph-grid, which however, produces more jagged connections between elements. We can straighten these lines by inserting gaps between the DEs. Because all elements at odd levels occupy only odd grid-points, and at even levels even coordinates, respectively, level-crossing edges must be jagged. This jaggedness can be taken out by letting the pseudonodes evaluate the x coordinates of its source and destination connections and eventually moving itself by one grid-point to the left. The last graph in figure 3 shows the final display after the elements have been sorted and the graph-grid has been stretched in width.

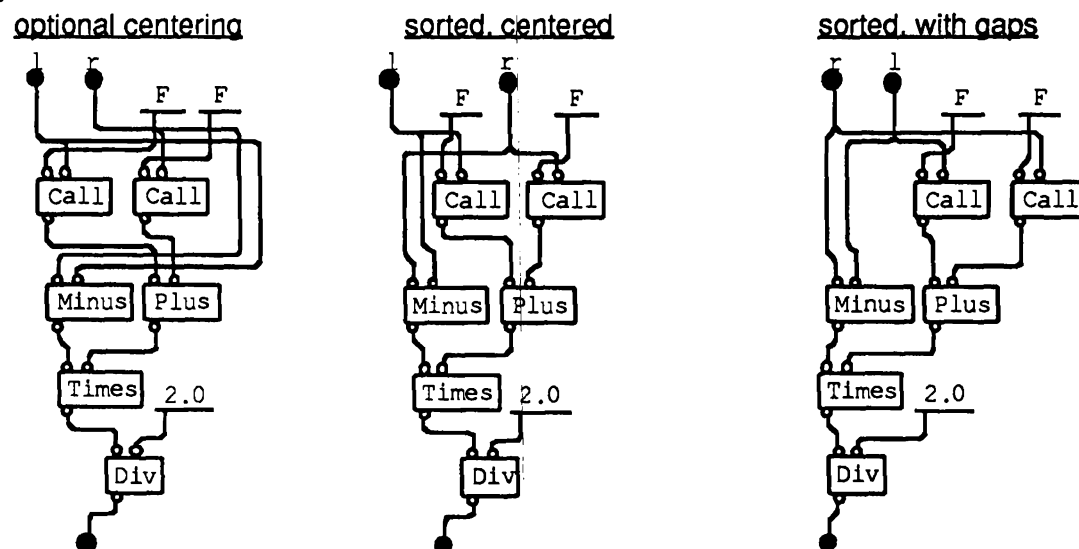


Fig. 3: The second phase

This solution is not very satisfying, because we have connections that cross each other twice, although only one crossing is necessary in the worst case. Assuming a fixed order of ports, one crossing is inevitable if the x -coordinates of the inputs and the outputs have different order. Is there a layout strategy solving the problem of double crossing?

We can work it out by first sorting the connections of a level by ascending x -coordinates of the outputs. Now we assign floors to the connection lines in this order applying the following simple rule: If the connection goes to the right, i. e. the output x -coordinate is less than the input x -coordinate, then the lowest free floor is assigned, and in the reverse case the highest free floor is assigned. The following illustration shows the same example as above routed by the improved algorithm.

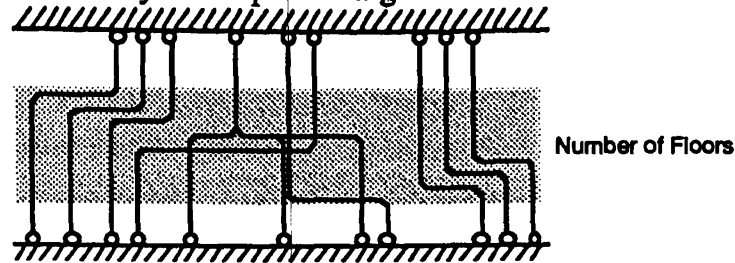


Fig. 6: Better allocation of floors

Although no connections cross each other twice this solution is not yet fully satisfactory, because in many cases the connection lines of two connections could be laid out on the same floor without double crossing or ambiguity. This would decrease the height of the display area for the same graph without sacrificing readability. Thus, the same information could be shown on a smaller area which is of course a goal to strive for. A simple modification of the above algorithm improves the situation significantly.

For each floor, the information up to which x -coordinate the floor has been filled by horizontal lines is memorized. Instead of assigning floors for rightgoing connections from the bottom and leftgoing connections from the top, we start at an imaginary center floor (0). We assign positive floors to rightgoing connections and negative floors to leftgoing connections in the following manner: Start at the highest floor for rightgoing connections and at the lowest floor for leftgoing connections. If the horizontal line of a new connection does not interfere with the horizontal lines already placed on this floor, we try to place it on the next floor closer to the center. As soon as an interfering floor is found, i. e. the maximum x -coordinate of this floor lies inside the x -range of the new connection, the last found free floor is assigned to the connection. The idea is to place each connection as close to the center as possible.

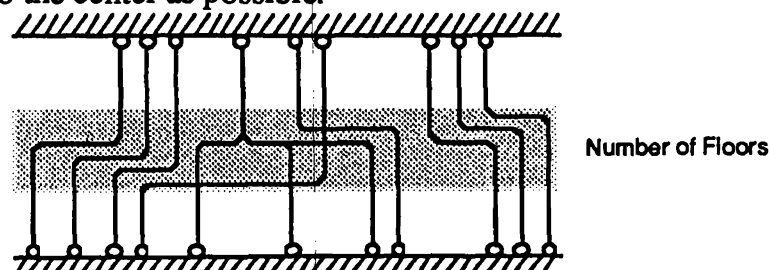


Fig. 7: Even better allocation of floors

As the example shows, this method is not optimal in the sense that it does not minimize the number of floors under the conditions stated above (no ambiguity and no double crossing). We could reduce this example from 6 to 4 floors by placing the fourth and fifth connection one floor lower and the three rightmost connections two floors lower. However, an optimal method would require some kind of backtracking mechanism leading to nonpolynomial time complexity, a property highly undesirable in an interactive environment. The experiments and extensive use showed that the current solution is a good compromise between time requirements and layout quality.

5. Measurements

The display quality of a graph is difficult to measure in quantitative units. We can state that the connectivity inside the graph should be understandable and the display area as small as possible. Therefore we assume that a smaller total sum of connection lengths improves the visibility, meaning shorter and clearer connections and a smaller display area. In figure 8, four larger SISAL functions have been used to display graphs. The implemented program is described in the following chapter. The description of the language SISAL is out of the scope of this paper, most information is provided in other publications ^{7,10,11}. We measured the sum of connection lengths in grid-points after four different phase-marks. All passes and phases have been described in chapter 4.

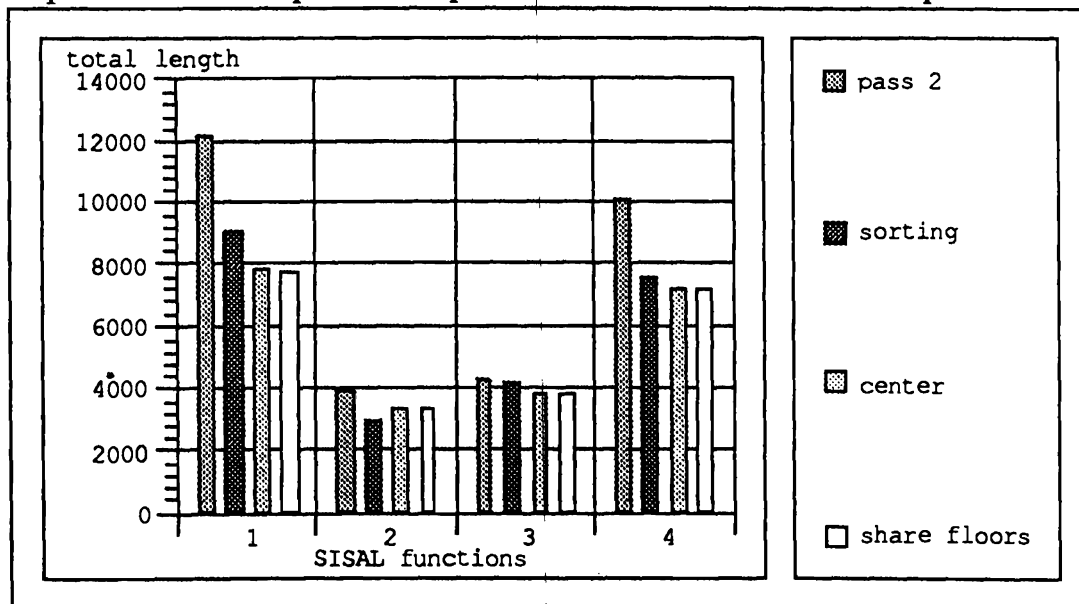


Fig. 8: Total connection length compared to various algorithm passes

The first mark is pass two. It is the minimal and most simple method to generate readable graph-displays, the graph is left justified. After the second mark we sort the graph according to the destination element positions. Depending on the function used we attain even 25% shorter total connection lengths. Centering the graph in the third stage does not always shorten the total connection length but usually improves the visual quality. The same can be said about the sharing of floor levels, which does significantly

decrease the height of the display, but not the connection length. Function 1 produces a graph with an area of 171×277 grid points. Using the smallest possible grid size of 4 points this draws a display of 684×1108 screen pixels. The generated graph-grid has 139 connections and a minimal build-up time of $1/3$ of a second on a Macintosh IIfx. The build-up time is the time it takes to generate the display-structure out of a dataflow-graph. The draw time depends on the visible part of the display and varies between $1/4$ and $1/3$ of a second. The draw-time is always longer than the build-up time if we make the complete graph visible. The visible region is defined by user through resizing and scrolling the display window.

It should be noted that the algorithms have not yet been tuned for speed. Also, because of the complex memory management, the Object Modula-2 based algorithms are slower compared to the traditional Modula-2 programs. Nevertheless, using an appropriate approach, a high level of interactivity even on the larger graphs has already been realized

6. Implementation issues

In the ADAM project we are developing two different environments for programming parallel computers. One is based on the in-house designed single-assignment language MFL. This environment includes the compiler, code-generator and the graph-display. The other environment is using a SISAL compiler, developed at Lawrence Livermore Labs and Colorado State University, Fort Collins ⁷. Only the latter is described below. Both programs have similar user interfaces, but the MFL environment is more complex and its description is beyond the scope of this paper. The programs have been written in Object Modula-2 with the user-interface based on the object-oriented application framework MacApp®¹. The programs are running on Apple Macintosh computers with at least 1Mb of memory.

After loading the SISAL source file into program, the corresponding IF1 file must be parsed ⁸. The IF1 file is generated by the SISAL compiler and represents the SISAL functions as dataflow graphs. The main window of the graph-display is split into two halves. On the left side the source text is displayed and on the right side the dataflow graph. To display the dataflow graph of a function, we select the corresponding function in the source-window and activate the display command. The proportions of the sub-windows can be changed as well as the size of the whole window. Invisible parts of a graph can be scrolled into view and the graph or the text can be copied to some commercial text-processing tools. In figure 9 we see a compound node named 'IfThenElse'. As mentioned in chapter 2, this node holds a list of subgraphs. When double-clicking this node, we can browse into this node, viewing a list of graphs. Every time the graph-display is changed, a synchronization with the text window is activated through highlighting the corresponding part of the function in the source window.

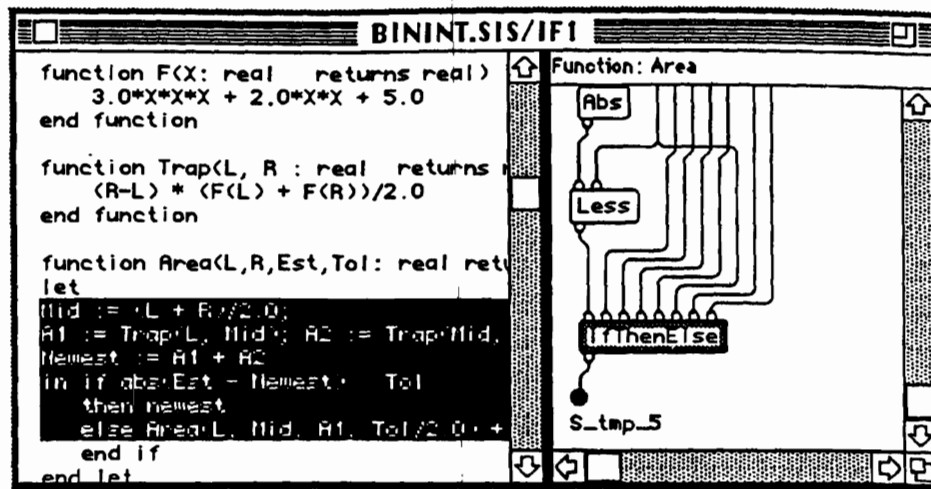


Fig. 9: main window of IF1 display program

Every edge can be selected separately or in groups, making it easier to track the source and the destination of an edge in large graphs. We can also get further information on nodes by selecting them. There are actually no restrictions to the size of the graphs that can be displayed. The only limiting factor is the memory available on the workstation.

The IF1 display program has been used at ETH and worldwide at institutions working with the SISAL programming language. The speed of the display algorithm and the easy and intuitive user interface make it a valuable tool.

7. Acknowledgements

The authors wish to thank all members of the ADAM research group, which in long, fruitful discussions helped to develop the ideas presented in this paper, especially Andi Bomatter, a graduate student who has implemented a very first version of the graph display tool, and Prof. Albert Kündig for his help in putting this paper into its final form.

Part of the work reported herein has been supported by grant no. 2.309-0.86 of the "Schweizerischer Nationalfonds".

8. References

1. Apple Computer Inc., *MacApp® 2.0 General Reference*, (1990)
2. M. Burstein and R. Pelavin, *Hierarchical Wire Routing* in *IEEE Trans. Computer-aided Design of Integrated Circuits and Syst. CAD-2* (1983), p. 235-246.
3. O. Maquelin, *ADAM: A Coarse-Grain Dataflow Architecture that Addresses the Load Balancing and Throttling problems* in *Proceedings of CONPAR '90* (Springer, 1990).
4. S. Mitrovic, et al. *A Distributed Memory Multiprocessor Based on the Dataflow Synchronization* in *Proceedings of International Phoenix Conference on Computers and Communication*, (1990).

5. S. Murer, *A Latency Tolerant Code Generation Algorithm for a Coarse Grain Dataflow Machine* in *Proceedings of CONPAR '90* (Springer, 1990).
6. L. A. Rowe et al. *A Browser for Directed Graphs* in *Software - Practice and Experience* Vol. 17 (1).
7. R. Simpson, *SISAL Compiler user manual*. (Lawrence Livermore National Laboratory Report, September 1986).
8. S. Skedzielewski and J. Glauert, *An Intermediate Form for Applicative Languages* (Lawrence Livermore National Laboratory Report, 1984).
9. K. Sugiyama, S. Tagawa and M. Toda, *Methods for Visual Understanding of Hierarchical System Structures* in *IEEE Transactions on Systems, Man, and Cybernetics* (February 1981).
10. J. T. Feo, D. C. Cann and R. Oldehoeft, *A Report on the Sisal Language Project* (Lawrence Livermore National Laboratory Report 1990).
11. D. C. Cann and J. Feo, *SISAL versus FORTRAN: A comparison using the Livermore Loops* (Lawrence Livermore National Laboratory Report 1990).

ASSEMBLE PARALLEL PROGRAMMING SYSTEM INYA

V.A. ANISIMOV, V.E. MALYSHKIN

*Computing Center, Siberian Division, the USSR Academy of Sciences,
630090, Novosibirsk, USSR*

ABSTRACT

The parallel programming system (PPS) INYA is described. It enables one to develop the applied parallel program which can be effectively executed on specially defined large-block linear hierarchical multicomputer systems (MCS). First of all these MCSs are intended for execution of the regular mass computations which occur for example on solution of problems of signal and image processing. The INYA integrates the software of all MCS components, supports the simple technology for problem parallelizing, provides the applied parallel program portable over all linear hierarchical MCSs. Moreover, INYA provides the tuning of parallel program on available resources of the specific installation and effective sequential program parallelizing. The INYA was developed for SIBERIA MCS, which consists of 3 ES-1066 computers, subsystem of 8 attached processors ES-2706, processors ES-2720 and PS-2000/2100.

Now the high performance multicomputer systems (MCS) can be created on the basis of large computer block including Intel, Motorola, Bit microprocessors, FPS attached processors, associative processors and many others as elements. Each of these processors is a computer, they can be transferred the data and the program and they can work independently. There are additional units: disk, bulk storage, buses, input/output processors, which can also be used to create the MCS. Such MCS are named large block MCS.

The large block MCS has the following advantages:

a) It is possible to realize the fine and coarse grain parallelism on the united resources of the MCS. Fine grain computations are performed on the processor element (PE), coarse grain computations are performed on the resources of the whole MCS. It is necessary to achieve a high performance by solving the complex problem.

b) The MCS can be created very readily as its PEs are included in the MCS with their software. Thus it is necessary to develop the integrating software.

c) The applied software to execute on MCS can be modified simply enough, as it is evolved, but not rebuilt.

However to make use of the above advantages is not easy. The basic difficulty is the vast diversity of possible MCS's and algorithm's structures and they do not always correspond each other. It slumps the MCS performance.

Here the parallel programming system (PPS) INYA is described.^{1,2} It enables one to develop the applied parallel program which can be effectively executed on specially defined linear hierarchical MCS. First of all these MCSs are intended for execution of the regular mass computations which occur for example on solution of problems of signal and image processing.

The INYA integrates the software of all MCS components, supports the simple technology for problem parallelizing, provides the applied parallel program portable over all linear hierarchical MCSs. Moreover, INYA provides the tuning of parallel program on available resources of the specific installation and effective sequential program parallelizing.

The INYA language includes the facilities for assembling of the applied program out of ready-made modules, to define the system of interacting processes, to synchronize them. These facilities can be realized in every sequential programming language. The INYA was developed for SIBERIA MCS, which was created in Computing Centre. It consists of 3 ES-1066 computers, subsystem of 8 attached processors ES-2706, processors ES-2720 and PS-2000/2100.

1. Linear hierarchical MCS

Leaving out the detailed definition of the linear hierarchical MCSs we shall consider a particular example (Fig.1).³ The basic linear hierarchical MCS properties are as follows:

a) There is a central host computer, it manages the processors

elements (assigns the task to the PE, transfers the data and program into it, gets the results, synchronizes the computation). The host computer has the complete set of peripheral devices and well-developed external communications.

b) The slave PEs are connected with the host computer through radial communications for transferring data, programs and control signals. Similar PE's constitute a subsystem, there can be several subsystems of different PE.

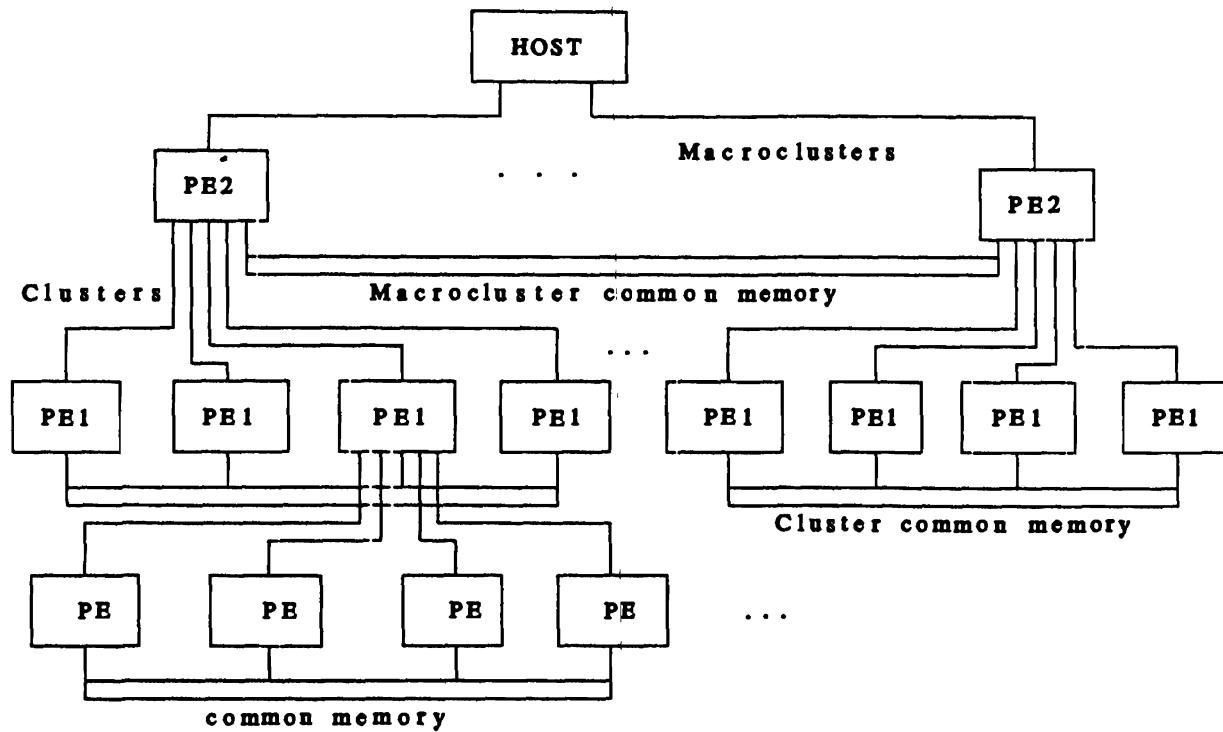


Fig.1

c) The PEs of subsystem are connected through the interface in order to transfer the data bypassing the host computer. The channels, bus, network, common bulk storage can be used. The interface always provides the interaction between PE_i and PE_{i-1} and PE_{i+1} .

d) Every PE in its turn can be the complex processor, in this case it must consist of a host computer and slave PEs, connected by the interface.

e) Any other additional connections and devices can be used in MCS.

The INYA applied program defines the system of parallel

interacting processes. These processes are linearly ordered and there exists such a finite integer number $m, m > 0$, that if the process p_i and p_j interact, then $p_j \in \{p_{i-m}, \dots, p_i, \dots, p_{i+m}\}$. It is linear representation of algorithm, or simply *linear algorithm*.

Evidently, there are not many algorithm which can be represented in this form, but regular mass algorithm can be easily transformed to this representation.⁴ Thus if "n" PEs are available, $n \geq 2m+1$, then they are distributed in such a way: the process p_i is assigned to pr_k processor, the process p_{i+1} on pr_{k+1} , but if $k = n$, then the process p_{i+1} is assigned to pr_1 , when it is idle (the *rolling* of processors) etc. Evidently the program will be effectively executed and it will use all "n" PEs. If for example interface between PEs is a bus, then the process can be assigned to any idle processor as any interactions can be realized also in this case. Both these strategies of PE's distribution actually exist in INYA.

2. Assemble parallel programming

The INYA programming has following peculiarities:

1. The applied program is assembled out of ready-made modules in the same way as a MCS is assembled out of PEs. The PE procedure is developed using its software, and program structure is described using the INYA facilities. Certainly, there must exist the PE to execute every module of program.

2. The MCS has linear hierarchical structure, and so the program has the same structure (unfolding in the similar structure during execution).

3. The selected assignment algorithm for processes enables dynamically tuning on the available PEs. The tuning to interface between PEs is provided by statements, which describe the interaction. These statements are realized differently for different interfaces. Programmer does not need to think about correct interaction execution, he must define them only.

4. The INYA applied program is developed in two stages. At the first stage the modules for PEs are preferred and the algorithm is

reduced to linear representation. At the second stage the whole applied program is assembled, its structure is described.

5. The PPS INYA can be realized similarly for all the linear hierarchical MCS without any modifications of INYA language. For every MCS assembled out of new components the new version of PPS INYA can be readily developed and the applied programmer will always work in the same environment.

3. The PPS INYA components

The PPS INYA consists of the following components:

1. The INYA parallel programming language is an extension of Fortran. INYA statements perform generation, execution and interaction of the parallel processes. These processes execute on the same level or the lower adjacent level of the MCS.

2. Error subsystem analyses abort of the process, outputs error diagnostics and message about interrupted parallel processes.

3. Information subsystem gives required information about language and the PPS INYA.

4. Preprocessor transforms the source INYA code into the Fortran code of the host computer. Preprocessor provides the possibility to use the different Fortran versions. Moreover the preprocessor can be used when it is necessary to develop the INYA facilities in some other sequential procedure-oriented languages such as C or Pascal.

4. INYA statements

INYA parallel programming language permits to create the parallel processes system, to define the interactions between processes and to distribute the resources dynamically. INYA program is an ordinary Fortran program with some special fragments of text defining mass computation. These fragments are enclosed in statement brackets `pardo` and `parend`. The program fragment between `pardo` and `parend` is the program of process generation. The brackets `pardo` and `parend` define the serial looping execution of this program, under the i -th execution of the program of process generation the i -th process is generated.

The INYA statements are subdivided in three groups to define data exchange, to synchronize and mutually to exclude the execution of the processes.

To transfer the data between the i -th and j -th processes the `pass` statement is used. Type of connection between SPP available when the i -th and j -th processes are running determines how the `pass` statement will be executed. The choice is made by process generation. If for instance the SPP are connected by a bus `pass` synchronizes the processes before data are transferred (rendezvous), if the SPP have common memory, asynchronous execution of the `pass` statement is possible. During the execution the PP has the information about the structure of the MCS where it is being executed.

The processes are synchronized by `wait` statement. The `sect` statement delays the execution of the $(i+1)$ -th process until the `post` statement is executed in the i -th process.

Statements which define the times of locking out the resources and their release are used to ensure an efficient sharing of resources. The processes lock out necessary resources during the execution. Resource allocation to the process by default is rolling but as for SPP the user can either choose the rolling or assign the process to the first available SPP. A similar approach is planned for controlling common memory. More resource allocation algorithms will be additionally embedded into INYA so that the user who employs them as elementary algorithms for specific resource allocation could develop his own algorithm for locking out all the resources required by the processes.

There are several auxiliary statements. They are used to distribute data, to divide the arrays in some parts, to retrieve and transform data, etc.

An example of INYA program is the program for solving the linear $M \times M$ system $AX = B$ by Jacobi method. This system is transformed to the equivalent system $X = \alpha X + \beta$, where

$$\beta_i = \frac{b_i}{a_{ii}}; \quad \alpha_{ij} = -\frac{a_{ij}}{a_{ii}}, \quad i \neq j; \\ \alpha_{ij} = 0, \quad i = j \quad (i, j = 1, 2, \dots, n).$$

At the beginning we choose $X^{(0)} = \beta$, then K approximations $X^{(k+1)} = \alpha X^{(k)} + \beta$ ($k=0,1,2,\dots,K$) are executed.

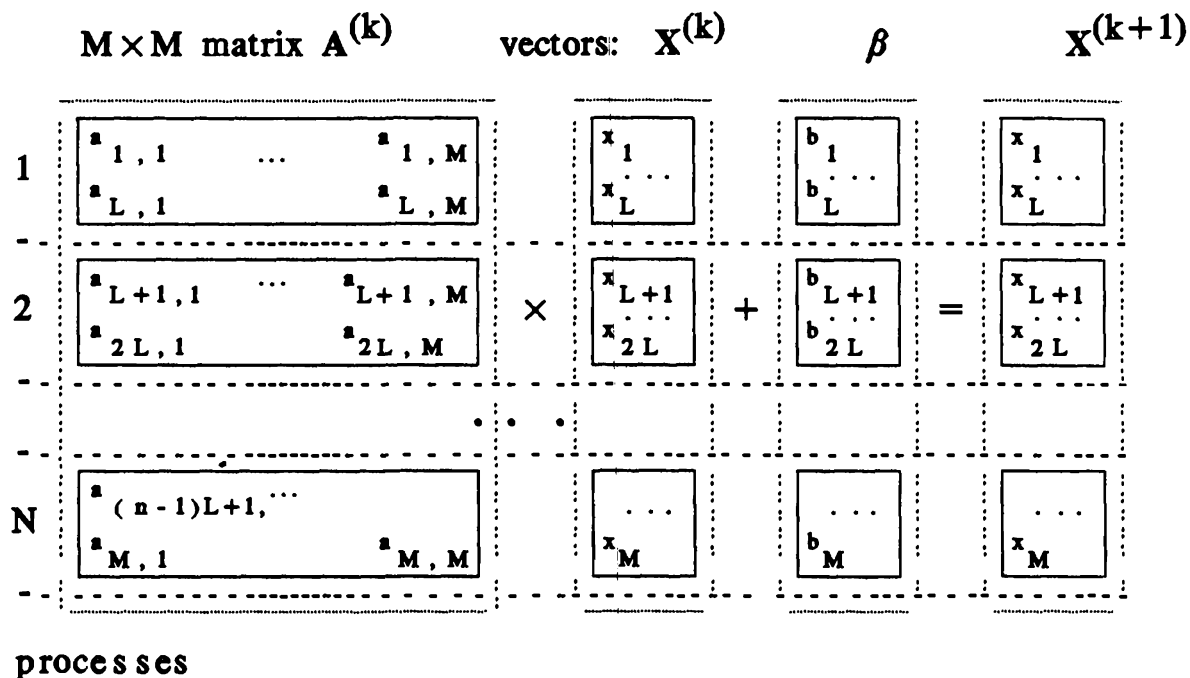


Fig. 2

The `snip` statement divides matrix A into N layers. The `iblock` function calculates the size of vectors block for each process. Then n -th layer and n -th block are transferred into the n -th process by `pass` statement. $L = M/N$ where N is the number of PEs. The blocks of the vector $X^{(k)}$ are returned to the host process as result of the k -th approximation (Fig.2).

The program is intended for the ES-2706 subsystem.

```

SUBROUTINE EQUAP (A,B,X,M,L)
C-----
C                               LINEAR EQUATION SOLVER
C
C A - M x M matrix of coefficients
C B - vector of right-hand side
C X - destination for solution vector
C M - order of matrix
C L - the number of iterations
C-----
C

```

```

        DIMENSION A(M,M),B(M),X(M),ITAB(2,20),PA(65536)
        JTR = 1
C source system transformation
        CALL FORMA (A,B,X,M)
C allocation PE main data memory
        IX = 0
        IA = IX + M + 1
C
        DO 1000 J= 1,L
            I1 = 1
            I2 = 1
            I3 = 1
C program of process generation
            PARDO I
C to determine i-th block size
            M1 = IBLOCK(M,I)
            IF (J.GT.1) GOTO 10
            NA = M1*M
            IB = IA + NA + 1
            IC = IB + M1 + 1
C data allocation to i-th process
            IF (IN$AP.GT.1) SNIP A,PA,JTR,M,M,ITAB
C
            IF (IN$AP.GT.1) PASS 0,PA,I,IA,ITAB
            IF (IN$AP.EQ.1) PASS 0,A,I,IA,NA
C
            PASS 0,B(I2),I,IB,M1
            I2 = I2 + M1
C
C 10
C
            PASS 0,X,I,IX,M
C
            CALL MMUL (IA,1,IX,1,IC,1,M1,1,M)
            CALL VADD (IC,1,IB,1,IX,1,M1)
C
C wait until calculation is finished before getting results
            CALL APWR
C get i-th process result in host
            PASS I,IX,0,X(I3),M1
            I3 = I3 + M1
C the i-th process is formed and started
            PAREND
C wait until all processes are finished
            WAIT
C
C 1000
C
            CONTINUE
C
            RETURN
            END

```

5. References

1. V.A.Anisimov, V.E.Malyshkin, N.N.Mirenkov, *Parallel Programming Language for Large-block Parallel Systems. Simultaneous Processes. Preprint N 824* (Novosibirsk, Computing Center, 1989).
2. V.A.Anisimov, V.E.Malyshkin, N.N.Mirenkov, *Parallel Programming Language for Large-block Parallel Systems. Weakly Dependent Processes. Preprint N 825* (Novosibirsk, Computing Center, 1989).
3. S.Markov and others, *Supercomputer*, vol.31, VI-3, (1989) p. 12.
4. V.E.Malyshkin, *Programmirovaniye*, N 2, (Moscow, 1991).

PARALLEL ALGORITHMS FOR GRAPH TOPOLOGY ANALYSIS AND THEIR IMPLEMENTATION FOR A VECTOR SUPERCOMPUTER

A. M. WABISHCHEVICH, N. S. KOVALENKO, M. I. OVSEETS and
L. I. SHEVCHENKO

*Institute of Mathematics of Byelorussian Academy of
Sciences, Surganova 11, Minsk, 220604, BSSR*

ABSTRACT

Vectorized algorithms for the solution of problems of graph topology analysis and the corresponding software implementations for the vector supercomputer are proposed. Their analytic time complexity estimates are given.

1. Introduction

In this report we propose parallel algorithms and the corresponding software implementations for a vector supercomputer (VSC) aimed at the solution of the following problems of graph topology analysis: connectivity and strong connectivity testing; acyclicity testing; finding and enumeration of fundamental cycle sets in undirected graphs and the contour sets in digraphs; location of hanging vertices, articulation points, bridges etc. Analytic time complexity estimates for all software implementations are given. Program modules are written in the assembly language for the VSC "Elektronika SSBIS" and they have been included in the Scientific Library of the computer.

They comprise a part of a software package for the solution of limited resource distribution problems for large-size networks. They can be used in various applications also. The following data structures are used for the considered problems: graph arc list; initial (A) and transposed (AT) adjacency matrices; row mask matrices (MA and MAT) for A and AT; digraph reachability matrix (R); boolean matrices $A[k]$ and $A[k-1]$ corresponding to k -th and $(k-1)$ -th degrees of the adjacency matrix A; initial (B) and transposed (BT) incidence matrices. Input data structures are determined by algorithms for the solution of the corresponding problems and by the convenience of the representation of output results.

2. Vectorized algorithms

Let us give below vectorized algorithms for the solution of the above-mentioned problems.

A1. The Construction of the Adjacency Matrices and Row Mask Matrices

Input: Graph arc list.

Output: Matrices A, AT, MA, MAT.

The row mask matrix is constructed according to the rule:

1. The vertex set of the digraph ordered by the vertex numbers in the ascending order is divided into k groups by t elements, where $k = \lceil n/t \rceil$, $k \geq 1$, $t = \text{const}$.
2. An element of the row mask $n \times k$ -matrix MA (MAT) from the i -th row and j -th column is equal to 1 if at least one arc starts (terminates) in vertex $v(i)$ connecting it with a vertex from the j -th group.

A2. Source and Sink Vertices Location

Input: Row mask matrices MA and MAT.

Output: SNW and SKW - the sets of source and sink vertices of a graph, respectively.

The algorithm is based on vectorized operations of the location of zero rows in MA and MAT.

A3. Finding the Set of Vertex Unreachable from a Given Start Vertex of a Graph

Input: The adjacency matrix A of the digraph and index NV of the start vertex.

Output: MNV - the set of vertices unreachable from the start vertex.

1. Initialization. $MNV := 0$, $MP(i) := 1$ for all $i = 1, 2, \dots, n$.
2. The "closest-vertex"-type graph traversal starting from vertex NV. Here $MP(j) := 0$ for each analyzed vertex.
3. For all $i = 1, 2, \dots, n$ if $MP(i) = 1$, then $MNV := MNV \cup \{i\}$.
4. Exit.

A4. Finding Strongly Connected Components

Input: Initial and transposed adjacency matrices A, AT.

Output: A number k of strongly connected components and a family of sets $SD(i)$ defining lists of arcs for each i -th strongly connected components.

1. A unit vector NP of length n (n is the number of digraph vertices) is formed.
2. An array N of the indices of digraph vertices is formed from the matrix A by the depth-first search. N-number i corresponds to vertex $N(i)$.
3. $i := 1$. $k := 0$.
4. $j := N(i)$. $MP(j) := 0$.
5. If $AT(j, *) \& MP(*) = 0$, then go to Step 8.
6. An N-region for vertex j is formed ($NR(j)$). $k := k + 1$. All arcs from the given strongly connected component are found by the "closest-vertex" method and put into set $SD(k)$.
7. For all $v(1) \in NR(j)$ $MP(v(1)) := 0$.
8. $i := i + 1$. If $i \leq n$, then loop to Step 4.
9. Exit.

A5. Digraph Acyclicity Testing

Input: Adjacency matrix A and reachability matrix R of the digraph.

Output: S is an indicator of contour presence /absence.

The algorithm is based on boolean multiplication of vectors $r(k)$ and $a(k)$ representing the k -th row of matrix R and k -th column of matrix A , respectively.

A6. Fixed Length Contour Search

Input: Matrices A , $A[k]$, $A[k-1]$ that are the digraph adjacency matrix and its k -th and $(k-1)$ -th degrees respectively.

Output: Contour matrix $D(c,k)$, where c is the number of contours, k is the contour length.

We propose the following two vectorized algorithms. The first (A6.1) one is based on the depth-first search method through all vertices of the digraph with checks of the conditions which allow to minimize the complete enumeration. The second (A6.2) is based on the Hadamard multiplication of $(0,1)$ -matrices and on the property of the k -th degree of the digraph adjacency matrix which contains an information about the existence of a path of length k between the digraph vertices. This in turn allows to limit the search of contours to a partial subgraph of the initial graph which considerably decreases the search time.

A7. Contour Enumeration in a Digraph

Input: Incidence matrix A and reachability matrix R of a digraph.

Output: Contour matrix $D(c,n)$, where c is the number of contours, n is the number of digraph vertices.

The algorithm is based on the construction of directed trees with roots in vertices $v(1), v(2), \dots, v(n)$ and followed by choosing paths from the root $v(i)$ to the end vertex $v(j)$ with coinciding indices, in each of them.

A8. Finding Fundamental Cycle Set in a Graph

Input: BT is the transposed incidence $(m \times m)$ -matrix of a graph, E is the unit $(m \times m)$ -matrix.

Output: C is a $((m-n+1) \times n)$ -matrix of a fundamental cycle set.

1. Initialization of the nextcycle index ($L:=0$) and the current column index ($j:=0$) for BT .
2. $j:=j+1$. If $j>n$ then go to Step 10.
3. Find the index $I1$ of the first "One" in the j -th column of BT .
4. In the $I1$ -th row of BT find the index k of the second "One". $I2:=I1$.
5. $I2:=I2+1$. If $I2>m$ then $BT(I1,k):=0$ and loop to Step 2.
6. If $BT(I2,j)=0$ then loop to Step 5.

7. $E(I2,*) := E(I2,*) \# E(I1,*)$. Here $E(I1,*)$ is the $I1$ -th row of E , $\#$ is the vectorised exploded/wise addition modulo 2.
8. If $BT(I2,k)=1$ then go to Step 9, else $BT(I2,k):=1$ and loop to Step 5.
9. $L:=L+1$. $C(L,*) := E(I2,*)$. Here the $I2$ -th row of E gives the next cycle. $BT(I2,k):=0$. Loop to Step 5.
10. Exit.

A9. Finding Bridges and Articulation Vertices of a Contourless Digraph

Input: Adjacency matrix A of digraph.

Output: List SM of the arcs which are the bridges.

This algorithm is based on the following properties of an articulation vertex v^* of a contourless digraph G :

1. v^* can't be a source or sink vertex of G .
2. For no arc $(v(i), v(j))$ the condition $N(v(i)) < N(v^*) < N(v(j))$ holds.

If $v(i)$ and $v(j)$ are articulation points of G and $N(v(i)) = N(v(j)) - 1$ then there exist an arc $(v(i), v(j))$ which is a bridge of G .

3. Conclusions

The algorithms for the solution of problems of graph topology analysis considered here belong to the class of logic-combinatorial ones. The latter algorithms are inherently hard to vectorize and parallelize. This is especially true for VSC, for which the efficiency strongly depends on the degree of vectorization and pipelining of computations. Therefore, the implementations of algorithms of this class use various speed-up techniques: vector-pipeline data processing with multilevel register memory, minimization of the total delay time of command issuing, bitwise representation of graph information and processing it in vector registers, loop unrolling with the optimum multiplicity value, macropipeline forming with several functional units, etc.

In order to minimize the complexity of the designed programs their structuring was carried out with the localization of time-consuming basic operations. For all program modules analytic estimates of run times in cycles of VSC are obtained as functions of problem sizes. These estimates are summed up in table. Here n is a number of graph vertices, m is a number of graph edges or arcs, $N = \max\{5, \lfloor n/64 \rfloor\}$, n_1 and n_2 are a numbers of source and sink vertices respectively, n_3 is a number of vertex unreachable from a given start vertex, k is a number of strongly connected components, $n(k)$ and $m(k)$ are a numbers of vertices and arcs of strongly connected components, respectively, L is the length of a contour, p is a number

of "Ones" in the main diagonal of matrix $A[L]$, c is a number of contours, m_1 is a number of partial subgraph arcs, t is a number of articulation vertices, M is a number of bridges.

Algorithms	Estimates of run times
A1 A2	$110\max\{5, m\} + 44(\lfloor m/64 \rfloor - 1)$ $2n + 66\lfloor n/64 \rfloor + 14(n_1 + n_2)$
A3	$N + 30n^3$
A4	$19n + 16n(k) + 30m(k) + 50k, n \leq 64$ $(35 + 2N)n + (30 + 3N)n(k) + (60 + 4N)k + 30m(k),$ $64 < n \leq 1088$
A5	$2,5n^2$
A6.1 A6.2	$(1, 4n(n+m) + 2, 8n + 30)(p-1) + (2L + 32)c + n$ $n^2(n+m_1) + 3, 3n^2 + 11n + (2L + 5)c$
A7	$1, 6n^2(m+n+c)$
A8	$O(mn^2)$
A9	$25n + 15t + 11M, n \leq 64$ $(52 + 2N)n + 34\lfloor n/64 \rfloor + 50t + 20M, 64 < n \leq 1088$

A PROGRAMMING METHODOLOGY FOR EFFECTIVE DATA EXCHANGES IN MACROCONVEYER PROGRAMS

DOROSHENKO A. E.

*V.M. Glushkov Cybernetics Institute of the Ukrainian
Academy of Sciences, Glushkov prosp., 40, 252207,
Kiev 207, U.S.S.R.*

ABSTRACT

Macroconveyer programs - coarse-grained parallel programs, that have distributed primary and shared secondary memory - are considered. Programming techniques to improve macroconveyer programs efficiency through asynchronous communications and systematic second memory accesses elimination are proposed. They include some semantic models for enhancing intramodule communications asynchrony, regular expressions called forcing expressions (FE) for enhancing intermodule communications asynchrony and FE-based method for control of external data exchanges buffering. The methods are illustrated by an example of linear algebra program.

1. Introduction

The principle of macroconveyer in parallel computations¹ was suggested by V.M. Glushkov as an architectural principle of coarse-grained parallelization, that integrates MIMD multiprocessor architecture with distributed memory and universal communication net. The essence of macroconveyer multiprocessing is in such a workload distribution among processors that maximizes for every processor computational periods of time and reduces quantity of data exchanges needed for their communication. Using this principle provides high speedup and efficiency of parallel computations in wide area scientific applications. Macroconveyer multiprocessor system ES 1766 developed in V.M. Glushkov Cybernetics Institute of the Ukrainian Academy of Sciences includes up to 256 processors, IBM-compatible ES 1066 computer as a host machine and universal communication net providing pairwise parallel communications. MAYAK language² and basic software system are developed for problem solving programming.

Advancing communication mechanisms is an important opportunity to

improve parallel programs efficiency. From this point of view there are two factors essential for macroconveyer programs efficiency. First is the principle of macroconveyer itself. Coarse-grained parallelism may provide a good deal of overlapping computations, but huge lost of speedup and efficiency may be possible as well in cases of parallel processes that have unbalanced volumes of computations and/or uncoordinated points of synchronization. So macroconveyer principle needs some techniques of asynchronous communications for smoothing possible differences in parallel processes computational rates. Second factor is a two-level memory of macroconveyer programs. Primary memory is fast but distributed over parallel components and localised in them. Secondary one is shared but slow. Great number of exchanges with secondary memory decreases program efficiency and may degrade system performance when the number of parallel components is too large. Therefore reducing such exchanges is another actual problem with data communications in macroconveyer programs.

2. Macroconveyer programs

A model of macroconveyer programs considered here is a tuple $p=(P,K,t,E)$ where $P=\{P_i\}$ - finite set of sequential component programs (modules), K - a set of components - logical names of parallel processes, $t:K \rightarrow P$ - (partial) initialization map providing initial program configuration, E - set of outer array names. Modules are ordinary sequential programs except data exchanges statements. These last are of two kinds: direct pairwise exchanges with statements $\text{TRANSFER } x \text{ TO } k_1 \text{ and } \text{ACCEPT } y \text{ FROM } k_2$ (for short, $x \rightarrow k_1$ and $y \leftarrow k_2$), that are executed at components k_2 and k_1 respectively, and external exchanges through shared memory with statements $\text{READ } A \text{ IN } x \text{ and } \text{WRITE } y \text{ TO } A$ (for short, $x \leftarrow A$ and $y \rightarrow A$), where x, y - inner, A - outer array name. Outer arrays being located in shared memory of the program are in effect multidimensional files. They are organised as two-level structures: multidimensional array of data blocks at the first level that are multidimensional arrays of data elements - at the second. So if, for example, we need to modify (i,j,k) -element of (m,n) -block of an outer array A with a function f we must execute in some

component following sequence of statements: $U \leftarrow A(m,n);$
 $U(i,j,k) := f(U(i,j,k)); U \rightarrow A(m,n);$ where U is a variable (array name) in local memory of this component. For external exchanges synchronization basic semaphore-like facilities are provided.

Basic parallel facilities of MAYAK language include explicit parallel components definition and parallel (remote) procedure call. Semantics of macroconveyor program execution establishes simultaneous execution of components whose initialization map is definite. Program configuration may vary during execution due to parallel procedure calls, that are synchronised with PAR DO... PAR END statement brackets. Input and output data are supposed to be in outer memory.

The model considered differs from well-known CSP - based languages³ model at least in three points. The first, macroconveyor programs allow dynamic parallel processes structure that may vary during program execution. The second, macroconveyor programs use both distributed and shared memory. Moreover shared memory is the secondary one and therefore is slower. The third, unlike CSP, where possible waiting for synchronization in parallel communicating processes can be partially compensated by quasi-parallelism inside of processes, modules of macroconveyor programs considered are inherently sequential and do not support internal parallelism. This simplifies programming problems for applied programmers and gives good possibilities in computer aided adaptation of software developed earlier and in other sequential languages but raises needs for thorough design of modules synchronization and providing efficient communications. These last are of our special interest.

Efficiency of parallel computations is evaluated usually as fraction $\alpha = T_1 / p \cdot T_p$, where T_p is a parallel program execution time at p processors of equal performances, T_1 is a time of execution of the same volume of computations at single processor that hasn't less of primary memory than the p processors. Fraction $\beta = T_1 / T_p$ is usually called speedup of computation due to parallelism and characterizes some overhead needed for parallel computations organization. Therefore when the parallel problem decomposition is already defined and program

component are fixed there are two ways to increase macroconveyer program efficiency - first, to reduce parallel processes synchronization waiting by means of asynchronous communications and second - to eliminate, when possible, component exchanges through slow external memory replacing them by fast direct data exchanges.

3. Asynchronous communications

Lost of macroconveyer program efficiency because of synchronization waiting may be too large as they are inherently coarse-grained. For this reason MAYAK language suggests semantics of "suspended" data exchange given one of partner processes is not ready. Unlike synchronous data exchanges as in CSP and OCCAM, it provides isolating initialization stage from completion one for every exchange and, while waiting mutual readiness for an exchange, asynchronous execution of statements that are data independent from this exchange. These last constitutes some program segment, environs of given exchange, that defines some synchronization freedom of communicating parallel components by means of evaluating readiness for an exchange not at point but inside of some time interval. A depth of such an asynchronous continuation depends on a underlying semantic model of communications. Some of theoretical communications models generalizing well-known concept of rendezvous have been constructed⁴. One of them is implemented in MAYAK language.

As an example consider a fragment of a component program for block mutrix multiplication given matrices are placed in outer arrays.

```

Z:= 0;
  for k:=1 to n do
    U:<- A(i,k);
    V:<- B(k,j);
    Z:=Z + U * V;
  end for;
Z: ->C(i,j);                                     (1)

```

Sum and product mean matrix operations there. One can see that read statements have empty synchronization environs because of data dependences on the assignment statement in loop body. For this reason

loop iterations must be computed sequentially. But this dependence may be broken providing auxiliary memory for data exchange with outer arrays.

```

U1: <- A(i,1);
V1: <- B(1,j);
Z:=0; b:=1;
  for k:=2 to n do
    if b=1 then
      U2: <- A(i,k);
      V2: <- B(k,j);
      b:=2;
      Z:=Z + U1 * V1;
    else U1: <- A(i,k);
      V1: <- B(k,j);
      b:=1;
      Z:=Z + U2 * V2;
    end if;
  end for;
  if b=1 then Z:=Z + U1 * V1;
  else Z:=Z + U2 * V2;
  end if;
Z: -> C(i,j);                                     (2)

```

In transformed program fragment (2) any two neighbour iterations are data independent and therefore may be computed concurrently exceeding in data exchange and arithmetic computation overlap. Indeed such a look-ahead strategy for data exchanges may be performed for arbitrary reasonable number of contiguous iterations.

4. Forcing expressions

The program transformation above is an example of introducing intracomponent data communications asynchrony but for extracomponent case it is possible as well. The matter is that in some cases of application programs there is a need to provide thinner than basic semaphor-like synchronization facilities and to use additional opportunities of overlapping component computations. Special case

regular expressions, called forcing expressions, were developed by the author to facilitate external exchanges synchronisation for race free macroconveyor programs, where race free feature means that any pair of data dependent statements of external exchange, one of them is to be WRITE statement, are always executed in the same order.

For generality reasons the notion of the forcing expression we give for the case of regular, in the sense of Glushkov's algebra of algorithms⁵, component program. Let V be a variable set and D - a data domain for these variables. A set of partial mappings $B=\{b:V \rightarrow D\}$ is called a set of memory states. Assume being known a set of basic operators $Op=\{y:B \rightarrow B\}$ with unit operator id and a set of basic conditions $Co=\{u:B \rightarrow \{0,1\}\}$ that also includes boolean constants 0 and 1. An algebra of partial transformations Y , generated on Op by means of three operations:

1. $(PQ)(b)=(P(Q(b)))$ - concatenation of operations P and Q ;
2. $\{P \vee Q\}(b)=$ if $u(b)$ then P else Q - branch operation;
3. $\{P\}(b)=$ if $u(b)$ then $\{P\}(P(b))$ else b - iteration, is called algebra of operators. An algebra U , generated on Co by means of boolean operations and operation of multiplication by condition:

4. $Pu(b)=u(P(b))$ - " u after P ",

is called algebra of conditions. The two-set algebra $A(Y,U)$ considered as set of operators and conditions closed under operations 1-4 is called algebra of algorithms. Each element of algebra $A(Y,U)$ can be represented as regular expression of elements basic operators and basic conditions sets and operations of algebra of algorithms. Regular expressions of algebra of operators Y are called regular programs.

Let now (P,K,t,E) be a macroconveyor program with regular modules where a configuration mapping is total. That is for every component k there is exactly one module that can be loaded in k . Let Z be a set of variables, called control ones, other than inner variables of modules, $C=\{c:Z \rightarrow D\}$ - a set of states of this kind of memory and S - an algebra of conditions over control variables. For every $k \in K$ operation symbols R_k - reading symbol, and W_k - writing symbol, are introduced. Binary operation (P,Q) , commutative and associative, where P and Q are regular expressions constructed with the reading symbols only, is

called parallel reading operation. Let $K(A)$ be the set of components of a race free program that are communicated externally through outer name A . Then the forcing expression (FE) is a regular expression $f(A)$, constructed with reading and writing symbols R_k and W_k , $k \in K(A)$, unit operator and with memory conditions from S by means of regular operations 1-4 and operation of parallel reading. The main idea of FE introduction, unlike path expressions⁶, is to enforce the order of component accesses to outer memory, that may be known a priori in the case of race free program, and to eliminate nondeterminacy for increasing program performance. Execution of race free program whose external exchanges monitored by FE consists in joint interpretation of component programs and forcing expressions assuming read (write) statement execution initialization in component k must correspond interpretation of R_k (W_k) as current symbol of FE otherwise synchronization delay in the component invades.

Because of more refined level of synchronization forcing expressions can provide more effectiveness for parallel programs compared with semaphore-like facilities. Besides, as may be seen below, FE may be use for data exchange speedup in multi-level memory system.

5. External exchange speedup

Data exchange speedup is another method of efficiency enhancement. One approach there is program analysis and transformations to reduce number of accesses to slow shared memory⁷. Another one is buffering this kind of exchanges in primary memory of multiprocessor system. But instead of general-purpose buffering algorithms, being used in operating systems, some dedicated methods, that take into account some "knowledge" of special cases, may be possible and more effective. One of these cases - race free programs mentioned above. For this class of programs a FE-based method of controlled buffering was developed by the author. A special case of this method - exchange pipelining - may be illustrated by the same program example for matrix multiplication. Given $K = \{(i, j)\}$ and (i, j) -element of result matrix is computed by component named (i, j) , forcing expressions for outer arrays are:

$$f(A_{ij}) = \{R_{ik}\}_k, f(B_{ij}) = \{R_{kj}\}_k, f(C_{ij}) = W_{ij}, \quad u = (1 \leq k \leq n), 1 \leq i, j \leq n. \quad (3)$$

It's clear from these expressions that C array exchanges buffering is unnecessary because of once being written an element of C matrix then never be read. But A and B arrays exchanges are multiple and therefore need to be buffered by means of single reading an element to one component and passing it then to another ones. Given the component coordinates are saved in pair of variables (q1,q2) in every component, a pipelined version of program is following:

```

Z:= 0;
  for k:=1 to n do
    if q2 = 1 then U<- A(i,k,);
      U-> K(q1,q2+1);
    else U<- K(q1,q2-1);
      if q2 ≠ n then U -> K(q1,q2+1);
    end if;
  end if;
  if q1 = 1 then V<- B(k,j);
    V -> K(q1+1,q2);
  else V <- K(q1-1,q2);
    if q1 ≠ n then V -> K(q1+1,q2);
  end if;
  end if;
  Z:=Z + U * V;
end for;
Z: ->C(i,j);

```

(4)

As a result, almost half a number of external exchanges, namely $2n^2(n-1)$, will be replaced by fast exchanges between components. It's important that because of the buffer migrates there no auxiliary memory is needed but only the one provided by compiler.

An important consequence of enhancing efficiency owing to external exchanges speedup is widening the range of processors, where speedup depends linear on number of processors involved in problem solving. Some analysis performed for a class of cyclic homogeneous component programs argues exsistency some critical numbers of processors that being exceeded cause a nonlinear slowing-down of multiprocessor speedup. In these cases some steps of access distribution to buffer

are to be performed to preserve a linear speedup.

6. References

1. V.S.Mikhalevich, Ju.V.Kapitonova, A.A.Letichovsky, *INFORMATION PROCESSING 86* (IFIP, Amsterdam, 1986) 975-980.
2. S.S.Gorokhovskiy, Ju.V.Kapitonova, A.A.Letichovsky, I.N.Molchanov, S.B.Pogrebinsky, *Cybernetics*, 20 (1984) 1-10.
3. C.A.R. Hoare, *Communications of the ACM* 20 (1978) 666-677.
4. A.A.Letichovsky, A.B.Godlevsky, A.E.Doroshenko, S.L.Krivoy, *Programming*, no. 5 (1983) 3-12.
5. A.A.Letichovsky, *Information Processing 83* (IFIP, Amsterdam, 1983) 895-864.
6. R.H. Campbell, N.A. Haberman, *Lect. Notes Comput. Sci.* 16 (1974) 89-102.
7. W. Abu-Sufah, D. Kuck, D. Lawrie, *IEEE Trans. Comput.* C-30 (1981) 341-358.

LINEARIZED MASS COMPUTATION

V.E.Malyshkin

Computing Center, Siberian Division, the USSR Academy of Sciences,
630090, Novosibirsk, USSR

ABSTRACT

An approach to organization of parallel computations on large-block high-performance multicomputer systems (MCS) is presented which is named the linearized mass computations. This approach includes the design of hierarchical linear MCS for large-scale computation, representation of the algorithm as a system of linearly ordered processes with linear interactions and algorithms of MCS resource allocation. This approach has been used in Novosibirsk Computing Center for the development of the MCS Siberia. On the basis of this MCS a user-friendly parallel programming system Inya has been developed. The Inya system supports the development of applied parallel programs which tune to available MCS resources in the course of their execution.

The module approach to the computer system design made it possible to develop the high-performance large-block multicomputer systems of different structures assembled out of various large computer blocks such as various microprocessors, special-purpose processors, communication channels, memory modules etc¹⁻³. There are additional units: disks, bulk storage, buses, input/output processors, which can also be used to create the MCS. Microprocessors, special-purpose attached processors are used as processor element (PE) of the MCS. Each of them has the control units, registers and memory of their own, so that the data and programs transferred to the PE can be processed independently of the host computer(HC). The large-block MCS has the following advantages:

a). It is possible to realize the fine and coarse grain parallelism on the united resources of the MCS. Fine grain computations are performed on the processor element, coarse grain computations are performed on the resources of the whole MCS. It is possible to achieve a high performance by solving the complex problem.

b). The MCS can be readily created as its PEs are included in the

MCS with their software. Thus it is necessary to develop the integrating software only.

c). The applied software to be run on MCS can be modified simply enough, as it is evolved, but not rebuilt.

d). Large-block MCS are especially necessary when the volume of computation must be drastically increased on the basis of hardware and software in current use.

However it is not easy to make use of the above advantages. The basic difficulty is the vast diversity of possible MCS's and algorithm's structures which do not always correspond to each other. It slumps the MCS performance. An important problem is to develop the software for such systems. Effective application programs must be developed, their portability to different MCS must be ensured, the accumulation of applied programs is necessary. Not only the software of different MCS items must be integrated within a united programming system, but also the parallel programs dynamically tuning to the available resources of a particular installation must be designed.

Our task in designing the MCS, consisted in developing: 1) the regular method of designing such systems, 2) the parallel programming system (PPS) for the creation of applied parallel programs (PP) which dynamically share the MCS resources, tuning to the available resources during the execution.

Generally the PPS for such MCS is to enable the user to define the mass computation as a system of interacting processes, each process being the program and the data it processes and PP must be assembled out of the ready-made modules⁴⁻⁵. Here the interaction is the data exchange between the processes. Each executing process dynamically locks out necessary resources (PE, memory). PPS must provide effective execution without deadlocks.

Surely, no acceptable organization of parallel computations is possible without restrictions imposed on either the MCS or the algorithm structure or both. In order to execute the mass computations which occur very often in practice, for example on solution of problems of signal and image processing, a simple approach

can be used which takes into account the regularity of mass computation. It is formulated here as a linearization method which determines:

- the class of allowed MCS - linear hierarchical MCS,
- acceptable structures of algorithms - linear algorithms,
- fixed algorithms of MCS resource allocation.

This approach has been employed in Novosibirsk Computing Center for developing the large-block linear hierarchical MCS Siberia and the language and PPS Inya for this MCS. Inya is an extension of Fortran with additional facilities for specification of the system of interacting processes. MCS Siberia is based on the high-end model of ES computers, subsystem of 8 attached processors ES-2706 (like AP-190L), processors ES-2720 and PS-2000/2100. MCS has no general fixed structure, depending on the current necessity various PEs and communication channels are being added in or removed from the MCS, effective tuning of PP being provided by Inya.

1. Linear hierarchical MCSs

First two steps define the rules of the MCS design. At the first step the basic elementary "building blocks" of the system are selected. The first "building block" is the host computer, say IBM PC or high-end IBM-370 with a complete set of peripheral devices and well-developed external communications. The second component is represented by PEs (e.g. AP-190L, ES-2709, ES-2704⁶, Intel and Bit microprocessors etc). Each PE is a high-performance processor with the main memory and control units of its own. PE operation is controlled by HC which initiates PE, transfers programs and data to the PE, synchronizes the computation. At the same time PE as a slave computer may even fail to have its own i/o devices. In addition to the processor components communication channels providing data exchange between them are also necessary.

At the second step the options of connections of "building blocks" are defined. These connections must be chosen in such a way that MCS can have some regular structure, in which the algorithm of MCS resource allocation (M-algorithm) could be fixed. There are two allowed types of connections. Radial connections provide the

host-slave interaction between HC and PEs. Identical PEs constitute a subsystem, there can be several PEs subsystems, e.g. subsystems of AP-190L and ES-2709.

The other type of connection is interprocessor interface between the PEs inside a subsystem. It provides a high data rate of exchange between PEs bypassing the HC and effective interaction of the lower-level processors. The interface always provides the interaction between PE_i and PE_{i-1} and PE_{i+1} . This interface is named the *linear interface*. The channels, bus, network, common bulk storage can be used. For example, the PEs can constitute the line or the ring. The above connections present only the minimum connections between the components of MCS, other connections may be added.

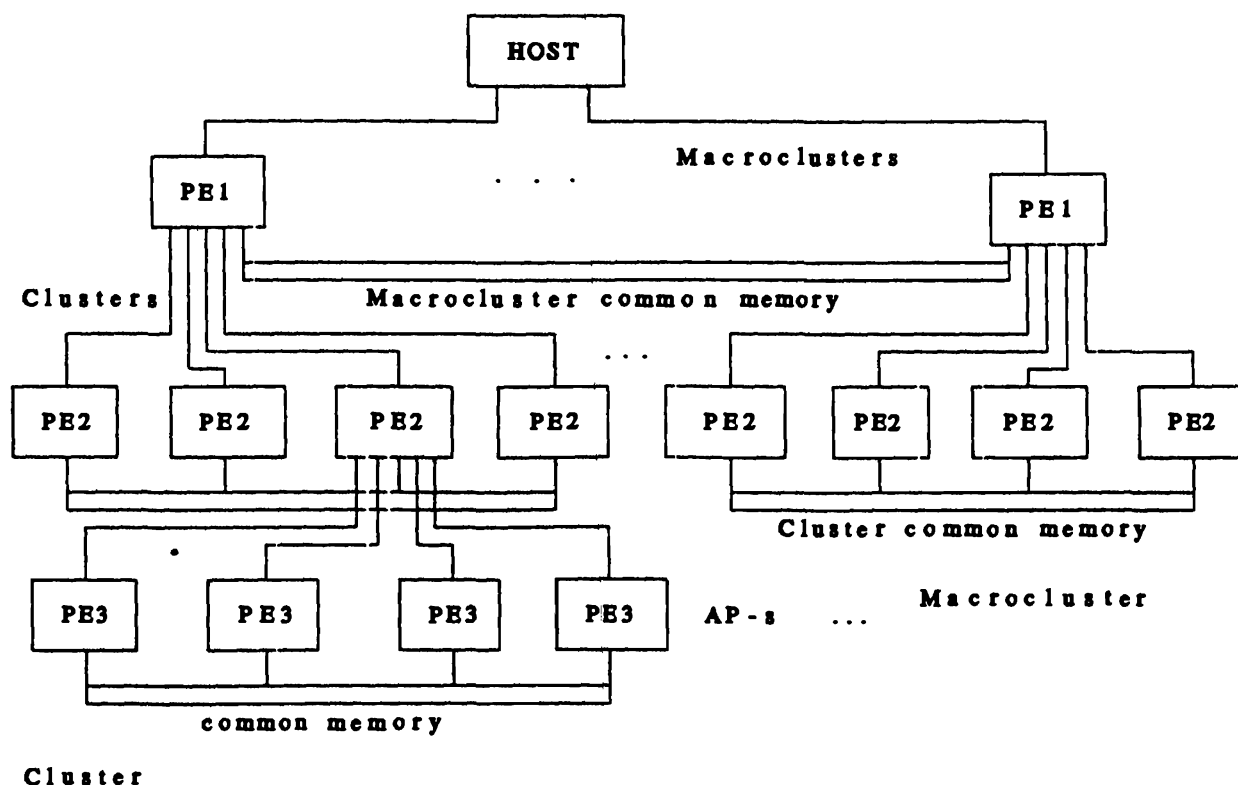


Fig.1

Every PE in its turn can have a complicated structure. In this case there also must be a host and slave processors. Thus the allowed MCS must have a hierarchical structure with linearly connected PEs, it is named a *linear hierarchical MCS*. Fig.1 shows an MCS with a macroclusters subsystem⁷.

Since intercomputer connections are linear and hierarchical, the parallel programming language must enable one to define the

computation of linear hierarchical structure providing their effective mapping in the MCS resources. It implies that in any program (process) being executed there must be the possibility of generating the slave processes and sending them to PEs for execution .

2. Linear algorithms.

Now taking into account the regularity of the PEs interprocessor interface it is necessary to define the restrictions imposing on the algorithm structure. These restrictions must enable to fix the M-algorithm in PPS. More exactly, it must be the parameterized scheme of PEs allocation. The assignment of processes on PEs must be done dynamically in PP depending on structure and number of locked out MCS resources. Programming the problem in this PPS the programmer will have to find the needed algorithm representation. Certainly it must be very easy for him.

The restrictions imposed on the structure of the algorithm - the conditions of the algorithm linearity - are as follows:

1. There exists some linear ordering of the operations of the algorithm. An operation here denotes a computation step, in PP it is an execution of a certain procedure or of a subroutine with its specific input data.

2. In chosen linear ordering all interactions must be linear , e.g. if a_i and a_j operations interact, then $j=i+b_k$, $k=1,2,\dots,n$, b_k,n are the integer constants. Therefore an integer constant m exists and any operation a_i can interact only with a_j operation, where $a_j \in \{a_{i-m}, \dots, a_{i+m}\}$. It is a linear interaction and algorithm in this representation is named the linear algorithm.

Hence there exists such an integer m that the operation a_i may interact only with the operation a_j , $a_j \in \{a_{i-m}, \dots, a_{i+m}\}$. Therefore if an algorithm is being mapped in the MCS resources, the operations of neighborhood $\{a_{i-m}, \dots, a_{i+m}\}$ will be assigned on the processors, after that all interactions of a_i operation can be performed.

The condition for the algorithm linearity may be sometimes replaced

by the condition of the existence of a finite neighborhood $\{a_{i-m}, \dots, a_{i+m}\}$ for each operation a_i where the interactions a_i with a_j can take place only, m is an integer constant. In this case only subsystem of PEs with complete interprocessor interface such as a bus, a common memory etc is to be used.

Though no restrictions are imposed on the structure of interprocessor communications there is the following assumption. When every operation of the neighborhood of a_i is assigned on the processor, all interactions of a_i operation can be performed, if the necessary links are available. The a_i will be finished and release the resources, which can be assigned to the other operation.

Evidently, the mass algorithm representation does not always satisfy the linearity conditions 1-2. But mass algorithms usually of a regular structure can be linearized. Consider the following example. Let A be a matrix whose elements must be sequentially recalculated (filtrations of a kind):

$$\begin{aligned}
 A_{ij}^0 &= A_{ij}, \\
 A_{ij}^1 &= f_1(E_1(A_{ij}^0), b_1), \\
 A_{ij}^2 &= f_2(E_2(A_{ij}^1), b_2), \\
 &\vdots \\
 A_{ij}^k &= f_k(E_k(A_{ij}^{k-1}), b_k)
 \end{aligned} \tag{1}$$

$$\forall i \in \{1, 2, \dots, s\}, \forall j \in \{1, 2, \dots, t\},$$

$k > 0$ is an integer constant.

Here E_k is some neighborhood of the element A_{ij} . For instance

$$\begin{aligned}
 &\begin{matrix} 0 & 0.5 & 0 \end{matrix} \\
 b_1 = &\begin{matrix} 0.5 & 1 & 0.5 \end{matrix}, \quad E_1 = (A_{i-1,j}, A_{i+1,j}, A_{i,j}, A_{i,j-1}, A_{i,j+1}), \\
 &\begin{matrix} 0 & 0.5 & 0 \end{matrix}
 \end{aligned}$$

(1) is a system of recurrent equations. They are often used to define the mass regular computations. The functions $f_1, \dots, f_k$ are executed in PP by procedures. Let us order the algorithm operations in such a way that elements of the matrix A are computed along the line (the superscript of the functions denotes the operation number in this ordering):

$$A_{22}^1 = f_1^1(E_1(A_{22}^0), b_1), \quad A_{23}^1 = f_1^2(E_1(A_{23}^0), b_1), \quad A_{24}^1 = f_1^3(E_1(A_{24}^0), b_1),$$

$$\dots, \quad A_{33}^2 = f_2^{(s-2)(t-2)}(E_2(A_{33}^1), b_2), \dots,$$

Clearly, m depends on the A matrix size. Therefore in this ordering the algorithm is not linear. However the algorithm can be linearized, can be transformed into another functional equivalent algorithm as many other mass algorithms with the regular structure of interactions.

A matrix is divided in the overlapping layers (several adjoining columns) $A_1, A_2, \dots, A_n$ (Fig.2). All computations

$$A_{ij}^p = f_p(E_p(A_{ij}^{p-1})), \quad p=1, 2, \dots, k, \quad \text{executed in each layer } A_r, \quad r=1, \dots, n,$$

constitute the new operation a_r . The operations $a_1, a_2, \dots, a_n$

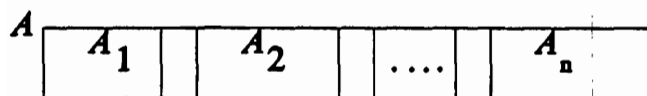


Fig.2

are linearly ordered, there are no interactions. If k is large then the number of overlapping columns is large too. In order to decrease their number after the computation of A_{ij}^p but before the computation of A_{ij}^{p+1} over all elements of A_r layer, it is necessary to restore the value of overlapping column elements through the exchange between a_{r-1} , a_r , a_{r+1} operations. In this case every overlapping will have only two columns, but the linear interactions between operations will arise. This last algorithm is linear for all such matrices and PEs,

where one layer A_r can be located in data memory of PE. The value of m is little and it is constant for a wide range of matrices.

3. Computation mapping in the MCS resources

Now the algorithm of MCS resource allocation can be fixed. This fixed M-algorithm is built into the PPS and it must be known to the user.

Consider the following example. Let the file f consist of n blocks. Each block must be read into the main memory of HC (operation in), transferred to PE (operation put) where it must be processed without interactions (operation run). The result must be transferred to the HC (operation get) and then to the external memory to the file fl (operation out), fig.3. The algorithm is linear. The set of operations is ordered as follows: $in^1, put^1, run^1, get^1, out^1, in^2, put^2, \dots$ the sequence of the i -th executions of the operations $in^i \rightarrow put^i \rightarrow run^i \rightarrow get^i \rightarrow out^i$ constitutes the process p_i . Each process p_i consists of a finite set of successive operations of

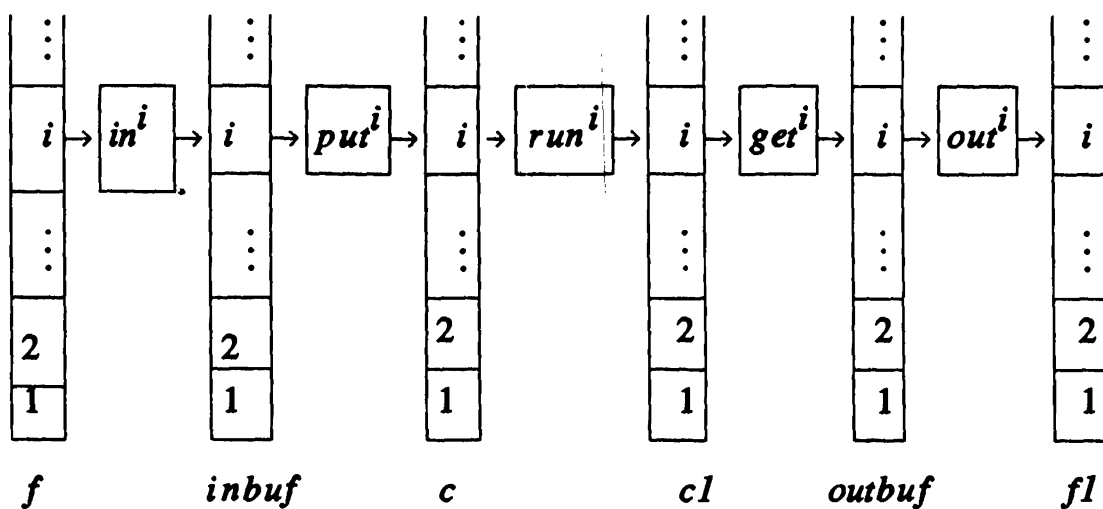


Fig.3

linearly ordered algorithm. The set of processes $\{p_1, p_2, \dots\}$ is, by definition, the same algorithm which uses larger operations, i.e. processes. Information on the structure of the processes provides a more effective resources allocation.

PEs are dynamically allocated resources as well as the HC main memory for locating the input (*inbuf*) and output (*outbuf*) buffers. Thus if a system of processes is executed using two available processors PE1 and PE2, three input buffers b_1, b_2, b_3 , and two output buffers a_1, a_2 , the resources will be dynamically allocated to the processes as follows $(p_1, b_1, PE1, a_1)$, $(p_2, b_2, PE2, a_2)$, $(p_3, b_3, PE1, a_1)$, $(p_4, b_1, PE2, a_2), \dots$. This M-algorithm is named the *rolling* of the resources. Here the buffer (e.g. b_1) denotes the slot of the main memory where input data of the current process can be located.

According to organization of computation, processes will be performed strictly in ascending order of numbers assigned to them in linear algorithm. A process with the lower number always has the priority in locking out the next needed resource. It will not allow a process with a higher number to get ahead. Certainly, as the MCS has a regular structure of interprocessor interface the rolling provides the effective execution of PP on different installations with different number of PEs.

The user of the PPS, which is based on the linearization of mass computation, must be informed about the rolling and explicitly use this information when developing his program. Allowing for the rolling he can provide PPS with additional information on the structure of the processes. In the last example it will be information about the time of locking out or release of resources (fig. 4). This makes sharing of resources possible. Example of such a timing chart is given in Fig.5, where dots designate the time of waiting for the resource release.

Fig.6 shows the possible organization of computations for the linearized algorithm of the matrix multiplication $C=A*B$. Matrix B is dynamically subdivided into layers whose numbers are equal to the numbers of PEs available during the program execution. The layers are input into PEs. Matrix A is subdivided into layers $A_1, A_2, \dots, A_n$ so that the multiplication $C_{ij}=A_i*B_j$ should be possible in a single PE. Then the layers $A_1, A_2, \dots, A_n$ are input

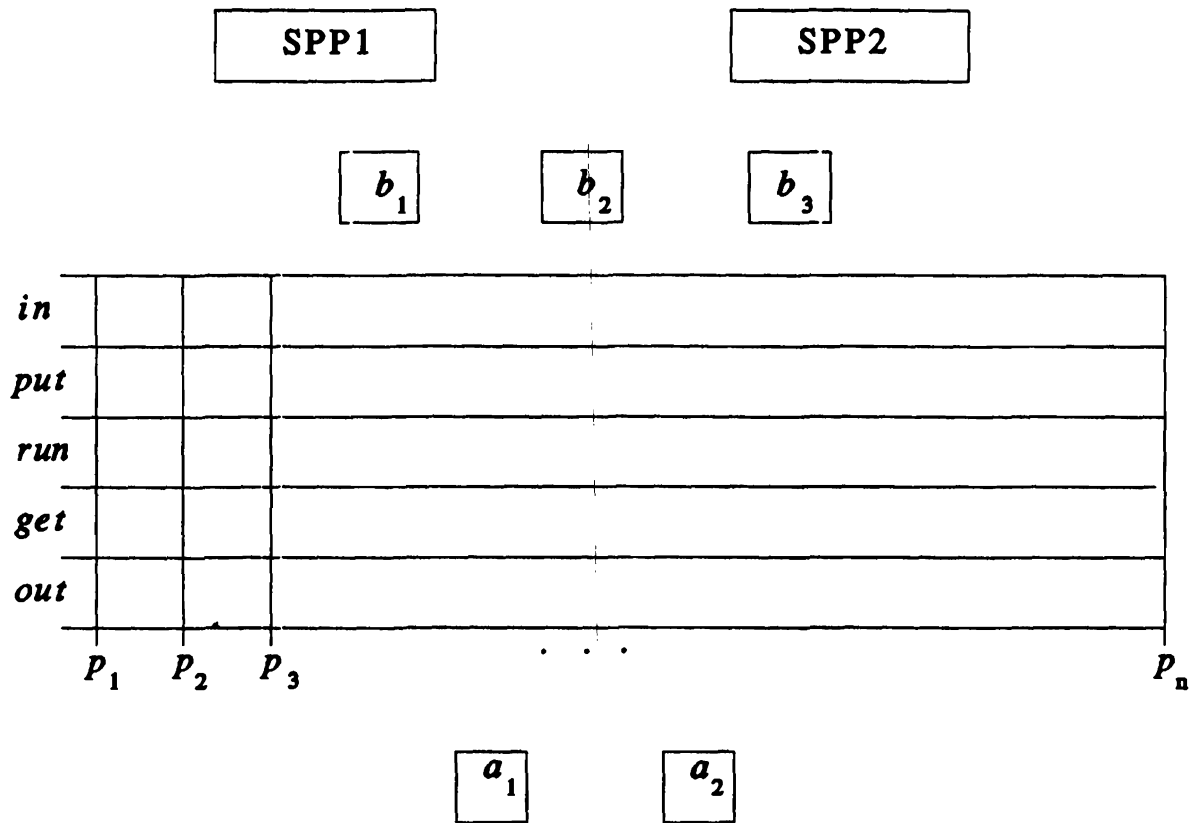


Fig.4

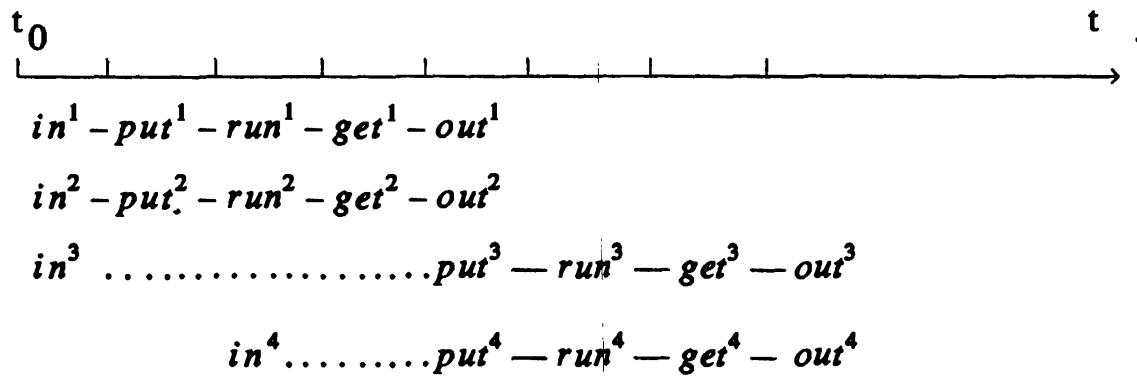


Fig.5

into the PEs one by one, each PE performs the calculations, then the layers of matrix A are shifted etc.

Under such organization of parallel computation the execution times of all the processes are almost the same. Such a restriction is unacceptable in the general case, but this condition can be easily

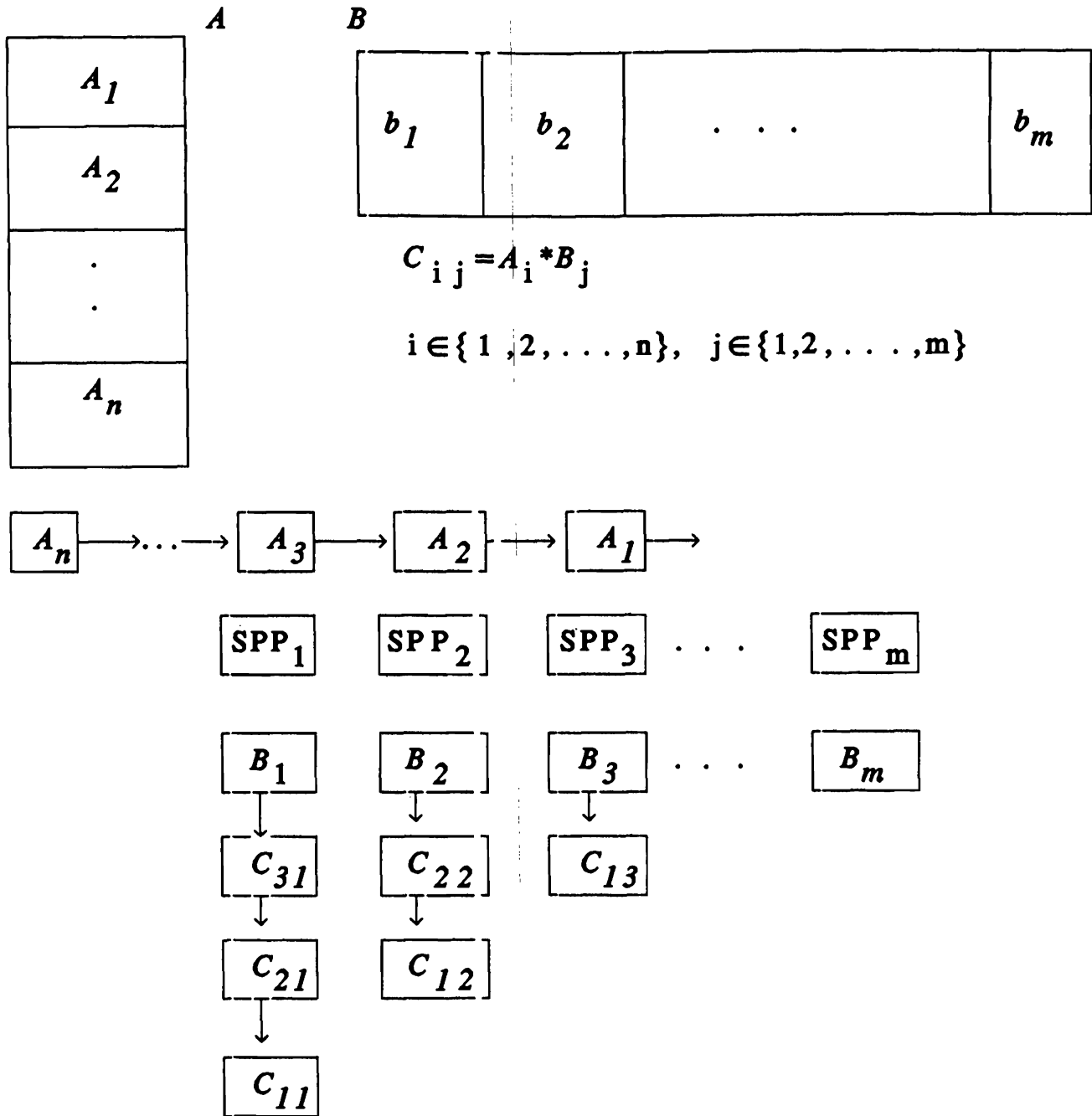


Fig. 6

satisfied for mass computation.

In implementing the linearized algorithm deadlocks can arise as is shown in Fig.7. Let the data exchange between PEs is performed through the channels, which provide linear interface. The processes must be synchronized, e.g. in order to transfer the the data block c_{12}^1 the processes p_1 and p_2 must wait for each other, transfer the data of c_{12}^1

(as shows arrow in fig.7) and then proceed with the asynchronous

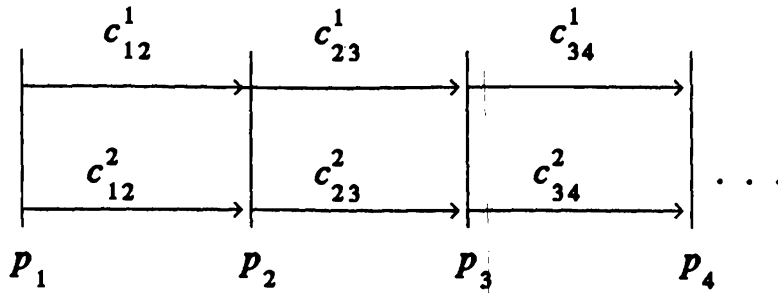


Fig.7

computations. If only two processors are available, the process p_1 will be executed by PE1, p_2 - by PE2, the process p_1 will be able to transfer the data block c_{12}^1 to the process p_2 and will continue the computation. Since the process p_3 is not executed (there is no PE available), the process p_1 will fail to transfer the data block c_{23}^1 to p_3 and will be in the suspended state. For the same reason process p_1 will be in the suspended state too when the c_{12}^2 will have to be transferred to the p_2 . If two PEs have the common memory as interprocessor interface or there are three or more PEs with linear interface, this algorithm will be successfully executed. Therefore the requirements of every program to the minimum sufficient resources of the MCS must be formed as one of the technical parameters of the program. There is naturally the upper bound of the necessary resources which may be due to the fact that it was impossible to prepare more data in time.

The linear algorithm and resource rolling can provide interactions in also not infrequent case when the MCS components have no common synchronization units. If the PEs shown in Fig.7 had the common cluster memory, a special memory region - asynchronous channel - could be allocated, where the processes could store the data and use them where necessary. The region of the asynchronous channel is also shared by the rolling and certain times t_0 and t_1 , $t_0 < t_1$, data blocks may be located as it is shown in Fig.8. To avoid the usual

competition of processes, the host process has precedence when locking out the memory of the asynchronous channel and names its only successor, which will get access to the memory. This successor must be named by a special statement of the programming language.

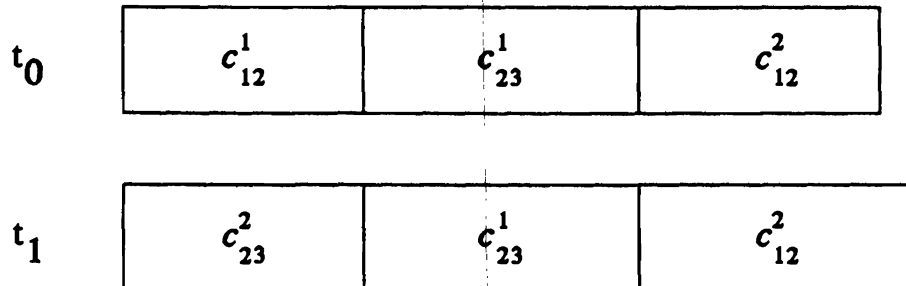


Fig.8

Therefore, the data exchange can be programmed in a different order, for example:

$$a) c_{12}^1, c_{23}^1, c_{34}^1, \dots, c_{(n-1)n}^1, c_{12}^2, c_{23}^2, \dots, c_{(n-1)n}^2$$

$$b) c_{12}^1, c_{12}^2, c_{23}^1, c_{23}^2, \dots$$

$$c) c_{12}^1, c_{23}^1, c_{12}^2, c_{34}^1, c_{23}^2, \dots$$

The resource rolling is not always effective. For instance, let the processes take essentially different PE time and not interact, or interact in such a way that the common memory is used as an interface and the interactions can be asynchronously executed. Then the process must be assigned to the first released PE instead of employing PEs rolling. Thus, by default, resource allocation in the PPS may be the rolling but a special statement must be available for the programmer in order to set another allocation algorithm.

4. Survey of PPS Inya

Program in PPS Inya⁸ is developed using Fortran to define the sequential computation; mass computation is described by means of special Inya statements. Therefore Inya program is an ordinary Fortran program (host process) with some special fragments of text to define mass computation. These fragments are enclosed in statement brackets

pardo and **parend**. The program fragment between **pardo** and **parend** is the program of process generation (PPG). The brackets **pardo** and **parend** specify the successive looping execution PPG. Under the i -th execution of PPG the i -th process is generated. The process generation concept includes the PEs allocation, preparation of the input data and their transfer to the PE, creation of the process program, its transfer to the PE, execution of this program by PE and the results transfer to the host process.

To transfer the data between the i -th and j -th processes the statement **pass** is used. The type of connection between available PEs when the i -th and j -th processes are running determines how the **pass** statement will be executed. The choice is made by process generation. If, for instance, the PEs are connected by a bus, **pass** synchronizes the processes before data are transferred (rendezvous). If the PEs have common memory, asynchronous execution of the **pass** statement is possible. During execution the PP has information on the structure of the MCS on which the PP is being executed.

The processes are synchronized by **wait** statement. The statement **sect** delay the execution of $(i+1)$ -th process until the statement **post** is executed in the i -th process.

Statements defining the times of locking out the resources and their release are used to ensure the effective sharing of the resources.

The processes lock out necessary resources during the execution, resource allocation by default is rolling. However, as for PEs, the user can either choose the rolling or assign the process to the first available PE. A similar approach is planned for controlling common memory. Some more resource allocation algorithms will be additionally embedded into Inya. Than the user using them as elementary algorithms of specific resource allocation could assemble his own algorithm of locking out all the resources required by the processes.

PEs are allocated in PPS Inya by the MCS resources management subsystem⁹. This subsystem enables one to lock out a given number of PEs, say 5, or some numbers of PEs, say all available PEs, or any number within the range, say from two to eight. The task can call

for a specific structure e.g. ring, network of not less than three PEs connected by channels etc. For more detailed information on PPS Inya, the reader is referred to⁸⁻⁹.

Summary

The large-block high-performance linear hierarchical MCSs like MCS SIBERIA is currently used in many firms and research centers of USSR for seismic data processing, signal and image processing, modeling of mass natural phenomena etc.

On the basis of parallel program synthesis method⁵ for the Inya PPS new nonprocedural facilities are now being developed. First, the facilities for the description of data input/output operations have been developed because their illtime execution is often the cause of a slump in MCS performance. Now the intelligent subsystems for the description of modules and their including in the PP, for the description of coarse grain computations with needed details are under development.

REFERENCES

1. E.J.Clementi and others, *Parallel computing*, V.5(1987), pp. 13-44.
2. iPSC/2 *User's Guide*, Intel Scientific Computers, Beaverton, Oregon(1988).
3. T. Manuel, in *EUSA* No.18(1988),pp.110-112.
4. V.E.Malyshkin, *Programmirovaniye*, N.4,(Moscow,1991).
- 5& V.A.Valkovskii, V.E.Malyshkin, *Syntheses of parallel programs and systems on the bases of computational models*.(Nauka, Novosibirsk, USSR, 1988).
6. V.A.Torgashev, V.U.Plyusnin, V.M.Ponomaryov, in *Electronic Computer Engineering*(Radio i svyaz, Moscow,1988), pp.172-182
7. S.Markov and others. *Supercomputer*, V.31, VI-3,(1989), pp. 12-23.
8. V.A.Anisimov, V.E.Malyshkin, N.N.Mirenikov, *Parallel programming language for large-block parallel systems*.
Preprint ~ 824,825, (Novosibirsk, Computing Center, 1989)
9. N.V.Kutchin, V.E.Malyshkin, S.A.Pankratov, in *High-performance computer systems and their software*(Computing Center, Novosibirsk, 1987), pp. 29-37.

Kinetical-consistent finite-difference gasdynamic schemes and their solution on multiprocessor computers

T.G.Elizarova, A.E.Dujsekulov, L.V.Kosarev

National Center of Mathematical Simulation

USSR Academy of Science, Moscow 125047, Miusskaya sq.,4

June 14,1991

Abstract

In this paper we consider the implementation of massively parallel multiprocessor computer systems for simulation of spacial flow problems.

We shortly describe a computational algorithm convenient for parallel computing and give examples of stationary supersonic flow around an obstacle and unstationary flow arising in boundary layer - shock wave interaction. The program package KIPARIS, based on the mentioned algorithm is described. The computational results were obtained on multitransputer systems Hathi-2 and APS-48.

1 Introduction

Multiprocessor computing systems based on transputer elements have in recent years become an outlooking tool for modern computational gasdynamics. There are several reasons for it. The first reason is the computationally heavy numerical methods used in viscous heatconducting compressible flows, implementation fine space computational grids for stationary problems and also small timesteps for unsteady problems. The second reason is the wish to compute 3-D flows for practical applications. The flows in nature are three-dimensional and unsteady, and the results that we can get based on more simple numerical simulation models may strongly differ from the results obtained from the full models. A third reason lays in the fact that the mentioned computational problems have inner structural parallelism, and the use of parallel computers for them seems to be natural.

In [4], [5] a new algorithm for Navier-Stokes equations was created. It was called kinetical-consistent finite-difference gasdynamic schemes (KCFDS). Based on KCFD schemes a number of flow problems both stationary types (flow near back and right step, flow in cavities) and unstationary types (flow in cylindrical cavities and near spiked cylinders) have been carried out [5], [7], [6]. In [1] and [2] problem of selecting algorithms convenient for parallelisation was discussed and it was shown that explicit forms of KCFD schemes on regular computational grids is well suited for parallel implementation on MIMD computers.

In the present paper we discuss computational results based on the KCFD method and program package KIPARIS. Three-dimensional flow computations have been carried out on a rectangular transputer networks. The results were obtained on the Hathi-2 system in Abo-Akademy and on APS-48 transputer system in Center of Informatics and Computer Technology, Sofia. A hight computational efficiency of the systems was shown.

The same algorithm may be implemented on the other distributed memory systems with local inner communications up to single-instruction-flow systems such as n-CUBE and connection-machine.

This paper is essentially based on [3] and extends previous results.

2 KCFD schemes and the problem statement

For the numerical simulation of 3-D flows we have used KCFD schemes in the following form:

$$\begin{aligned} \rho_t + (\rho u)_x + (\rho v)_y + (\rho w)_z = \\ (\tau^x(\rho u^2 + p)_x)_x + (\tau^y(\rho v^2 + p)_y)_y + (\tau^z(\rho w^2 + p)_z)_z \end{aligned} \quad (2.1)$$

$$\begin{aligned} (\rho u)_t + (\rho u^2 + p)_x + (\rho uv)_y + (\rho uw)_z = \\ (\tau^x(\rho u^3 + 3pu)_x)_x + (\tau^y(\rho uv^2 + pu)_y)_y + (\tau^z(\rho uw^2 + pu)_z)_z + \\ M_\infty/Re_\infty((\mu u_x)_y + (\mu u_z)_z) \end{aligned} \quad (2.2)$$

$$\begin{aligned} (\rho v)_t + (\rho uv)_x + (\rho v^2 + p)_y + (\rho vw)_z = \\ (\tau^x(\rho u^2 v + pv)_x)_x + (\tau^y(\rho v^3 + 3pv)_y)_y + (\tau^z(\rho vw^2 + pv)_z)_z + \\ M_\infty/Re_\infty((\mu v_x)_x + (\mu v_z)_z) \end{aligned} \quad (2.3)$$

$$\begin{aligned} (\rho w)_t + (\rho uw)_x + (\rho vw)_y + (\rho w^2 + p)_z = \\ (\tau^x(\rho u^2 w + pw)_x)_x + (\tau^y(\rho v^2 w + pw)_y)_y + (\tau^z(\rho w^3 + 3pw)_z)_z + \\ M_\infty/Re_\infty((\mu w_x)_x + (\mu w_y)_y) \end{aligned} \quad (2.4)$$

$$\begin{aligned} E_t + (u(E + p))_x + (v(E + p))_y + (w(E + p))_z = \\ (\tau^x(u^2(E + 5/2p))_x)_x + (\tau^y(v^2(E + 5/2p))_y)_y + (\tau^z(w^2(E + 5/2p))_z)_z + \\ (\tau^x/(\rho(\gamma - 1))p^2_x)_x + (\tau^y/(\rho(\gamma - 1))p^2_y)_y + (\tau^z/(\rho(\gamma - 1))p^2_z)_z + \\ M_\infty/Re_\infty(F + \gamma/Pr((\mu \varepsilon_x)_x + (\mu \varepsilon_y)_y + (\mu \varepsilon_z)_z)) \end{aligned} \quad (2.5)$$

The dissipative function F has the following form:

$$F = 0.5((\mu u^2_x)_x + (\mu v^2_y)_y + (\mu w^2_z)_z) \quad (2.6)$$

where

u, v, w are the velocity components,
 p is pressure,
 ρ is density,
 $E = \rho((u^2 + v^2 + w^2)/2 + \varepsilon)$ is energy,
 ε is internal energy,
 $\gamma = c_p/c_v$, $p = \rho(\gamma - 1)\varepsilon$ is the state equation,
 Pr is the Prandtl number,
 $Re_\infty = \rho H u_\infty / \mu_\infty$ is the Reynolds number,
 $M_\infty = u_\infty / c_\infty$ is the Mach number,
 c is sound velocity,

$\tau^x = h^x/2V$, $\tau^y = h^y/2V$, $\tau^z = h^z/2V$ is the characteristic time, determined by the computational grid step h^x , h^y , h^z , and

$V = c + \sqrt{u^2 + v^2 + w^2}$ is the characteristic gasdynamic velocity, i.e. the average time for a gas-particle to pass a cell of a computational grid.

We have used the viscosity μ in the Sutherland form $\mu = \mu_\infty(\varepsilon/\varepsilon_\infty)^\omega$, where $\omega = 1$ or 0.5 .

In (2.1) - (2.5) the following dimensionless relations were used:

$$x = H\tilde{x}, u = c_\infty\tilde{u}, \rho = \rho_\infty\tilde{\rho}, \mu = \mu_\infty\tilde{\mu}, p = \rho_\infty c_\infty^2\tilde{p}.$$

Then

$$u_\infty = M_\infty, \rho_\infty = 1, p_\infty = 1/\gamma, \varepsilon_\infty = 1/(\gamma(\gamma - 1)) \quad (2.7)$$

The finite-difference scheme (2.1) - (2.6) must be completed with the boundary conditions: on the walls of the step in the form

$$u = v = w = 0, \partial p/\partial n = 0, \partial \varepsilon/\partial n = 0$$

drift or soft conditions at the outlet of the domain in the form

$$\partial f/\partial n = 0, \text{ where } f = (u, v, w, \rho, \varepsilon)$$

and come special conditions at the entrance of the computational domain.

3 Parallel implementation of spacial algorithmes

Let us consider the numerical solution of the explicit finite difference schemes on transputer network and data parallelism for regular computational grids consisting of rectangular cells.

There are several approaches for geometrical decomposition in 3-D computational problems. Let us examine two of them. The first one is matching 3-D problem directly to the 2-D processor grid and putting third space-coordinate into the slave processor memory. Second approach provides to connect transputer elements in pairs, then to construct 3-D processor grid from the pairs and than to match 3-D computational problem to the 3-D processor grid.

Let P denotes the number of computing elements and N denotes the number of computational grid points. Let us assume also that the computational domain is parallelepiped. Then in the first approach the number of computational nodes in one processor will be

$$N^{1/3}/P^{1/2} * N^{1/3}/P^{1/2} * N^{1/3}$$

So the communication time will be

$$T_{com} = t_{com} * N^{1/3}/P^{1/2} * N^{1/2}$$

Calculation time for single processor is $T_{cal} = t_{cal} * N/P$. Than the relation between communication and calculation time is

$$\beta_1 = T_{com}/T_{cal} = t_{com}/t_{cal} * \sqrt{P}/\sqrt[3]{N}$$

where t_{com} - is communication time for one real number t_{cal} - is calculation time for one grid point

In the second approach we should combine transputer elements in pairs to have the processor element with six links for external communications and one common link for inner communication. The scheme of double computing element is shown in Fig 1.

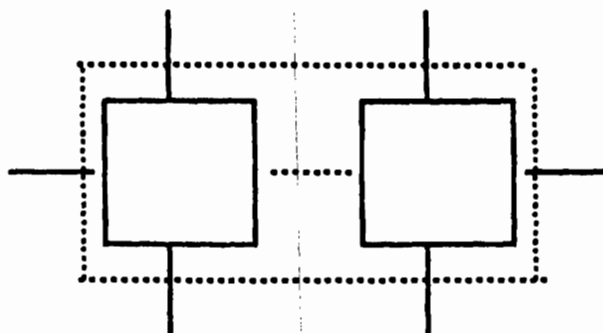


Figure 1: Double transputer element for 3-D network

The number of the computational grid points in a coupled processor element would be

$$\sqrt[3]{N}/\sqrt[3]{P/2} * \sqrt[3]{N}/\sqrt[3]{P/2} * \sqrt[3]{N}/\sqrt[3]{P/2}$$

To prepare external communications the inner communications must be done. It will take time

$$T_{com} = t_{com} * \sqrt[3]{N}/\sqrt[3]{P/2} * \sqrt[3]{N}/\sqrt[3]{P/2}$$

The time for external communications will be the same. The number of computational grid points are the same as in the first approach and equal N/P , so the relation between communication and calculation time is

$$\beta_2 = 2 * \sqrt[3]{4 * P/N} * t_{com}/t_{cal}$$

The relation between β_1 and β_2 clarifies the validity of the examined approaches.

$$\beta_1/\beta_2 = \sqrt[3]{P/2} * \sqrt[3]{4}$$

So the connection transputer elements in pairs is more suitable if $\beta_1 > \beta_2$ that corresponds to the processor number $P > 1024$. In real used transputer systems with $P \sim 100$ the first parallelisation approach is more convenient and restricted only by the memory volume in slave processors.

In the examined flow problem we have used 2-D data parallelism and put the third space-coordinate into the slave processor memory. Rectangular computational domain was divided in rectangular subdomains with minimal perimeters (to minimized the amount of data to be transferred) and each subdomain was assigned to a corresponding processor.

4 Short description of the package KIPARIS

The program package KIPARIS is developed for numerical simulation of a number of fluid dynamic problems on transputer networks. The computational algorithm is based on kinetical-consistent finite-difference schemes mentioned above. The name of the package was constructed based on the key words kinetical (KI), parallel (PAR), simulation (IS).

With a transputer multiprocessor system we mean a system typically consisting of three main hardware components:

- a host computer system - a personal computer IBM-PC AT, with external memory not less than 640 Kb

- a root transputer, which is installed in the host computer system, with at least 2 Mb memory. The root transputer can for instance be mounted on an Inmos B004 or Inmos B008 transputer board.
- a transputer network, consisting of any number of elements.

The package KIPARIS is implemented in the operating system IBM PC-DOC or MS-DOS. The programs are written in Occam-2.

The package KIPARIS runs on any rectangular transputer network. When increasing the network dimensions in sufficiently large scale, the efficiency of the system increases almost linearly. The results of the computation can be presented to the user without interrupting the network running. The way of result visualisation was described in [3]. To execute the package on a transputer network, it must be configured in a matrix topology, with more than three columns and two lines. It is also possible to use a pipeline containing more than three transputers except the root. The root transputer does not take part in the computations and only communicates with the host computer.

The nodes of the computational grid are distributed between the processors in such a way that every processor has an equal number of grid points. The computational grid is irregular. The number of computational grid points is chosen by the user when he creates the file with the initial data and is restricted by the volume of external memory on the root transputer. When KCFD scheme running each transputer in the network communicates with the four neighboring processors in the grid. In the current version of the program, communications are not done in parallel with computations. Communications account for about 10 percent of the total execution time.

The common program structure for the slave transputers and the communication algorithm for neighboring transputers was presented in [3].

The installation of an application program onto a concrete processor network topology can be made automatically in an interactive mode.

In present version of the package, the user must first configure the transputer network into a pipeline, matrix or torus. When the package is installed onto a processor network, the network topology will be automatically recognized and a picture of the network will be presented on the screen. In the recognition process all necessary network parameters will be determined, in particular the number of columns and rows, the numbers of links and how the links of the processors are connected to each other. As a result of the execution of this program, the configuration file for the application problem will be automatically created and then the program will be loaded onto the processor network. This makes it possible to release the user from a considerable part of routine work needed to create configuration file (Fig.2).

5 Experiments with multiprocessor systems

The program package KIPARIS was implemented on the Hathi-2 multiprocessor system in Abo-Akadeemy, Finland, and on the APS-48 multiprocessor system in the Centre of Informatics and Computer Technology, Sofia. An IBM-PC computer was used as a host-machine, with the programming system TDS D700D.

Hathi-2 is a reconfigurable transputer-based multiprocessor system consisting of 100 IMS T800 processors, each currently with 1.25 Mbytes of local memory. The Hathi-2 system has a distributed switching network built of IMS C004 crossbar switches, which makes it possible to configure the processors in the system into different topologies like grids, toruses, trees, etc. The switching network is controlled by a separate control system, consisting of 25 IMS T212 transputers. The control system also contains hardware that support nonintrusive performance monitoring of application programs.

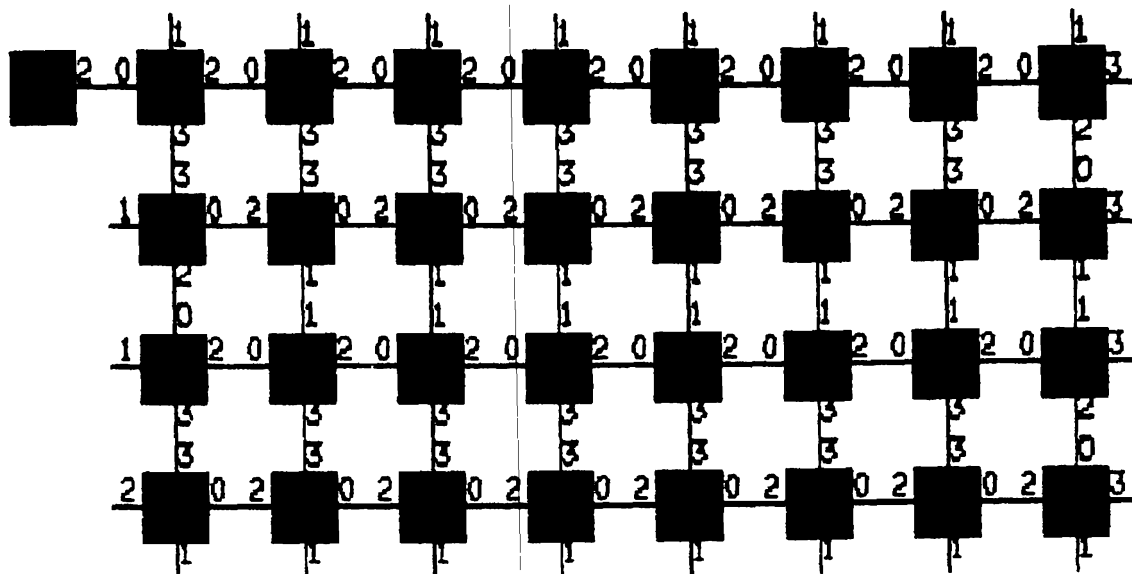


Figure 2: Picture of processor network obtained in interactive mode from package KIPARIS. Processor numbers show the order in which the network tester recognizes the processors.

APS-48 system consists of 32 IMS T800 processors, each currently with 2 Mbytes of local memory, root transputer with 4 Bbytes and graphical board. The system also is supplied with two 20 Mbytes hard disks.

In Hathi-2 system two types of network topologies were used: a 32 processor matrix network (4×8) and a 64 processor matrix network (8×8). In APS-48 system only the first type of network topology was used.

5.1 Description of the computational examples

Several examples of the flow near the step and unsteady flow in the boundary layer-shockwave interaction were computed.

In the case of a flow near the step for the first example the computational grid consists of $N_x = 42$, $N_y = 26$, $N_z = 26$ points. The total number of points is 28.392. The minimal step of the grid, in all three directions, is $h_m = 1/9$. In the region of the separate flow near the obstacle, grid steps are minimal and equal to each other. The entrance parameters for this example are: $M_\infty = 3$, $Re_\infty = 1000$, $Pr = 0.72$, $\omega = 1$, $\delta = 0.3$, $L = 2$. The iteration number for the convergence point was 10.000. The computational time in 32 transputer network was 4 hours and 40 minutes. When computing the same example on 64 transputer network the computational time decreases exactly with a factor of 2, giving a linear speedup.

The computational time for 500 iterations in a 32 processor network was 14 minutes and 2 seconds, while in the 64-transputer network it was 7 minutes and 3 seconds without taking into account the time for writing out the checking points.

Several other examples of mentioned problem with different initial data were computed on a more detailed computational finite-difference grid consisting of $N_x = 58$, $N_y = 32$ and $N_z = 32$ points (the total number of points is 59.392). The minimal space step is $h_m = 1/12$. 13.000 iteration steps were made to achieve convergence. The program execution took 6 hours 13 minutes [3].

Isobar patterns for the first computational example are shown in Figures 3 - 4. The problem of unstationary shockwave - boundary-layer interaction was computed for the initial data: shock-wave Mach number $M_\infty = 2,52$, $Re_\infty = 5 \cdot 10^3$, pressure jump in the initial shock is $p_1/p_2 = 8,84$. $Pr = 0.73$, $\gamma = 1,29$.

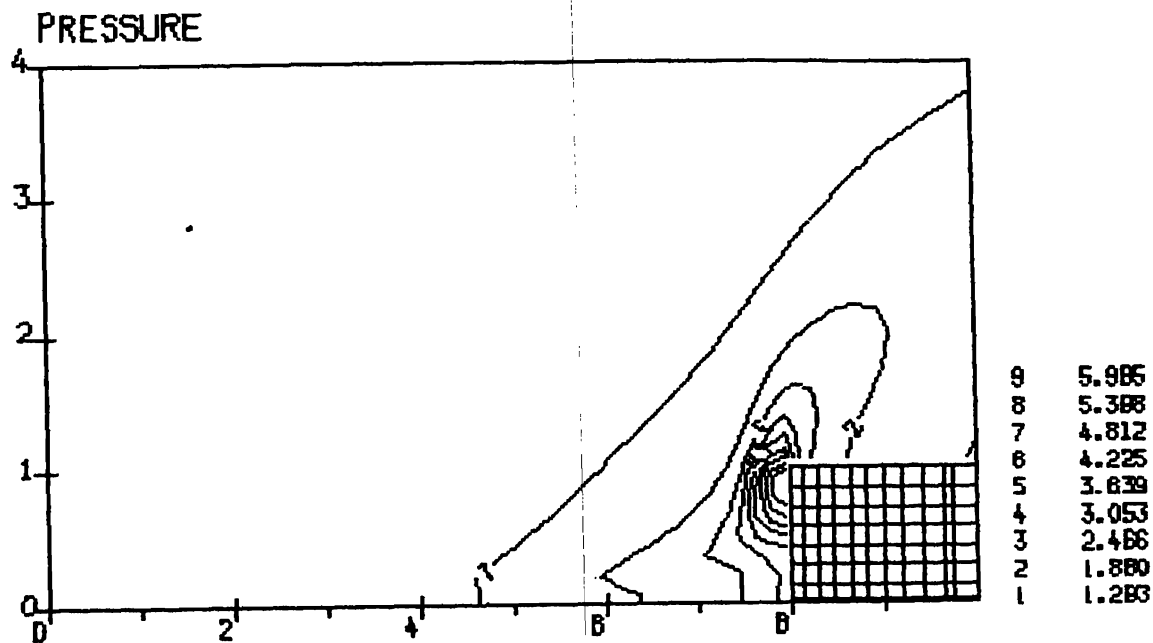


Figure 3: Isobars in plane of simmetry

Isobars in 2-D case for nondimensional time $t = 200$ is shown in Fig 5. Computations for spacial grids 300×150 and 10200 iterations takes approximately 100 minutes in 32 IMS T800.

The implementation on the massively transputer systems gave authors the opportunity to calculate three dimensional viscous supersonic flows and to achieve new information about its physical structure. These results for 3-D flows were obtained for the first time and they differ from the corresponding results for 2-D problems. In particular, position of two-dimensional leakage streamline before the step was found. This 3-D leakage makes pressure distribution on the step monotonic while in 2-D case it has extremum.

5.2 Estimations of the efficiency

It is well known that an accurate estimation of the performance efficiency of a parallel system in computing some concrete numerical task is a rather complicated problem and may be done for example, by observing the CPU and link activity in the transputer network.

Performance estimation for Hathi-2 system with package KIPARIS was done in [3]. For 64 IMS T800 it was approximately 40 Mflops. Communication takes about 10 percent of total execution

ime. The similar results were obtained for APS-48 system.

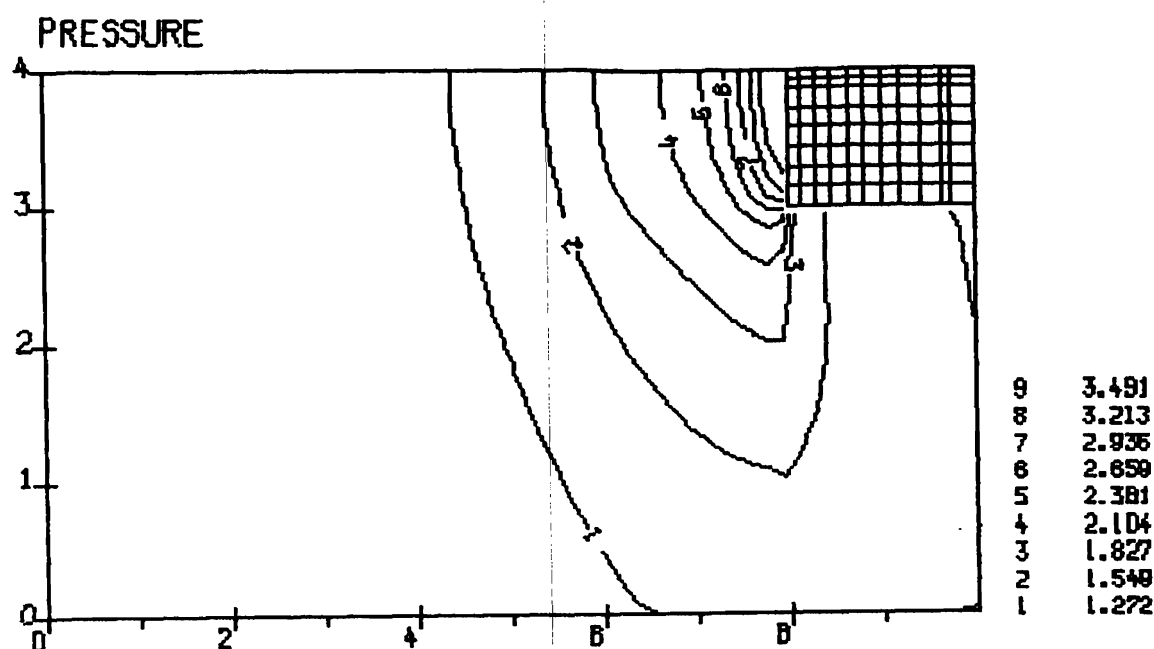


Figure 4: Isobars in horizontal section

6 Conclusions

The main goal of the presented work was to show experimentally the possibility of computing complex three-dimensional viscous gasdynamic flows on massively parallel transputer systems.

This goal was achieved by implementing the program complex KIPARIS, especially created based on a new computational algorithm, on the Hathi-2 and APS multiprocessor system.

The numerical experiments carried out have given new physical results about the flow structure and have shown a high performance efficiency of the implementation of KIPARIS on multiprocessor systems. The achieved efficiency was about 40 Mflops for a network consisting of 64 IMS T800 processors. We also show that the speedup of the implemented program increases linear with the number of processors.

It was also shown that the package KIPARIS can be easily installed on different regular transputer networks.

Our long-time goal is further generalization of the package and using it for computer simulation of complex application problems on massively parallel systems.

7 Acknowledgments

We would like to acknowledge Ivan Dimov and Krassimir Georgiev for many helpful discussions related to this work. We also thank the Center for Computer Science and Technology of Bulgarian Academy of Sciences for allowing us to use their facilities.

References

- [1] Dujsekulov A.E., Elizarova T.G. The application of multiprocessor computers for the solution of kinetical-consistent finite-difference gasdynamic schemes. J. Math. Model., 1990, v.2, N 7 p. 139-147 (in Russian).
- [2] Elizarova T.G. Kinetical-consistent finite-difference gasdynamic schemes and their solutions on multiprocessor computers. Transputer Initiative, Mailshot, February 1991, p. 42-48.
- [3] Dujsekulov A.E., Elizarova T.G., Aspnas M. An implementation of gasdynamic problems on multiprocessor systems. Reports on computer science and mathematics, Ser. A, No 123, 1991 Abo-Akademy, SF-20500, Abo, Finland.
- [4] Elizarova T.G., Chetverushkin B.N. One computational algorithm for calculating gasdynamic flows. Rep. Ac. of Science., USSR, 1984, v.179, N1, p. 80-83 (in Russian).
- [5] Elizarova T.G., Chetverushkin B.N. Kinetical-consistent finite-difference schemes for simulation viscous heat-conducting gas flows. J. Comp. Math. and Math. Phys., 1988, v.28, N 11 p. 695-710 (in Russian).
- [6] Elizarova T.G., Chetverushkin B.N. Computer simulation of some types of flows arising at interactions between a supersonic flow and a boundary layer, Lecture Notes in Physics, 11-th Inter. Conf. on Num. Meth. in Fluid Dynamics, Springer-Verlag, 1988, p. 245-250.
- [7] Elizarova T.G., Chetverushkin B.N. Kinetical-consistent finite-difference gasdynamic schemes.-Intern. Symp. on Comput. Fluid Dynamics.-Nagoya, Japan, 1989, p. 501-506.

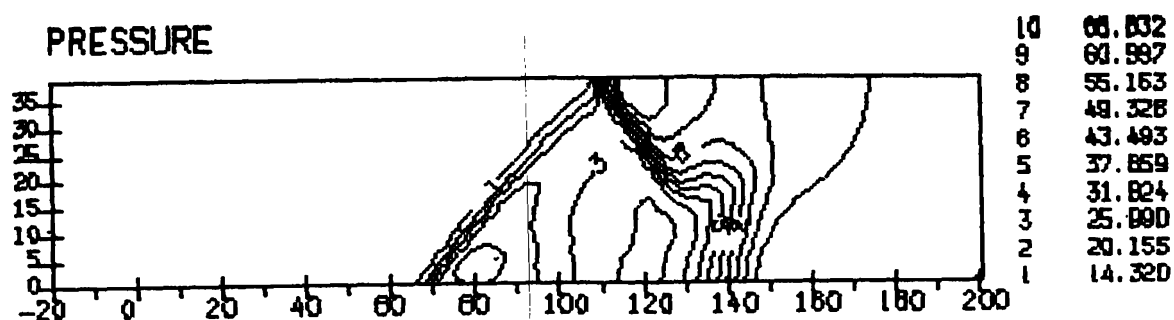


Figure 5: Isobars for shockwave-boundary-layer interaction

CONSTRUCTION OF PARALLEL PROGRAMS FOR MULTIPROCESSORS IN COMPUTATIONAL PHYSICS PROBLEMS

D.P.KOSTOMAROV, A.V.PEDORENKO, A.M.POPOV

**Department of Computational Mathematics and Cybernetics
M.V.Lomonosov Moscow State University, Moscow, USSR**

In this paper we consider the efficient organization of parallel programs for multiprocessors for long-scale simulations in computational physics. The programming system for modeling of complicated interaction of basic fortran-modules in the process of computational physics simulations for multiprocessors is proposed. SAPA system for parallel program construction is created, and the technology of parallel programs adaptation for multiprocessors is described.

1.INTRODUCTION

Basic considerations of the report are following:

- Large-scale simulation and hybrid modelling in plasma physics seems to be new and complicated step in computational physics.
- Computational process must be periodically reconstructed due to the problems nonlinearity and multidimensionality.
- For multiprocessors this means the multi-tasking organization of computations with hierarchical structure.

Nowadays, the actual problem is the efficient use of multiprocessors for real problems in large-scale simulations.

In this report we consider the efficient organization of parallel programs for multiprocessors for large-scale simulations in computational physics. The programming system for modelling of complicated interaction of basic fortran-modules in the process of computational physics simulations for multiprocessors is proposed. SAPA system for parallel program construction is created, and the technology of parallel programs adaptation for multiprocessors is described.

2.HIERARCHY OF MODELS AND COMPUTER CODES IN PLASMA PHYSICS SIMULATIONS

The processes in high-temperature plasma have a long time-scale. Different areas of applied models of these processes and corresponding computer codes are presented in logarithmic scale on Fig.1(L and a are the typical perturbation scale and the physical device size respectively). The typical time-scales are following: τ_A - Alfvén time, τ_R - resistive time with $10^8 > R_{em} > 10^2$ (R_{em} - magnetic Reynold's number) and γ^{-1} - MHD-instability time.

The lines $t = x^2 \tau_R R_{em}$ for various values of magnetic Reynold's number R_{em} are shown. $R_{em} \sim T^{-3/2}$ is a function of plasma temperature: $R_{em} = 5 \cdot 10^2$ ($T = 0.01$ Kev, line 1), $R_{em} = 0.5 \cdot 10^6$ ($T = 1$ Kev, line 2), $R_{em} = 0.4 \cdot 10^8$ ($T = 20$ Kev, line 3).

The models based on the particle separation are in the down-left corner Fig.1 (area 1). Here the codes with Monte Carlo method application and 3-dimensional codes for kinetic equations are placed. The ideal magnetohydrodynamics (MHD)

models are placed in the area 2 under line 1. In these models plasma in a strong magnetic field is regarded as a solid stretch. The processes with dissipation are placed in the area 3. This models differ from models of area 2 by a small resistive layers, where plasma has other properties. In the models of diffusion (area 3) the diffusion displacement of magnetic field through the plasma may be compared with plasma size. It is the transport codes of energy balance area. Nowadays the hybrid codes are most perspective. These models take into account the influence of MHD-instabilities on the diffusion coefficients. They are placed in 2 and 3 areas.

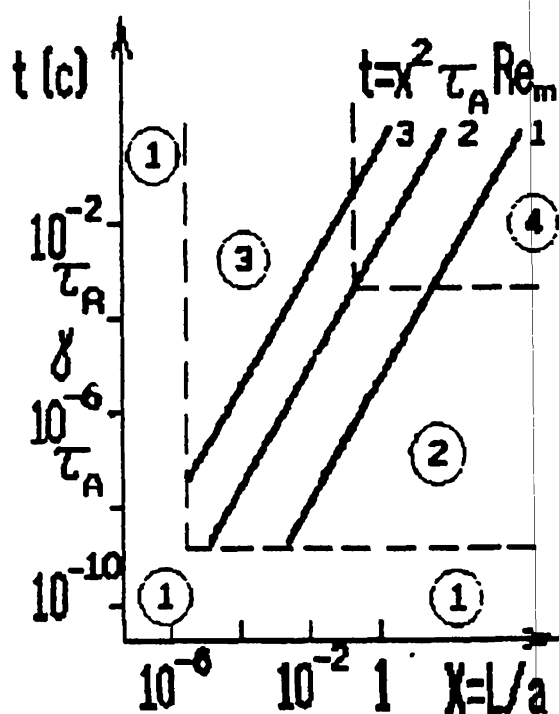


Fig 1.

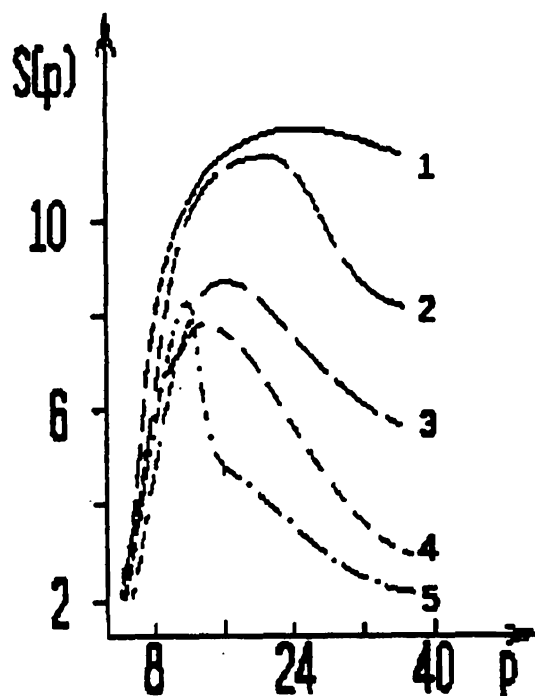


Fig 2.

Evidently oscillations correlated with pressure and magnetic pressure and dissipative processes develop on a totally different time-scales. So, modern computational methods are needed for hybrid codes construction.

3. REQUIREMENTS TO MULTI-PROCESSORS PARAMETERS FOR LARGE-SCALE SIMULATION IN PLASMA PHYSICS

3.1 MHD-codes

The system of equations which describes the nonlinear MHD-evolution of plasma is complicated because of multi-dimensionality and the presence of small parameter before main derivative which leads to the solutions with a resonant surfaces.¹ So, numerical solution is difficult and demands more complicated algorithms and high-performance computational systems with parallelism and performance about 10^3 MFlops

We can evaluate computational resources we need. Three-dimensional computational MHD-models have 7 equation in partial derivatives. So, if we take a grid with N points per direction, we'll have N^3 points. The computation of unknown functions in each point for one time step requires more then 20 float-point operations. If the steps number is J , the total number of operations equals $140 \cdot JN^3$. Minimal number of time steps can be obtained by physical parameters:

$$J = 20 t_{sk} N \sqrt{\beta R_{em}} \quad (1)$$

where β is a ratio of plasma's pressure to its magnetic field pressure, t_{sk} - time-scale of diffusion processes (it is typical time of MHD-simulation). Number N of grid points depends on the size of dissipative layer. Usually we need 10 points per layer, so $N \sim 108 R_{em}^{1/3}$ and $J = 200 t_{sk} \beta^{1/2} 8 R_{em}^{1/3}$.

In modern experiments typical parameters are following: $t_{sk} \sim 1$ ms, $\beta \sim 0.05$, $T \sim 1$ Kev. So, number of operations for simulation of real MHD-process must be

$$U = 140 J N^3 = 5 \cdot 10^4 t_{sk} R_{em}^{4/3} \sim 2 \cdot 10^{13} \text{ oper} \quad (2)$$

The volume of RAM must be $V = 7 N^3 \sim 700$ Mbyte. Run-time per one task must not be greater than 1 hour what requires $r = 10^7$ MFlops of computer performance.

Similar MHD-computations (with $N \sim 200$ -800 point per direction) were run on CRAY-2 computer² with 10^3 MFlops performance. More then tens and hundreds hours were needed for these computations.

3.2 Monte Carlo code for plasmas kinetic problems simulation

Monte Carlo method is based on direct simulation of trajectories of the separated particles from statistic ansamble in the phase space. Macrocharacteristics are defined as average over the ansamble. Mathematical task consists of solving the system of ordinary differential equations which have accidental functions in the right side.

The class of the kinetic tasks is so large that there is no overall demand to computational system. So, we consider typical problem of injection of hot particles into tokamak's plasma.

The main parameters are N_0 - particle number in statistic ansamble, τ_E - particle squeeze time to thermal velocity, τ_b - particle's moving time along the trajectory. The main difficulty is that $\tau_E \gg \tau_b$. In real tasks $N_0 \sim 10^3$ - 10^4 , $\tau_E = 10^{-7}$ s, $\tau_b = 10^{-2}$ s $J = 10^7$ time-steps are needed for the simulation of this process and number of operations is $U = N_0 N \alpha$ (where α is the number of operations per step of simulation,

$\alpha \mp 100$). This means that $U \sim 10^{12} - 10^{13}$ of operations and if the computer has $r=1$ GFlops performance the runtime of this code would be approximately 20 min $\sim$ 3 hours.

3.3 Magnetic field computation

The problem of magnetic field computation may be reduced to full $N \times N$ matrix reversion. For the grid points number $N \sim 10^3 - 5 \cdot 10^3$ number of operations is $U \sim N^3 = 10^9$. Demand to RAM is $V = 5-10$ Mbytes. Main difficulty is solving of the system with ill-conditioned matrix because of truncation errors.

4. NEW NUMERICAL APPROACHES AND ORGANIZATION OF COMPUTATIONS

Specific code's development in different branches of plasma physics is nearly exhausted. Nevertheless, the processes outlined above are complicated and single code can't describe them. The hybrid model's development is a new and difficult step in plasma computer simulation.

Special MHD-codes describe the development of a low-scale unstable modes. Due to this development magnetic surfaces reconnect and change the diffusion coefficients. Transport model with new coefficients leads to new bifurcations and to generation of a new MHD-instabilities. Amalgamation of this processes gives the hybrid code. The interaction of Fourier harmonics is the main part of computational process in this code. In other words it is the evolution of convolution

$$C_{m_0, n_0}^k(i) = L(i) \sum_m^M \sum_n^N \sum_l^L A_{m,n}^l(i) B_{m-m_0, n-n_0}^{l-k}(i) \quad (3)$$

This sum is computed in each point of space-grid and on each timestep for modes with indexes $m, n, m_0, n_0, m + m_0, n + n_0, k, l$ which are MHD-unstable.

In traditional codes the set of MHD-modes is fixed at the beginning and their evolution is observed. This approach is very unefficient because of the requirement to store and to sum stable harmonics which are equal to zero for a long time and have no influence on system's development. On the other hand, the artificial truncation of the mode's set leads to numerical instability of this method in comparison to initial set harmonics .

We consider the new approach to simulation in hybrid models. We propose the method of projection of solution on the unstable manifold where only unstable harmonics are investigated ³. One moment MHD-computations are interrupted, diffusion coefficients are changed to corresponding to MHD-activity, and then transport model is integrated. New equilibrium is investigated by MHD-codes. The unstable manifold's dimension and number of unstable modes is changed. This leads to reconstruction of the computations inside the process. Following conditions define the way of process: either system come to a total stochastisation and computations must be interrupted or the process is repeated.

This approach not only leads to the study of algebraic tasks but also to the problem of harmonics sortification in the convolution. The typical hierarchical module structure of the code corresponding to multitasking structure of computations is very efficient for multiprocessors.

5. DIFFICULTIES OF EFFECTIVE MULTIPROCESSORS APPLICATION

We have an experience of ISOT-1014 multiprocessor computer application. This computer consists of ES-1037 host-computer and four ES-2706 array processors (analogous to AP-120 by FPS corporation with 10 MFlops peak performance). This is very advanced system but there are great difficulties in programming because of high performance and limited RAM of AP and low speed of host-AP interchange. So, the problem is in the construction of large Fortran-modules with minimum data exchange. The speedup of parallel program with U operations and S_a data exchange operations on multiprocessor may be presented as follows:

$$S(p) = \frac{p}{1 + \left[\frac{S_a}{U} \mu_0 + \frac{L_a}{U} \mu_{0s} \right] p^2} \quad (4)$$

where p - is number of processors, $\mu_0 = r/V_0$, $\mu_{0s} = r/V_{0s}$, r - processor performance, V_0 - data exchange velocity, V_{0s} - channel program speed, L_a - number of channel programs calls. For computational physics codes $U(N) \sim N^3$, $V(N) \sim N^2$, $L_a(N) \sim N$ and, if $\mu \gg \mu_{0s}$

$$S(p) = \frac{p}{1 + \frac{c}{N} \mu_0 p^2} \quad (5)$$

The multiprocessor is quite effective if $S(p) \sim p$ or $N \gg c \mu_0 p^2$.

The Fig.2 gives the speedup $S(p)$ versus number of p processors for experimental measurements for Cm^* computer (cur-

ves 2,4,5) with SMAP, MEDUSA and STAROS operation systems⁴. This figure gives theoretical results (curves 1,3) which correspond to formula (4) with $\mu_0=0.26$, $N=150$ and formula (5) with the same μ_0, N and $c = p/8 + 2$. The last result corresponds to Hipercube computer. As you can notice statements (4),(5) have a good coincidence with practice. The optimal number of processors or number of processes in the parallel program can be obtained from (5):

$$P_{\text{opt}} = [(N/\mu_0)^{1/2}] \quad (6)$$

6. S A P A SYSTEM OF ANALYSIS OF PARALLEL ALGORITHMS AND OF PARALLEL PROGRAMS ADAPTATION FOR MULLIPROCESSORS

The main principles of parallel programming for multiprocessors described above are following:

- multitasking organization of computational process;
- large-module programming;
- optimization of the number and structures of independent computational processes. These roles lead us to SAPA system which consists from following parts:

a. The Fortran-routine, which consists from subroutines calls is compiled by the syntactic analyzer to the weight graph of present algorithm. Each node of the graph corresponds to a single subroutine call (or to the set of operators between the neighbor calls) and their weight equals to the run-time of the subroutine. This subsystem forms the adjacency matrix, with coefficients that are equal to the volume of module-to-module data exchange. Result of this com-

pilation for the simple algorithm is presented on Fig.3.

b. Second subsystem truncates the graph of algorithm to subgraphs with minimal connection. This section is done accordingly to the multiprocessor's structure. On Fig.4 the screen copy of SAPA realisation for graph on Fig.3 is presented. In the upper-left corner the connections between subgraphs are shown. In the other corner the task schedule for 4 processors is presented. On this figure there are run-time, speedup, efficiency of this section and parameters of single processes.

c. On the third step SAPA restores programs for single processors and constructs the connections between the processes.

These three parts of SAPA system may have their own rules for analysis of parallel algorithms and parallel program adaptation to various multiprocessors. The main purpose of this system building is the optimization problem solving for construction of optimal structure of multiprocessor system.

7.CONCLUSION

We have seen the specific of efficient organization of parallel programs for multiprocessors for simulations in computational physics. The programming system for modelling of complicated interaction of basic fortran-modules in the process of computational physics simulations for multiprocessors is proposed. SAPA system for parallel program construction is created, and the technology of parallel programs adaptation for multiprocessors is described.

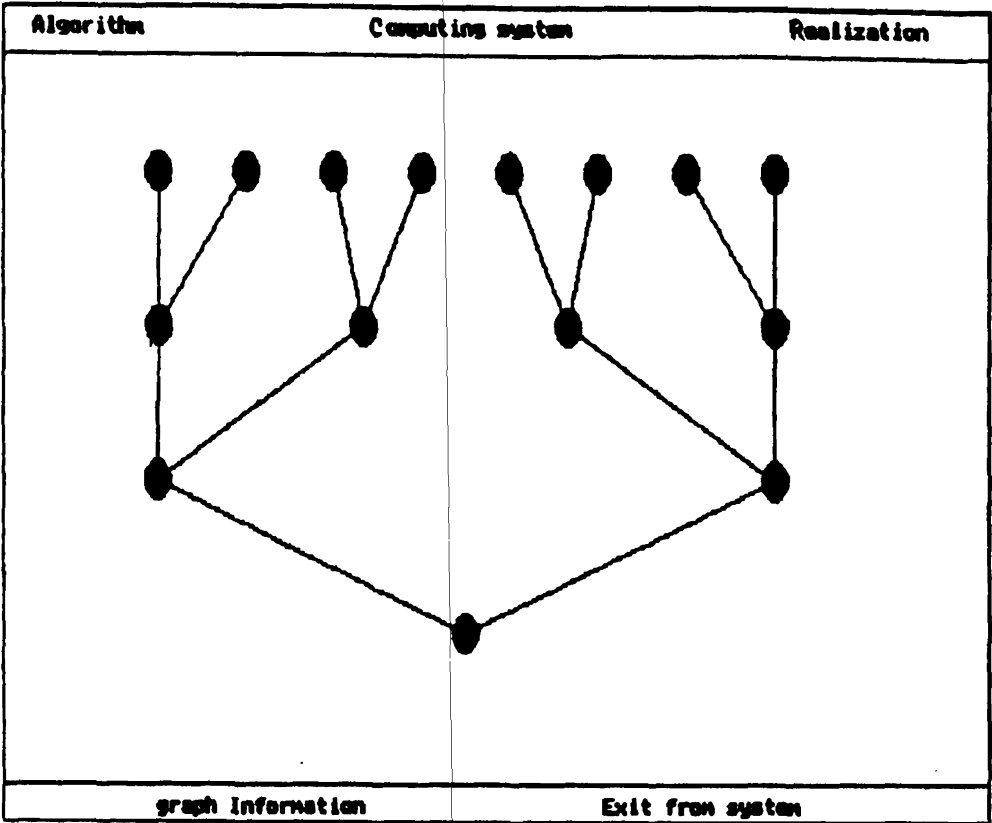


Fig 3.

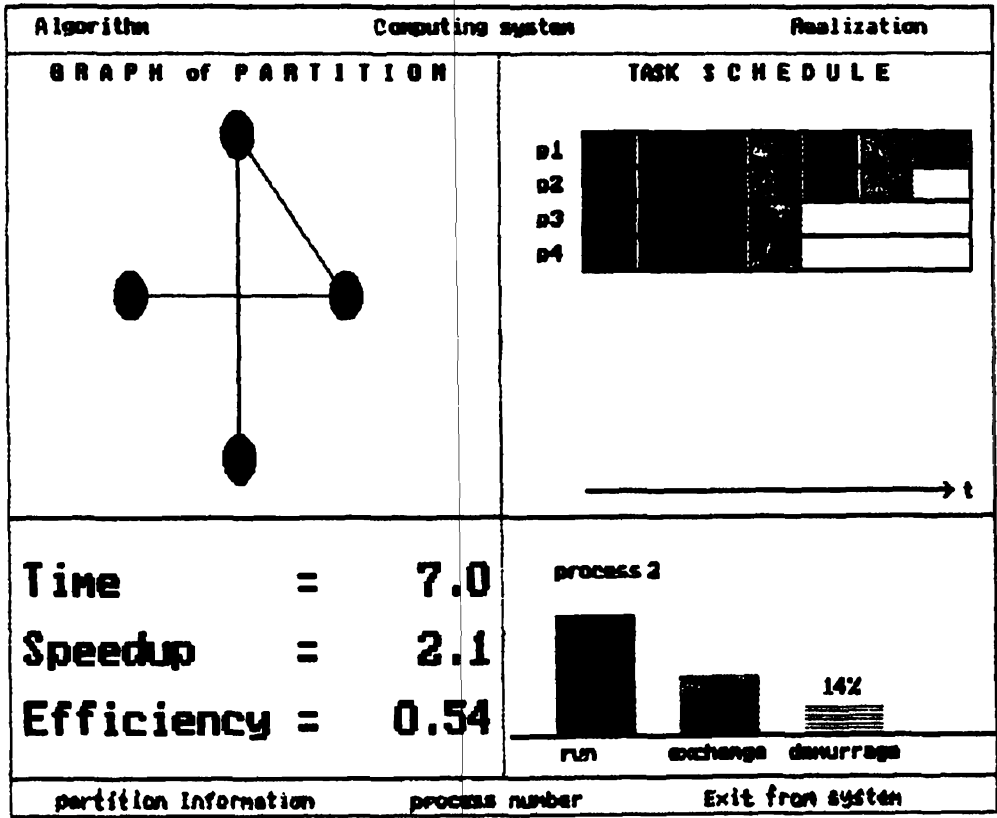


Fig 4.

8.REFERENCES

1. D.P.Dnestrovskij,D.P.Kostomarov, *Numerical simulation of plasma* (Springer-Verlag, 1986).
2. A.Y.Aydemir , W.C.Wiley, D.W.Ross ,*Phys. Fluids*,**B.1**(4), (1989),774-787.
3. E.V.Andreeva, Yu.N.Dnestrovskij, D.P.Kostomarov, V.V.Nefedov, A.V.Pedorenko, A.M.Popov *In: 17th EPS Conference on Controlled Fusion and Plasma Heating*,(Amsterdam, 1990), **Vol.14B**, Part 2, 894-897.
4. K.Hwang , F.A.Briggs *Computer architecture and parallel processing*,(1984) p.627

ALGORITHM BASED FAULT-TOLERANT TECHNIQUE FOR MATRIX INVERSION

M.K.Stojcev , E.I.Milovanovic,I.Z.Milovanovic

*Faculty of Electronic Engineering, Beogradska 14, P.O.Box 73,
18000 Nis, Yugoslavia*

ABSTRACT

In this paper the technique of algorithm-based fault-tolerance (ABFT) which is used to detect and correct transient faults by checksum matrices is considered for matrix inversion on triangular processor array. Schultz' method for inversion of triangular matrices is used. Using this method, for suitable defined initial matrix, an inverse matrix of triangular matrix is obtained in finite number of iteration steps. The processor array for inversion of triangular matrix of order n , consists of $n(n+1)/2+2n$ processing elements and redundant hardware required by the checksum technique.

1. INTRODUCTION

Today's technologies enable the fabrication of the whole parallel system (PS) on the indivisible silicon area, i.e. wafer. On the other hand the production of WSI PS, usually realized as a processor array (PA), is greatly affected by with production failures. Fault-tolerance (FT) is undoubtedly a desirable property of any PA which enables acceptable yields. The FT system is one that can continue to correctly perform its specified tasks in the presence of hardware failures and software errors¹. FT strategies can be designed to deal with three distinct types of failures which can occur during fabrication time compile-time and run-time. In current technology, a relatively large number of defects is expected when manufacturing a silicon wafer. The connection links have a low probability of failure compared with the processors. Run-time errors have a considerably lower probability of occurrence than production defects, but this faults cannot be ignored because the effects of their occurring can have much wider implications. Existing FT techniques which are applicable to this kind of errors can be divided into three categories: a)error masking, b)concurrent error detecting followed by reconfiguration, and c)algorithm-based FT.

First two techniques, require backup hardware for fault tolerance. The inclusion of hardware to achieve FT in the architecture of the PA require an increase in hardware complexity and cost also, while the inclusion of software to achieve FT decrease the cost and real-time operational performance. Both opposite requests suggest that in the final technical solution some design trade-offs have to be adopted.

Recently, a new fault-tolerance technique specially tailored for systolic algorithms and architectures has been proposed by Abraham *et al*^{2,3,4}.

ABFT technique has been proposed for matrix operations which include matrix-vector multiplication, matrix-matrix multiplication, LU decomposition, and matrix inversion⁴⁻⁷. In this paper a method for inversion of triangular matrix based on weighted-checksum approach is proposed. To compute an inverse matrix we use iterative Shultz' method, which in the case of triangular matrix gives the solution in a finite number of iteration steps. The architecture of the system for weighted-checksum matrix inversion is proposed also. It includes processor array (PA) and redundant hardware. Detection and correction of a single error in each row (column) of a matrix is considered. We give the diagram which represents time-redundancy ratio against matrix dimension n , and number of errors as a parameter. In all cases curves are monotonically decreasing.

2. WEIGHTED CHECKSUM APPROACH

In this section, the WCC₃ and the fault model is outlined briefly. The reader can refer to Huang et al³ for a detailed description.

2.1. Encoding scheme

Let $A = (a_{ij})_{n \times n}$ is a given nonsingular square matrix. One can define an $(n+2) \times n$ column checksum matrix A_c , as follows

$$A_c = \begin{bmatrix} A & \vec{e}^T A & \vec{e}_w^T A \end{bmatrix}^T \quad (1)$$

where $\vec{e}$ and $\vec{e}_w$ are $n \times 1$ vectors given by

$$\vec{e} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \end{bmatrix}^T, \text{ and } \vec{e}_w = \begin{bmatrix} 1 & 2 & 2^2 & \dots & 2^{n-1} \end{bmatrix}^T.$$

Similarly, row checksum matrix A_r can be defined as

$$A_r = \begin{bmatrix} A & A \vec{e} & A \vec{e}_w \end{bmatrix}. \quad (2)$$

The full $(n+2) \times (n+2)$ checksum matrix A_f is defined as

$$A_f = \begin{bmatrix} A & A \vec{e} & A \vec{e}_w \\ \vec{e}^T A & \vec{e}^T A \vec{e} & \vec{e}^T A \vec{e}_w \\ \vec{e}_w^T A & \vec{e}_w^T A \vec{e} & \vec{e}_w^T A \vec{e}_w \end{bmatrix}. \quad (3)$$

Five matrix operations which preserve the weighted checksum property exist: addition, multiplication, LU decomposition, transpose, and product of matrix with a scalar.

2.2. Error correction and detection

Every square matrix of order $n \times n$ can be represented as n row or n column

independent vectors. According to that it is of interest to describe how a single error can be detected and corrected in a n -dimensional vector⁴.

Let $\vec{a}^T = [a_1, \dots, a_n]$ is an arbitrary vector. We define two scalars, WCS1 and WCS2, as a direct product of two vectors in the following way:

$$WCS1 = \vec{a}^T \cdot \vec{e}^T = [a_1, \dots, a_n] [1 \dots 1] = a_1 + \dots + a_n$$

$$WCS2 = \vec{a}^T \cdot \vec{e}_w^T = [a_1 \dots a_n] [12 \dots 2^{n-1}] = a_1 + \dots + 2^{n-1} a_n \quad (4)$$

A vector $\vec{a}_c^T$ can be represented as

$$\vec{a}_c^T = [\vec{a}^T \quad WCS1 \quad WCS2] .$$

Let two separate processors in the processor array compute expressions WCS1 and WCS2. Since the matrix multiplication process has been finished, the main processor calculates the following expressions:

$$S1 = \sum_{i=1}^n a_i - WCS1 \quad \text{and} \quad S2 = \sum_{i=1}^n 2^{i-1} a_i - WCS2 .$$

Now, detection and correction of the single error is performed according to the following scheme:

- If $S1 = S2 = 0$, then error does not exist in $\vec{a}_c^T$.
- If $S1 \neq 0$ and $S2 = 0$, then WCS1 is erroneous.
- If $S1 = 0$ and $S2 \neq 0$, then WCS2 is erroneous.
- If $S1 \neq 0$ and $S2 \neq 0$, then $S2/S1 = 2^{j-1} \Rightarrow a_j$ is erroneous. The correct value for a_j can be obtained as $a_j = a_j - S1$.

3. MATRIX INVERSION

The fundamental direct method for determination an inverse matrix of regular square matrix $A_{n \times n}$, is based on Gaussian method of elimination⁸. Namely, the Gaussian method of elimination is applied on the extended matrix $(A|I)$ which is, after finite number of steps, transformed into matrix $(I|A^{-1})$.

From the iterative methods for matrix inversion we will point out to the Sultz' one, because it will be used in this paper. It is defined with the following theorem:

Theorem 1. Let for given matrix X_0 the following inequality

$$\| I - A \cdot X_0 \| \leq q < 1$$

is valid. Then a sequence $(X_k)_{k \in N_0}$, defined by

$$X_{k+1} = (2I - X_k A) X_k, \quad k = 0, 1, \dots$$

converges towards A^{-1} .

Shultz' and other similar methods were considered by Milovanovic⁸ and Schreiber⁹. Let us point out that Shultz' method is especially suitable when matrix A is a triangular one. In that case, for suitable defined initial matrix X_0 , an inverse matrix A^{-1} , is obtained in finite number of iteration steps⁹.

3.1. The proposed method

The concentration of this paper is on developing algorithm for weighted-checksum inversion of triangular matrices L and U . The inversion is based on Shultz' approach.

A procedure for weighted-checksum matrix inversion of regular square matrix $A_{n \times n}$ can be decomposed into three steps:

1. Weighted-checksum LU decomposition of matrix A ,
2. Weighted-checksum inversion of triangular matrices L and U ,
3. Weighted-checksum multiplication of triangular matrices.

PHASE 1: Fault tolerant arrays for matrix product and LU factorization using the triangular arrays were considered by Megson⁷. A methodology which maps a matrix multiplication algorithm to a fault tolerant array processor with different topologies and dimensions was considered by Liu¹⁰.

PHASE 2: In this phase a weighted-checksum matrix inversion of matrices L and U are performed. We assume that one of the matrices L or U is with unit diagonal (say matrix U). The procedures for calculating weighted-checksum inverse matrix of matrices L and U is as follows:

a) Computing U_c^{-1}

This subphase consists of the following steps:

$$\begin{array}{ll}
 \text{a.1. } X^{(0)} = \text{diag}(1, 1, \dots, 1)_{n \times n} & (X^{(0)} = I) \\
 \text{a.2. } R^{(0)} = I - X^{(0)} U & \\
 \left. \begin{array}{l}
 \text{a.3. } P^{(i)} = I + R^{(i)} \\
 \text{a.4. } X_c^{(i+1)} = P_c^{(i)} X^{(i)} \\
 \text{a.5. } R_c^{(i+1)} = R_c^{(i)} - R^{(i)}
 \end{array} \right\} & i = 0, 1, \dots, k
 \end{array}$$

After k iteration steps we obtain $X_c^{(k+1)} = U_c^{-1}$.

b) Computing L_r^{-1}

This subphase consists of the following steps

$$b.1. Y^{(0)} = \text{diag}\left(\frac{1}{l_{11}}, \dots, \frac{1}{l_{nn}}\right)_{n \times n}$$

$$b.2. R^{(0)} = I - Y^{(0)} L$$

$$b.3. P^{(i)} = I + R^{(i)}$$

$$b.4. Y_r^{(i+1)} = P^{(i)} Y_r^{(i)}$$

$$b.5. R_r^{(i+1)} = R^{(i)} R_r^{(i)}$$

$i=0,1,\dots,k$

After k iteration steps we obtain $Y_r^{(k+1)} = L_r^{-1}$.

In both cases, a) and b), the matrix $R^{(0)}$ is strictly triangular matrix of order $n \times n$, for which the following is valid:

$$i) (R^{(0)})^n = 0, \text{ and}$$

$$ii) R^{(k)} = (R^{(k-1)})^2 = (R^{(0)})^2.$$

So, in both cases, a) and b), the number of iteration steps k is the smallest integer that satisfies the inequality

$$2^k \geq n. \quad (5)$$

When inequality (5) is satisfied then $R^{(k)} = 0$, and according to that it follows that $X_c^{(k+1)} = X_c^{(k)}$. Since $P_c^{(k)} = P_c^{(k+1)} = I$, it follows that

$$X_c^{(k+2)} = P_c^{(k+1)} X_c^{(k+1)} = P_c^{(k)} X_c^{(k)} = X_c^{(k+1)}.$$

This means that iterative process a.4 (b.4) reaches fixed point for final number of iteration steps, k .

In the third phase weighted-checksum inverse matrix A_f^{-1} of matrix A is obtained according to the product $U_c^{-1} L_r^{-1}$.

4. HARDWARE REALIZATION

A proposed computer system consists of one host computer and processor array (PA) attached as coprocessor peripheral. The task of a host computer is to prepare data for PA, carry-out I/O, and to detect and correct transient and intermittent errors according to the checksums calculated in the PA. The processor array for inversion of triangular matrix of order $n \times n$ consists of

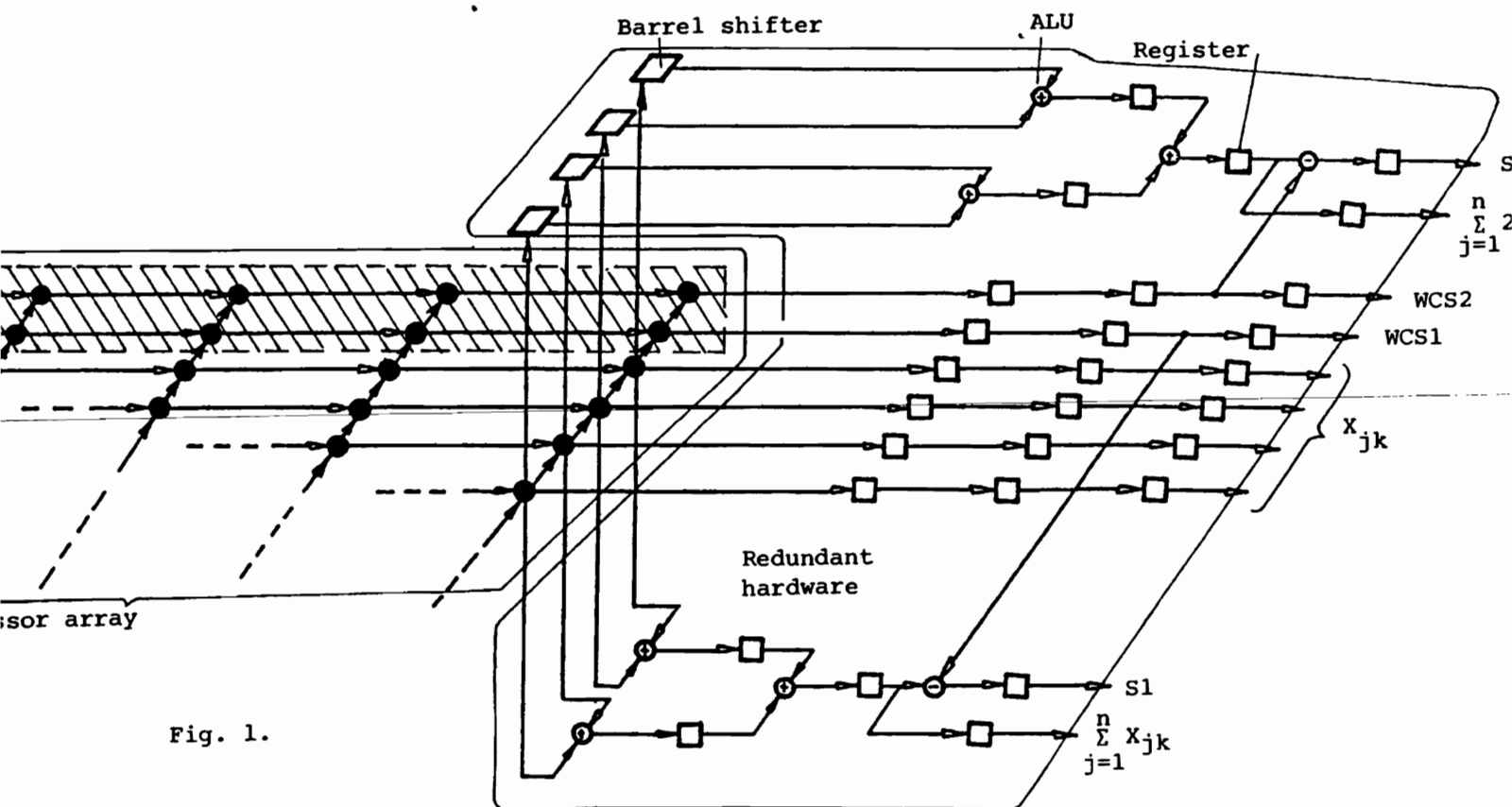


Fig. 1.

$n(n+1)/2+2n$ processing elements (PEs), and redundant hardware required by the checksum technique. Fig.1 shows the structure of the PA and redundant hardware in the case of inversion of triangular matrix of order 4×4 . Shaded PEs are intended for checksum-elements calculations. All PEs in the PA perform the following operation: $c \leftarrow c+ab$.

According to the described mathematical model error detection and correction is performed after every iteration step a.4 and a.5 during calculation of U_c^{-1} , and after steps b.4 and b.5 during calculation of L_r^{-1} .

The structure of the PA and input formats for calculating U_c^{-1} is shown in Fig.2.a and 2.b. Fig.2.a (2.b) corresponds to input formats for calculating matrix $X_c^{(i+1)}$ ($R_c^{(i+1)}$) according to the step a.4 (a.5) defined in mathematical model for calculating U_c^{-1} . Input data for calculating checksum elements prepares host computer.

Since matrix $X_c^{(i+1)}$ (i.e. $R_c^{(i+1)}$) has been calculated, error detection and correction process starts. For each column of $X_c^{(i+1)}$ (i.e. $R_c^{(i+1)}$) the redundant hardware computes in a pipeline fashion the following expressions:

$$\left. \begin{aligned} S1_k &= \sum_{j=1}^n X_{jk} - WCS1_k, \text{ and} \\ S2_k &= \sum_{j=1}^n 2^{j-1} X_{jk} - WCS2_k \end{aligned} \right\} k=1,2,\dots,n$$

Differences $S1_k$ and $S2_k$ ($k=1,2,\dots,n$) for all columns are obtained after $\log_2 n + n$ steps.

The host performs the following activities:

- i) Accepts calculated elements of matrix $X_c^{(i+1)}$ ($R_c^{(i+1)}$), differences $S1_k$ and $S2_k$ ($k=1,2,\dots,n$), and checksum elements $WCS1_k$ and $WCS2_k$ ($k=1,2,\dots,n$)
- ii) Perform detection and correction of a single error in each column of $X_c^{(i+1)}$ ($R_c^{(i+1)}$) according to the procedure described by Jou⁴.

The same structure of the PA is used for calculating L_r^{-1} .

5.PERFORMANCE EVALUATION

Now, we will make some comments about the timing measurements on the described algorithm for matrix inversion. We give actual runtimes for the proposed algorithm with and without error detecting and correcting procedure. Our analyze takes into account computation and communication time in PA and error detecting and correcting time in the host. The analyze gives measure of actual overhead due to implementation on *algorithm-based error detection* (ABED).

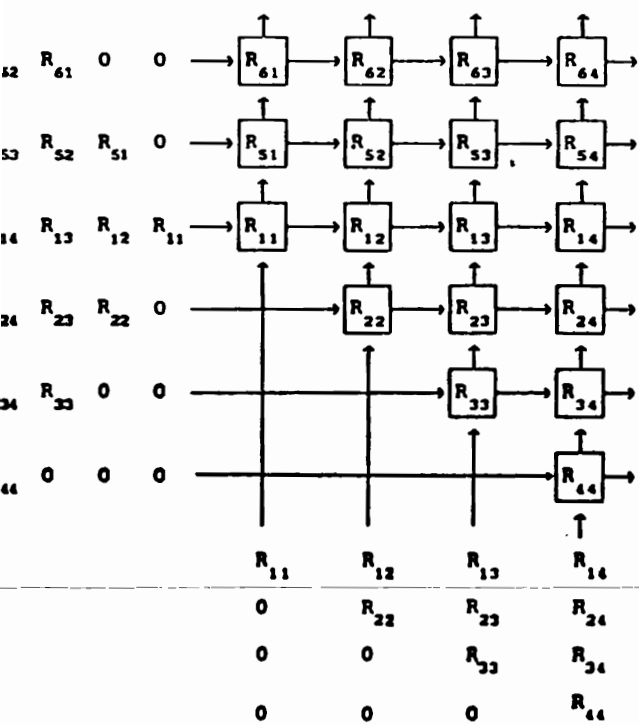


Fig. 2. b

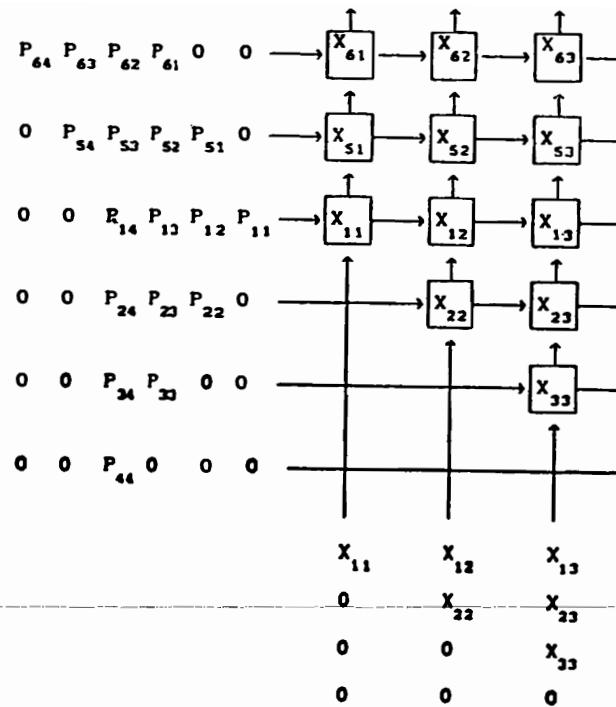
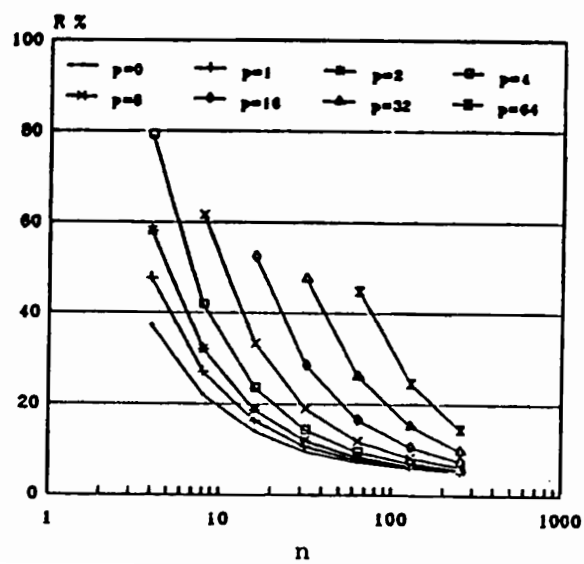


Fig. 2. a



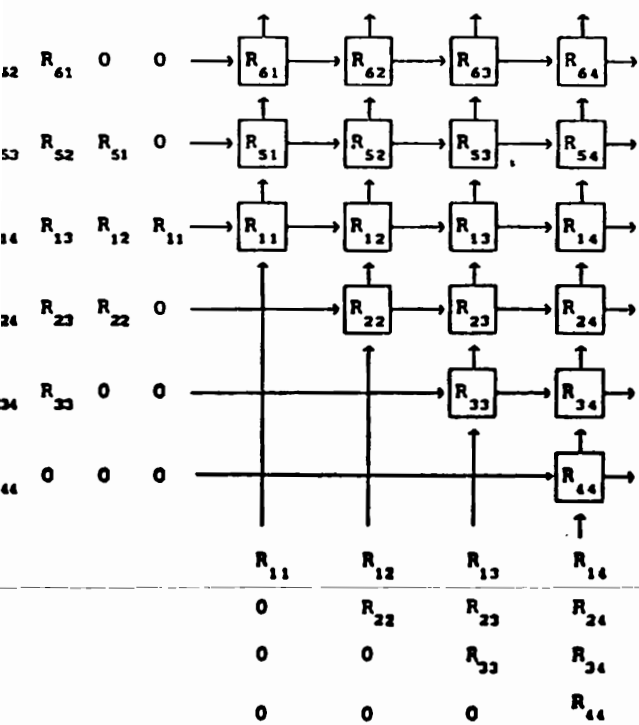


Fig. 2. b

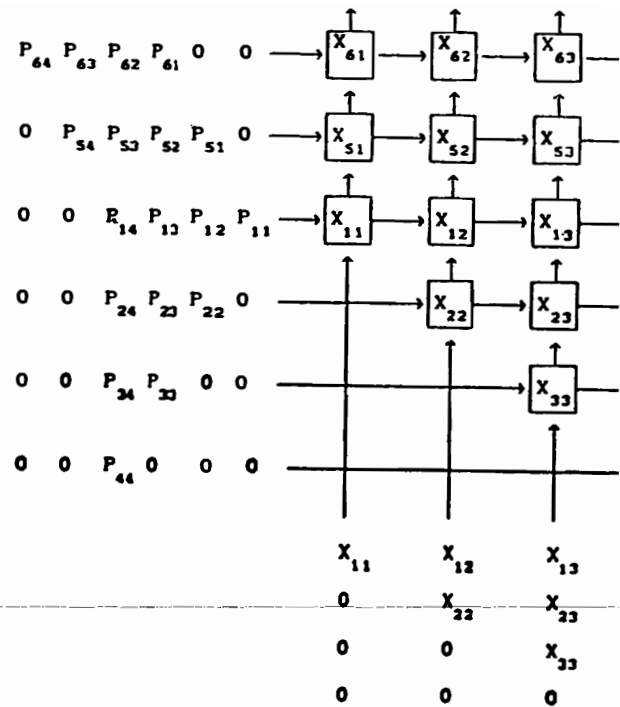
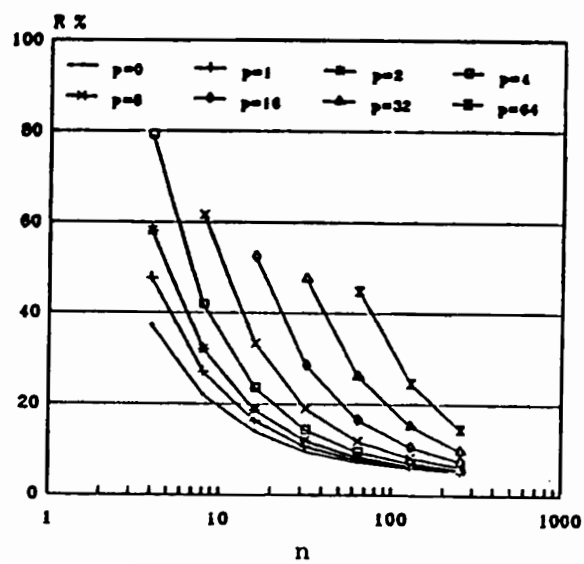


Fig. 2. a



5.1. Inversion without ABED

Multiplication of two triangular matrices is realised in $2n-1$ steps. Duration of one step is

$$T = T_{\text{com}} + T_{\text{comput}}$$

where

$T_{\text{com}} = 0.4T$ is communication time, and

$T_{\text{comput}} = 0.6T$ is computation time needed to perform the operation $c \leftarrow c + ab$.

The host accepts the results for

$$T_H = nT_{\text{com}} = 0.4nT.$$

Time needed for multiplication of two triangular matrices and transfer the result to the host is

$$T_{\text{mul}} = (2n-1)T + T_H = (2n-1)T + 0.4nT = (2.4n-1)T.$$

Since the inversion of triangular matrix by the described algorithm is performed for k iteration steps and each iteration step includes computation of two matrices, we have that inverse matrix of a given matrix is obtained after

$$T_{\text{inv}} = 2kT_{\text{mul}} = (4.8n-2)kT.$$

5.2. Inversion with ABED

When ABED is used detection and correction procedure is performed in each iteration step. Time needed for multiplication of two triangular matrices, detection and correction is

$$T_{\text{check}} = (2n+1)T + T_{\text{buff}} + nT_{\text{com}} + T_{\text{DC}},$$

where

$T_{\text{buff}} = T_{\text{com}} \log_2 n = 0.4T \log_2 n$ is time for buffering the results in redundant hardware,

T_{DC} includes time for testing the differences $S1_i$ and $S2_i$ ($i=1,2,\dots,n$) and time needed for error detection and correction, i.e.

$$T_{\text{DC}} = nT_{\text{test}} + pT_{\text{corr}} = 0.1nT + 0.9pT,$$

where p is a number of single errors.

According to this we have that

$$\begin{aligned} T_{\text{check}} &= (2n+1)T + 0.4T \log_2 n + 0.4nT + 0.1nT + 0.9pT = \\ &= (2.5n+1)T + 0.4T \log_2 n + 0.9pT. \end{aligned}$$

And, finally, the inverse matrix with ABED is obtained for time

$$T_{\Sigma CH} = 2kT_{\text{check}}.$$

Fig.3 depicts time-redundancy ratio, R , against matrix dimension n and a number of single errors p as a parameter. It can be seen that curves are monotonically decreasing in all cases. The greatest time-overhead ratio is obtained when the order of matrix is low (i.e. $n=4$) and the number of errors

equal to n ($p=n$). As the order of matrix increases the time overhead decreases.

6. CONCLUSION

A suitable method for inversion of triangular matrix based on weighted-checksum approach was considered. Detection and correction of a single error in each row (column) of matrix is considered. To calculate an inverse matrix we use iterative Shultz' method which in the case of triangular matrix gives the solution in a finite number of iteration steps. The architecture of the system for weighted-checksum matrix inversion includes processor array and redundant hardware. The diagram which represents time-redundancy ratio against matrix dimension, n , and number of errors as a parameter, is given. The obtained results indicate that the greatest time-overhead ratio is achieved when the order of matrix is low and number of errors equal to n . As the order of matrix increases the time overhead decreases.

7. REFERENCES

1. B.W.Johnson, *Design and analysis of fault-tolerant digital systems*, Addison-Wesley Publishing Company, Reading, 1989.
2. J.A.Abraham, P.Banerjee, C.-Y.chen, W.K.Fuchs, S.-Y.Kuo, A.L.N.Reddy, *Fault tolerance for systolic arrays*, IEEE Comput., Vol.20, No 7 (1987), 65-74.
3. K.H.Huang, J.A. Abraham, *Algorithm-based fault tolerance for matrix operations*, IEEE Trans. Comput., Vol. C-33 (1984), 518-528.
4. J.-Y. Jou, J.A.Abraham, *Fault-tolerant matrix arithmetic and signal processing on highly concurrent computing structures*, Proc. IEEE Vol. 74 (1986), 732-741.
5. F.T.Luk, H.Park, *An Analysis of Algorithm-based Fault Tolerance Techniques*, J.Parallel and Distributed Computing, 5 (1988), 172-184.
6. F.T.Luk, H.Park, *Fault-tolerant matrix triangularizations on systolic arrays*, IEEE Trans. Comput. Vol. C-37, 10 (1988), 1434-1438.
7. G.M.Megson, D.J.Evans, *Algorithmic fault tolerance for matrix operations on triangular arrays*, Parallel Comput. 10 (1989), 207-219.
8. G.V.Milovanovic, *Numerical Analysis I*, Naucna knjiga, Beograd, 1988. (In Serbian).
9. R.Schreiber, *Block Algorithms for Parallel Machines*, Technical Report, RPI Troy, New York 12180-3590, No 5 (1987), 1-16.
10. C.-M.Liu, C.-W.Jen, *Design of algorithm based fault tolerant VLSI array processor*, IEE Proc., Vol 136, Pt.E. No 6 (1989), 539-542.

SOFTWARE AND HARDWARE FOR SIMULATION OF PHYSICAL-MECHANICAL FIELDS

Rvachev V.L., Shevchenko A.N.,
Koshelenko A.I., Levin I.I., Fomin S.Yu.

Institute for Problems in Machinery
Ukr. SSR Academy of Sciences
2/10, Pozharsky Str., Kharkov-46 310046 USSR

ABSTRACT

The paper treats a brief description of a complex of software and hardware for efficient and operative solution of problems of the mathematical simulation. The intelligent tools (a system of POLE family) are proposed to investigate temperature, forced, deformed, electromagnetic and the other physical-mechanical fields, the mathematical models of which are the boundary-value problems for partial differential equations in the domains of the complex geometry. The ideas of direct methods and R-function theory, the structural facilities of which allow to take account of the boundary and geometric information at the analytical level are the basis of problem-oriented languages and specialized systems. Hardware of some algorithms yields the significant increase of efficiency of the computational complex and opens wide prospects for development of specialized processors and professional computers for engineering calculations of physical-mechanical fields.

1. Mathematical basis

When solving different scientific-technological and engineering problems the researchers often are faced with the necessity of interacting field calculation (problems of thermoelasticity, stability, etc.). E.G., when calculating high temperature systems (turbine blade, rocket engine nozzle, heat releasing elements of a nuclear reactor) there

arises a problem of high temperature stresses. The study of strength, reliability, static and dynamic stability of different thin wall structures in civil engineering, aviation, ship building also yields to complex problems of analysis of interacting forced and deformed fields.

From the mathematical point of view such problems are reduced to solution of a sequence of interconnected boundary-value problems for equations $A_j U_j = f_j$ in Ω_j , $j=1, \dots, l$ with the boundary conditions $L_{ij} U_j = \varphi_{ij}$ on $\partial\Omega_{ij}$, $i=1, \dots, m_j$. Here A_j and L_{ij} are some given differential operators, f_j and φ_{ij} are functions defined within the domain Ω_j and on boundary $\partial\Omega_{ij}$ respectively, U_j are the unknown functions describing physical processes. Analytical and geometric components of each boundary-value problem depend on the solution of the previous problems.

When studying and solving the above it is rather important to use efficient computational methods that allow to solve the problems formulated and to take account of complex geometric, logic and analytical information.

The problem of geometric information account is solved in a new manner by R-functions. For the given boundary-value problem we may construct the formulae of the type

$$U \equiv B(\Phi, \omega, \omega_i) + \varphi_0,$$

which exactly satisfy all (or a part) boundary conditions at any option of an indefinite component $\Phi = \{ \Phi_1, \dots, \Phi_m \}$. Here φ_0 is a known function; B is an operator, depending on domain shape Ω and sections $\partial\Omega_i$ of its boundary. These formulae called as solution structures define sets of functions satisfying on $\partial\Omega$ given system of conditions.

The paper /1/ describes a general approach to constructing solution structures of boundary-value problems based on application of R-functions and methods of analytical description of geometric objects of the complex configuration. Specially, the equation $\omega = 0$ of the boundary $\partial\Omega$ of the domain Ω and also the equations $\omega_i = 0$ of sections

$\partial\Omega_i$ of the boundary $\partial\Omega$ usually are used when constructing structures. The requirement of differentiability, normalization up to the necessary order is placed for functions ω and ω_i .

The logical operations over sets having a form of simple (support) objects for which there are finished inequalities $f \geq 0$ presented by the analytical geometry and analysis, are used to describe complex geometric objects. Algorithmically complete systems are constructed by R-functions. By using these systems we may give complex geometric objects as an equation $\omega = 0$.

The special mathematical tools - algebra of differential corteges are proposed for the efficient realization of the R-function method on the computer. Simple tools allow to present solution program of the boundary-value problem as the formulae and to apply the ordinary algebraic methods of their transformation.

Let be given a set of continuously-differentiable functions $f(x)=f(x_1, \dots, x_n) \in C^k$, defined everywhere in the space of their arguments.

Definition 1. The differential cortege of a function is $f(x)$ a set $d=d(f)$ which elements are values of a function and all its partial derivatives up to the m -th order inclusive at a certain point $a = (a_1, \dots, a_n)$. Here m is an order of the differential cortege, n is its dimension.

We treat some finite and infinite sets D of differential corteges that are defined simply by giving sets of generating functions, e.g.;

$$D^R = \{ d(f) \mid f(x) \in C^R \},$$

$$D^E = \{ d(f) \mid f(x) \in E \cap C^k \},$$

where:

C^R is a set of integer rational functions;

E is a set of elementary functions.

Definition 2. A system $U_n = \langle D; \pi \rangle$, consisting of the

basic set D of differential corteges and the set of partial operations defined on it,

$$\pi = \left\{ P_i^{n_i}(d_1, \dots, d_{n_i}) \mid i = 1, 2, \dots \right\} -$$

partial universal algebra of the differential corteges of the type $(n_i \mid i=1, 2, \dots)$ with operation signature $\pi = \{P_i^{n_i} \mid i=1, 2, \dots\}$.

The system $U_D = \langle D; \{+, *\} \rangle$, consisting of the basic set of differential corteges of continuously-differentiable functions on which binary operations $+$ (addition of corteges) and $*$ (multiplication of corteges) are defined and associative, commutative, distributive laws are performed for them and there exist neutral and inverse elements - a ring of differential corteges (algebra is not a field as a set D contains zero denominators, i.e. there exist such corteges $g \neq 0$ and $f \neq 0$ that the equality $g * f = 0$ is true).

The theorems are proved resolving the problem of completeness for algebras, their signatures include binary operations $+$, $*$, Λ_0 and a finite number of unary operations. Here Λ_0 is any continuously differentiable R-conjunction.

Product differentiation : let $u=u(x, y)$, $v=v(x, y)$, $f=f(x, y)$ be some continuously-differentiable functions and $f=u*v$. Then for derivatives of the product the Leibnits formula is true

$$\frac{\partial^n f}{\partial x^l \partial y^k} = \sum_{i=0}^l \sum_{j=0}^k C_l^i C_k^j \frac{\partial^{i+j} u}{\partial x^i \partial y^j} * \frac{\partial^{n-i-j} v}{\partial x^{l-i} \partial y^{k-j}}.$$

2. Intelligent tools

The theoretical results obtained are the basis of new technology of programming and its realization as the specialized systems of POLE family /2/ allowing to reduce

expenditures on composition and program debugging and computational experiments by solution of the boundary-value problems of the mathematical physics by dozens times. Application spheres of these systems are scientific studies, the engineering practice, education, where the whole spectrum of modern computers is used (from personal to supercomputers).

Unlike application packages based on methods of finite difference, finite and boundary element etc. the unique possibilities presented by R-function theory to construct approximate solutions of boundary-value problems for account of complex geometric configurations of domains are used in POLE system. Geometric and physical parameters in lettering may enter in the equations of boundaries of domain and their sections and also solution structures of boundary-value problems. This circumstance provides for additional functional possibilities of the system when solving optimization problems.

The structural facilities of the approximation theory, methods of R-function theory to construct solution structures, cortege algebra for the exact differentiation of superpositions of elementary functions and other facilities of the applied mathematics are included into database of POLE system besides the standard software. Database is the semantic net realizing a model of the domain under study and containing, on the one hand, algorithms and packages of programs and on the other hand, sets of different computing schemes, descriptions of models, methods, different physical and geometric characteristics. These data express the semantic content of stages of solution of the problems stated and are considered as the knowledge of the domain under study used to perform the necessary computations.

The POLE system traditionally includes components providing for man-computer dialogue (interpretation of dialogue schemes, control of a script and display windows

etc.) and also an operation with the database and packages of applied programs.

The basic structure of knowledge in POLE system has three obvious levels: applied, mathematical and programming. The informative and directive frames that are intended for construction of algorithms and programs at solving the boundary-value problem, are arranged at the lower level. The most active dialogue with a system occurs and the user is assumed to have rather deep knowledge of the domain under study (models, methods, algorithms and programs). At the middle level the dialogue interaction is provided for the mathematical statement of the problem. Many frames of the lower level are defined automatically. At the upper level the problem statement is realized in the terms of engineering concepts associated with computation of physical-mechanical fields. A mathematical model, a solution algorithm of the problem and a corresponding program are practically chosen without user's interference. In special cases POLE system may put a leading or particularized question concerning problem statement and output forms of the results obtained.

The POLE system automates the operation according to the new programming technology and solution of boundary-value problems of the mathematical physics and includes technological, mathematical, programming, informative, linguistic and methodical facilities.

Hardware of POLE system includes such computers as БЭСМ-6, ЕС ЭБМ, СМ ЭБМ and IBM PC with standard peripherals. Software of POLE system is defined by a set of the following methods: variational and projectional methods for solution of boundary-value problems for partial differential equations; methods and structural facilities of R-function theory to transform geometric information into the analytical one and to construct solution structure of boundary-value problems.

Software of POLE system is given by a wide set of

programs: a program package of differentiation of complex function superposition, a program package of function integration in the domain of complex configuration; a program package of realization of structural facilities of R-function theory; a program package of solution of problem on linear algebra; a program package for input-output of geometric information.

Information facilities of POLE system includes the specialized libraries characterized by a significant variety of the information stored: from the results of the computational experiments to the text of initial programs.

The linguistic facilities contain: the language-superstructure over the procedure-oriented language of programming (an extension of FORTRAN-77); the problem-oriented language RL to describe boundary-value problems of the mathematical physics; high-level languages to describe the problems with the application of terminology of a specific area of knowledge.

The methodical facilities include the following manuals: application description, program description, language description and the instruction for the programmer.

3. A specialized processor

The analysis of boundary-value problems of the mathematical physics and R-function theory methods allows to indicate the widely spread computational and logical procedures for hardware realization of POLE system elements and development of specialized processors increasing the speed response of the computational complex by 30-50 times when solving a problem of calculation and optimization of physical-mechanical fields. The structure of such specialized processors depends on set of basic operations (satisfying the requirements of functional completeness) the operations over the differential corteges, in the first turn, should be for the main arithmetic operations,

elementary functions and structural facilities of R-function theory.

The paper /3/ proposes the technique and structural solutions of constructing the high speed processor realizing the complex mathematical algorithms of the boundary-value problem solution at the hardware-microprogramming level. The basis for construction such processor is a conception of pipelining architecture with a dynamic reconfiguration of data flows allowing to carry out flexibly the orientation of technical facilities to the solvable problem.

The processor structure is oriented to the efficient operation of the type

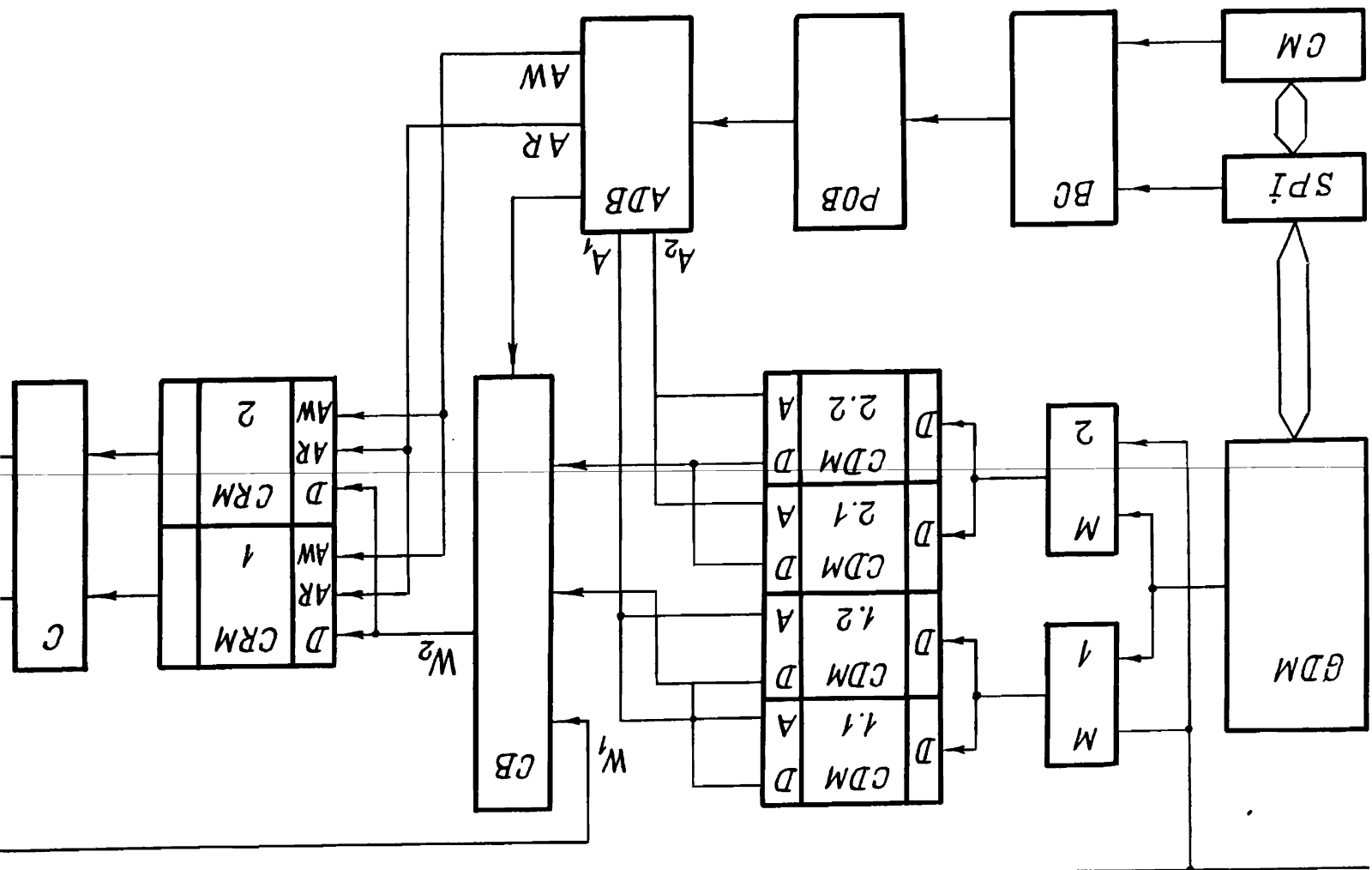
$$F(p) = \sum_{i=0}^l \sum_{j=0}^k C_l^i C_k^j U(p_1) V(p_2),$$

that enters into the nucleus of the majority of cortege operations. The other simpler operations entering into the nucleus of computations of different series, matrix and vector operations etc. may be efficiently performed.

To increase processor speed of response the pipelining of computations is used. The depth of the pipeline is chosen optimal to ensure the efficient operations both over the scalar values and differential corteges. The flexible, microprogram controlled configuration of internal dataway allows to fine-adjust the pipeline processor to the calculation algorithm realized.

Fig. 1 presents the structural scheme of the specialized processor that contain the following assemblies and blocks: general data memory (GDM), specialized processor interface (SPI), command memory (CM), block of control (BC), multiplexers (M1,M2), two two-page cache-data memory (CDM 1.1, CDM 1.2, CDM 2.1, CDM 2.2), pipelining organization block (POB), address driver block (ADV), computational block (CB), two-page two-port cache-result memory (CRM1, CRM2), output commutator (C).

The central processor (CP) controlling the



micro-computer via special processor interface (SRI) loads programming information into command memory and the digital one loads into data memory. Introduction of cache-data memory allows to load/unload programs and data on the background of cortege operation fulfilment in the processor. The availability of multiplexers M1, M2 ensures the possibility to record from a processing block in one of the pages of cache-data memory, avoiding information rewriting in the basic memory. So preparation time is reduced for pipeline computations for operations of addition and subtraction of corteges and for operations of multiplication and division of corteges, the data preparation is carried out on the background of computations.

The integrity of methods of R-function theory, its software in the specialized processors for the solutions of the problems of the mathematical physics is good facilities to conduct computational experiments for calculation and simulation of different physical-mechanical fields in the objects of the complex configuration.

References

1. Рвачев В.Л. *Теория R-функций и некоторые ее приложения*. -Киев: Наукова думка, 1982. -552 с.
2. Рвачев В.Л., Шевченко А.Н. *Проблемно-ориентированные языки и системы для инженерных расчетов*. -Киев: Техника, 1988. -198 с.
3. А.с. 1293737 СССР, МКИ³G06F 15/31. *Процессор для обработки массивов данных* /Рвачев В.Л., Шевченко А.Н., Галькевич А.А. и др. -Опубл. 28.02.87. Бюл. 8.

PERFORMANCE EVALUATION OF PARALLEL SIMD-SYSTEMS BY INTERPRETING AUTOMATICALLY TRANSFORMED ALGORITHMS ON SIMULATED ARCHITECTURES

MICHAEL M. GUTZMANN

*Computer Science Department IMMD VII
University of Erlangen-Nürnberg,
Martensstr. 3, D-8520 Erlangen, Germany*

ABSTRACT

A simulator for SIMD type architectures is presented. Starting from an architecture independent algorithm description based on recurrence equations, transformation steps for automatical parallelization, mapping and code generation are outlined. The final pseudo code program, together with architecture dependent parameters and execution time tables, are fed into the simulator in order to gain performance results for a set of SIMD architectures by interpreting the pseudo code instructions and evaluating performance characteristics.

1. Introduction

Performance evaluation of parallel SIMD systems is an important aim for industrial applications and a challenge for many research groups. Finding or developing a suitable supercomputer for a given massive parallel problem can be done experimentally or theoretically². Experimental research is very expensive and limited by the available machines⁷. Pure theoretical investigations are complex and difficult to handle. Although there are many contributions analysing the performance of SIMD machines, only a few systematic approaches are known^{5,6}.

The difficulty in comparing a set of different SIMD architectures is caused by the large variety of algorithms, mappings and compilations for one problem on the observed architectures¹². The efficiency of programming SIMD machines basically depends on the mapping. Different architectures and problem sizes cause different mappings⁸. As a result of that, a special program has to be developed for each machine and each problem size.

The approach presented here (see figure 1) is based on a definition of performance evaluation with exactly characterized measures and limits concerning modifications and varieties of algorithms as well as machine dependent optimisations: An algorithm is specified in an abstract specification language and machine dependent modifications are not permitted. First transformations of the observed algorithm create a machine independent parallelized specification for a fixed problem size. Mappings of the transformed algorithm are strictly rule based, hence they are of the same quality for

each target machine; machine oriented optimisations are not allowed. These restrictions avoid intuition and guarantee comparability of different architectures, especially with regard to the amount, power and interconnection pattern of the processing elements.

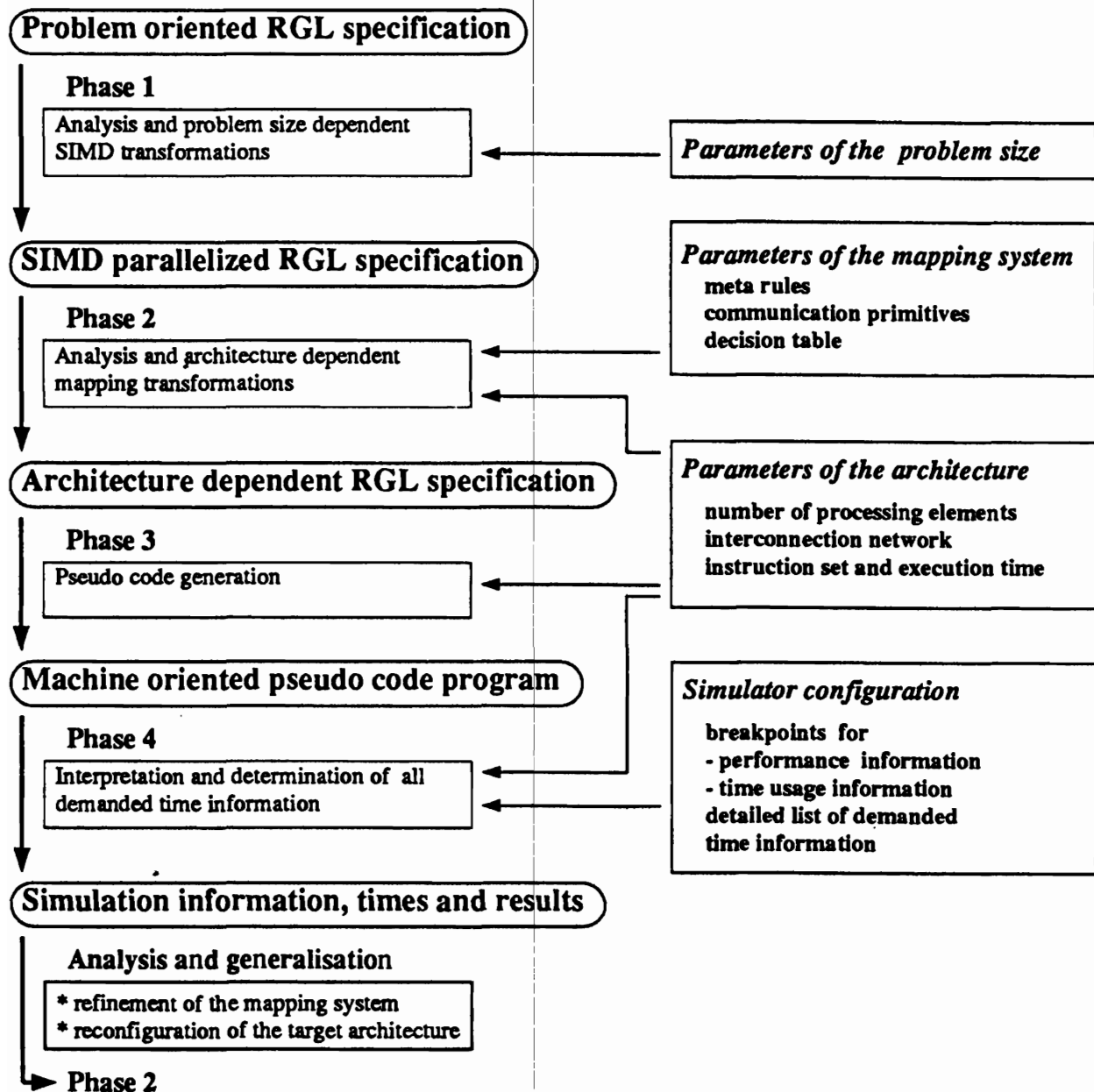


Figure 1: The basic concept

The goal is to find a well suited SIMD machine for a given set of algorithms. We simulate different SIMD architectures which are defined on a high level to avoid overhead and to concentrate on the important features. Programs to be interpreted on these machines are automatically derived from abstract specifications, with special regard to the definition of the performance evaluation.

The simulator gains information about performance, processor and network utilisation as well as detailed measurements about partial and total time usage of each instruction. By analysing the different case studies, the results given by the simulator can be generalized and formalized to refine the mapping rule system. In addition to that, the target architectures can stepwise be modified to eliminate bottlenecks and to gain a better performance of a given algorithm.

A specification language RGL¹¹, based on recurrence equations, is used as carrier language for the system: The specification of the algorithm and the transformations are based on RGL. The final machine oriented representation of the algorithm in RGL is directly translated into a pseudo code program which is interpreted by the simulator.

Basis for the mapping rule system and for the simulator is the description of the target SIMD architecture. The description allows for the definition of different SIMD target machines not yet built or available. The instruction set represents the pseudo code; with each instruction, its execution time is associated. The execution time table offers sufficient possibilities to characterize the processing elements. In addition to that, the time table efficiently hides details and avoids overhead. The direct correlation between the mapping rule system and the simulator, both based on the target machine description, allows for the comparison of different SIMD architectures with regard to a given algorithm automatically.

2. The specification language RGL

In SIMD computers a number of processing elements operate on a large set of data items synchronously. The operations possible on this set can be divided into computational and communicational operations. During a computation step, the same operation is performed in parallel on each component of the data structure (except for deactivated components). During a communication step, the items of the data structure are interchanged according to a communication function which reflects the topology of the interconnection network. This is called the SIMD operation principle (see figure 2).

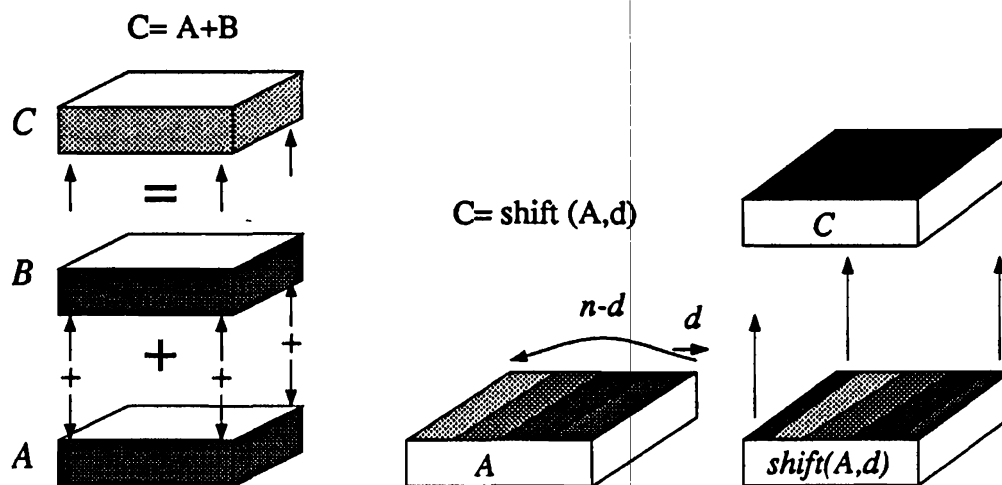


Figure 2: The SIMD principle

Using e.g. a mesh connected SIMD architecture, the set of data items could be the elements of a matrix, the computational operation a componentwise addition of two matrices and the communication operation a horizontal circular shift of all matrix elements.

To make use of the SIMD operation principle on a given machine, the programmer has to use an extended and thereby machine dependent programming language. The major disadvantages of this approach are:

- the lack of scalability: change of problem size or number of processing elements requires the recoding of an algorithm for an efficient implementation
- the lack of portability: since different manufacturers provide their machines with different language extensions even if the same SIMD type architecture is used.

To overcome these problems, a specification language RGL (from German 'Rekurrenz GLeichungen') was developed at our institute¹⁰. A compiler for the generation of programs for different mesh connected SIMD computers out of RGL specifications is being developed. In this paper, RGL will be used as carrier for the generation of pseudo code for a set of SIMD type computers. The pseudo code is then fed into an SIMD architecture simulator.

An RGL specification is an abstract description of the mathematical relation between input and output parameters in terms of recurrence equations. Recurrence equations are widely used and well suited for the description of (in particular parallel numerical) algorithms. Due to the single assignment principle for the identifiers (not variables) of the equations, artificial dependencies - which would suppress parallelization - are avoided. The resulting algorithm specification is independent of an underlying architecture or problem size.

An RGL specification does not contain any operational description on how (in which order) to compute results and intermediate values from the input parameters. This offers the chance to rearrange and transform the equations in order to meet the requirements of the SIMD operation principle in general and of a given SIMD architecture in particular. RGL allows for carrying out such transformations automatically. For a detailed description of RGL see the publication¹¹.

The keywords **algorithm**, **equations**, **input** and **compute** indicate special sections of the specification (see figure 3). The header of a specification consists of the keyword **algorithm** followed by an identifier giving the name of the specification. By this name, the algorithm specified can be addressed as a sub-algorithm, whose results can be imported via an **obtain** phrase. The **equations** section consists of simple recurrence equations and equation templates, i.e. equations that are to be applied to a given range of identifiers. In place of equations the **obtain** phrase can be used, which allows to import results from other algorithms without loss of potential parallelization possibilities.

Other operators, like e.g. **assoc**, used to declare a binary operator to be associative, or **first**, used to find the first data item in a given range, that fulfills a given predicate, or **record**, used to structure complex data fields, allow for abstract machine independent programming and offer possibilities for automatically machine dependent transformations.

```

algorithm gauss_jordan
equations
  m(0,i,j) = a(i,j), j=1,...,N, i=1,...,N;
  r(0,i)   = b(i),   i=1,...,N;
          /* The algorithm requires N steps, in the */
          /* t-th step the t-th row is normalized .. */
  m(t,t,j) = m(t-1,t,j) / m(t-1,t,t), j=1,...,N, t=1,...,N;
  r(t,t)   = r(t-1,t) / m(t-1,t,t),   t=1,...,N;
          /* .. then the other rows are transformed */
  m(t,i,j) = m(t-1,i,j) - m(t-1,i,t) * m(t,t,j),
          j=1,...,N, i=1,...,t-1,t+1,...,N, t=1,...,N;
  r(t,i)   = r(t-1,i) - m(t-1,i,t) * r(t,t),
          i=1,...,t-1,t+1,...,N, t=1,...,N;
  x(i)     = r(N,i), i = 1,...,N;
input
  N; /* number of equations */
  a(i,j), j=1,...,N, i=1,...,N; /* matrix coefficients */
  b(i),   i=1,...,N; /* right hand side */
compute
  x(i),   i=1,...,N.

```

Figure 3: A gauss-jordan specification

The main properties of RGL are as follows:

- *description of algorithms by recurrence equations according to mathematical habits*
- *algorithm specification independent of architecture and problem size*
- *no side effects due to single assignment principle*
- *formal syntax allows automatic transformations*

3. Transformations

The automatic generation of programs from RGL equations is performed in two steps.

- ☐ Architecture independent parallelization transformations, generating an RGL SIMD normal form
- ☐ Architecture dependent mapping transformations, generating a machine oriented RGL specification

Phase 1 transformations meet the requirements of the SIMD operation principle, since the equations are divided into calculation and communication. Before the analysis of completeness and uniqueness, especially necessary for parallelizing transformations, several elementary steps are performed⁹:

- textual substitution of explicite functions
- insertion of problem size parameters in specification constants
- breaking up the structuring operator record and converting the first construct
- elimination of obtain calls and recursions, transformed into iterations

After the dependence analysis and the validation of all correctness demands⁴, parallelizing transformations separate calculation and communication equations and generate the SIMD normal form. Two examples for compression¹ and adaptation of index expressions will illustrate basic schemes (see figure 4 and 5).

□ Compression

$$a(2*i,j)=b(2*i,j)-c(2*i,j), i=1,...,5, j=2,...,6;$$

$$\begin{aligned} a(2*i,j) &= a'(i,j), i,j \in D; \{\text{communication}\} \\ a'(i,j) &= b'(i,j) - c'(i,j), i,j \in D; \{\text{SIMD form}\} \\ b'(i,j) &= b(2*i,j), i,j \in D; \{\text{communication}\} \\ c'(i,j) &= c(2*i,j), i,j \in D; \{\text{communication}\} \end{aligned}$$

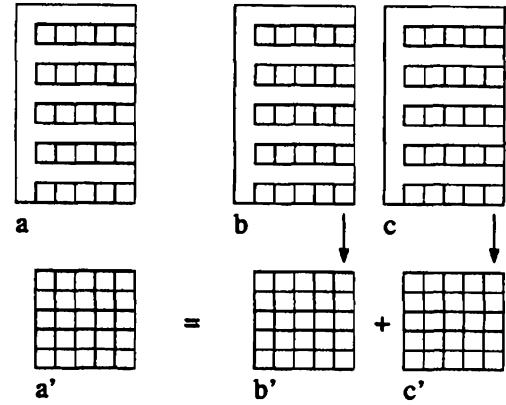


Figure 4: Transformations for compression

□ Adaptation of index expressions

$$a(2*i,j)=b(i,j)-c(i+j), i=1,...,3, j=1,2,3;$$

$$\begin{aligned} a(2*i,j) &= b'(2*i,j) - c'(2*i,j), i,j \in D; \{\text{SIMD form}\} \\ b'(2*i,j) &= b(i,j), i,j \in D; \{\text{communication}\} \\ c'(2*i,j) &= c(i+j), i,j \in D; \{\text{communication}\} \end{aligned}$$

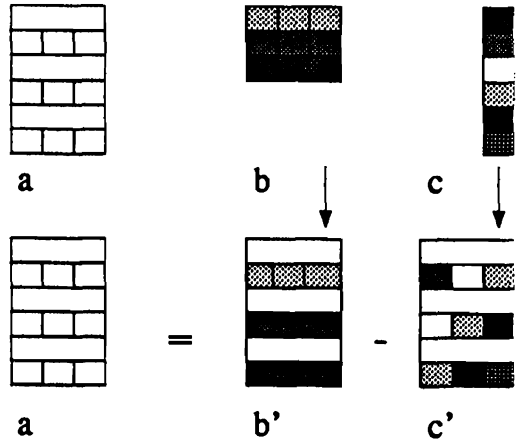


Figure 5: Transformations for adaptation of index expressions

Phase 2 transformations realize an architecture dependent mapping from the RGL SIMD normal form and communication equations, involving the target architecture parameters. Two main steps are performed:

1. Communication and computation requirements are analysed separately. The architecture configuration file is used to gain concrete execution times for the determined demands. Standard communication patterns and complex data compression functions are provided by RGL libraries.

2. Several mapping strategies (e.g. slicing or crinkling for mesh connected architectures) are investigated. Different mapping methods for special parallelization algorithms (e.g. divide and conquer for associative operations, "assoc" in RGL) are observed and for each case the 'mapping costs' are determined. Mapping meta rules, not associated with special target architectures, evaluate the different alternatives and select one appropriate mapping.

An important request in all phase 2 transformations is the observance of the non-architecture optimizing mapping rules. The mapping rules are not regarded to find best mappings on each architecture, hence they guarantee evaluation and comparison of different architecture models. At the end of these transformations, the mapped equations can directly be used for pseudo code generation.

4. Simulation

We simulate different SIMD target machines to evaluate their performance according to elementary and numerical algorithms. The simulator offers the possibility to investigate SIMD architectures, complex and expensive in hardware, on a high level. The simulation is realized on system level, directly interpreting pseudo code instructions, using HLL libraries. Compared with simulation on register transfer or gate level, this avoids overhead and efficiently gains all demanded results in acceptable time¹⁰.

Two different configuration files, used by the simulator, offer additional facilities:

- **Architecture Configuration File (ACF):** The description of the target architecture, especially the detailed instruction set execution time table, offers the possibility to observe, influence and analyse the performance data, determined by the simulator.
- **Simulator Configuration File (SCF):** The simulator receives the demands of all performance and utilisation data to be determined. The SCF allows for gaining and comparing architecture specific results on different detailed abstraction levels.

The significance of simulation results directly depends on the quality of the instruction set execution time table. An indirect influence is given by the transformation system: Architecture independent SIMD parallelizing transformation (phase 1) and architecture dependent target machine mapping transformations (phase 2). Phase 1 transformations do not influence simulation results, since these transformations are architecture independent. Phase 2 transformations are different for several target architectures, but the mapping and transformation meta rules are identical: The system does not allow architecture dependent optimisations. The machine invariant mapping rule system guarantees comparability of different models, moreover the simulation results of utilization can be used to refine the mapping rule system. The advantages of the simulator in summary:

- *efficient code interpretation, using HLL libraries*
- *reducing simulation time, hiding details and avoiding overhead*
- *one simulator for different architectures, configured by the ACF*
- *different abstraction levels for performance determination, configured by the SCF*

The SIMD architecture model for the machines to be simulated is based on the Master Control Unit (MCU), the Processing Elements (PEs) and the Interconnection Network (IN) of the PEs (see figure 6).

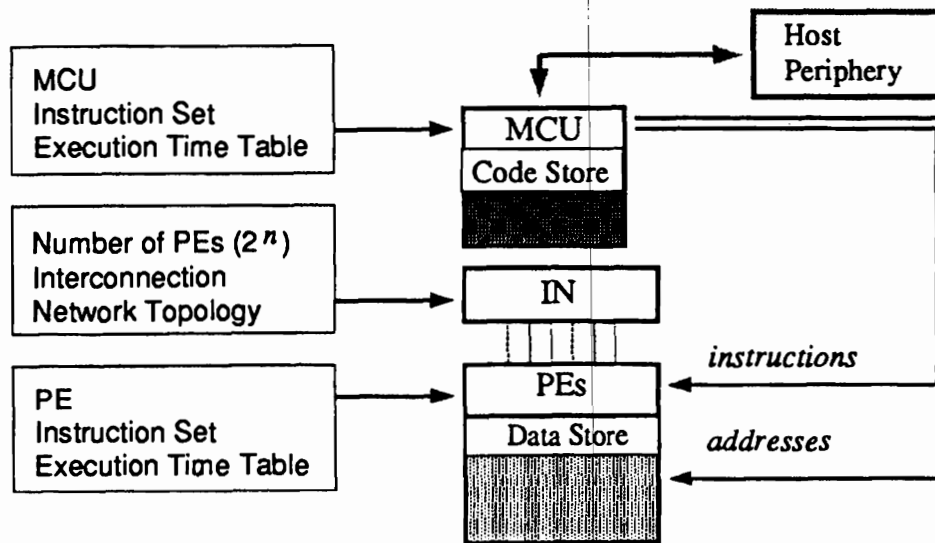


Figure 6: SIMD model

The features of the main components more detailed:

☐ **MCU**

- fetches pseudo code instructions from the MCU code store
- decodes and interprets the instructions, executes scalar and control instructions
- broadcasts parallel operations to the PEs, generates addresses
- manages all I/O functions (host computer, peripherie)
- **Configuration:** MCU instruction set and associated execution time

☐ **PEs**

- execute elementary operations, compute and communicate synchronously
- can be masked: the result of an operation is ignored
- **Configuration:** PE instruction set and associated execution time table

☐ **Interconnection Network of the PEs**

- connects the PEs due to a given topology
- **Configuration:** Number of PEs (2^n) and topology (2-D grid, 3-D grid, ...)

The pseudo code defined for the simulator is a machine dependent high level assembly language. Serial commands are defined for the MCU, parallel commands for the PEs. The first version of the simulator is restricted to two data types: integer and float used for variables, constants and program labels. We separate computing instructions

network dependent communication instructions and complex functions which are using both, computation and communication instructions (see table 1).

Table 1: Pseudo code instruction classes

computing insructions	communication instructions - network-dependent					complex functions
	bus	2-D grid	3-D grid	cube	shuffle- exchange	
arithmetical	bus(1)	shift(N)	shift(N)	cube(1)	shuffle	data compression
logical	bus(2)	shift(S)	shift(S)	cube(2)	unshuffle	data expansion
bitmanipulation	...	shift(E)	shift(E)	cube(4)	exchange	
trigonometrical	bus(n)	shift(W)	shift(W)	...		
transcendental		...	shift(U)	cube(n)		
move & store		shift(NW)	shift(D)			
rounding		shift(NE)	...			
type converting			shift(UN)			
comparison			shift(US)			
(cond.) jumps			...			
extended			shift(UNE)			
other functions			shift(UNW)			

Computing instructions are classified in twelve categories. Communication instructions are limited to five different network types next to. Complex functions are based on special algorithms, depending on the network and architecture size, and are not realized for the time being.

The architecture configuration file (ACF) is used by the mapping system and the simulator. The architecture description in the ACF includes all information for configuration defined in the SIMD architecture model. The pseudo code is the same for all architecture models. The performance of the PEs is defined by the execution time table.

For every model, all pseudo code instructions are defined, different execution times are associated with the instructions for different models. Floating point operations for an architecture, naturally not supported by hardware, e.g. could be described as shown below (see table 2).

The instruction set execution time table has to be calculated separately for every machine. For the mapping system and for the simulator it is nescessary, that all instructions are defined.

Table 2: Execution time definition

instruction	execution time (elementary cycles) - fine grain definition -	max time	min time (pipelined)
add	96	$+(add)/12$	$-(add)/20$
mult	800	$+(add)$	-10
fadd	$9*(add)+5$	$+(add)+2$	$-(add)$
fmult	$2*(mult)+4*(add)+12$	$+(add)*2$	-0

The execution time table for computation instructions can be added directly. Communication instructions are more difficult to handle: In many cases it is useful to emulate a communication instruction from one topology on another, eg. a 'shift' from a mesh interconnection network onto a hypercube topology (see figure 7).

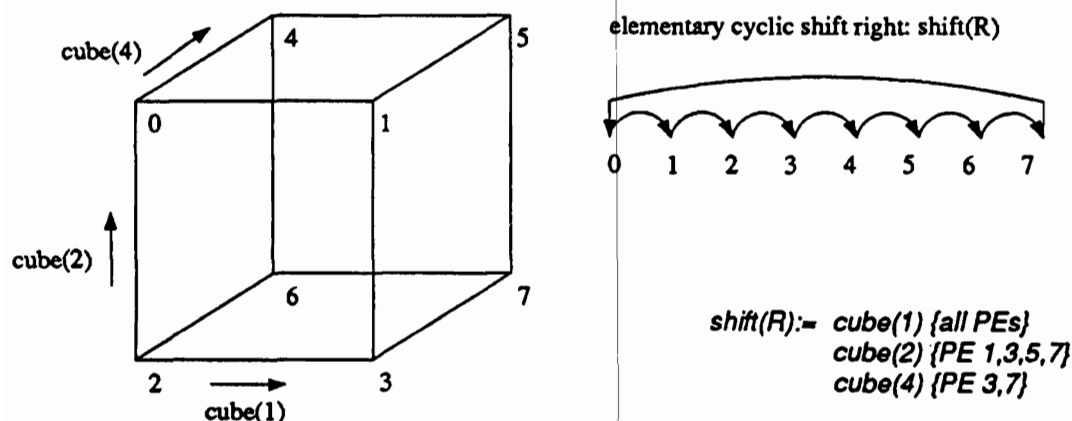


Figure 7: Shift on the hypercube

To overcome this, one approach is to stepwise specify communication tasks from one topology for other topologies in RGL libraries. Hence, the execution time table for mutual communication demands on different topologies can be gained automatically from the elementary instructions. The realisation and the execution time of powerful functions, e.g. the data evaluation function sum, directly associated with the associative RGL construct *assoc*, depend on the problem size, the topology and on the instruction set of the target architecture: Implementations may differ, although the algorithmic strategy is well known as 'divide and conquer'. So the RGL libraries include a whole class of specifications for one function. To gain the execution time for the associated machine dependent pseudo code instructions, the RGL libraries have to be analysed. The advantages of the execution time tabel concept are:

- *the simulator interprets the same pseudo code for each architecture model*
- *utilisation and performance data can be determined from the execution time table*

- *the power of each PE can be defined and modified with very fine granularity*
- *the execution time is based on elementary instructions, most complex functions can be added automatically*

The simulator configuration file (SCF) is used to define all performance and utilisation data which are determined by the simulator (see table 3).

Table 3: The simulator configuration file

Utilisation of:		MCU	MEM	Communication					Processor Computation											
				I1	I2	I3	I4	I5	C1, C2, C3, C4, C5, C6, C7, .. C12											
cycles in total	P0																			
cycles in special parts	P1		5 Interconnection Network classes (I1 .. I5) and 12 Classes of computation commands (C1 .. C12) are predefined. Focussing to any class, different instructions can be specified for detailed simulation information. A pseudo code program is regarded as a set of divisions (P1 .. Pn), the user is animated to separate for more detailed information During simulation, any time a defined breakpoint (B1 .. Bn) is found, all cycles up to there are added to the breakpoint list.																	
	P2																			
	...																			
	Pn																			
cycles at breakpoints	B1																			
	B2																			
	...																			
	Bn																			

The SCF contains information for the simulator and is used as a filter for simulation data:

- *the programmer defines all points of interest in the table, representing main properties of the SCF*
- *the simulator determines cycle times and adds them to the table*
- *information not demanded is filtered*
- *different levels of abstraction, according to the table, are possible in one simulation*

5. Conclusion

RGL was originally developed as a specification language for mesh connected SIMD architectures. Different algorithms for many, especially numerical problems, have been specified in RGL. It was shown that RGL is well suited for automatical derivation of parallel SIMD programs.

Besides mesh connected computers e.g. the AMT DAP, other network topologies, like e.g. hypercubes or shuffle exchange networks were investigated.

The definition of a unique pseudo code for different target SIMD models was the basis for the simulation.

Machine independent parallelizing transformations and machine dependent mapping transformations, both based on mathematical, stepwise refined rule systems, are used to generate different target machine oriented pseudo code programs automatically from RGL specifications.

The performance and utilisation results, determined by the simulator during interpretation of pseudo code programs, will be analysed to evaluate and compare the performance of different SIMD architecture models. Different restrictions and definitions, concerning the automatical derivation of programs, avoid intuition, limit the variety of possible resulting target programs and permit the comparison of different SIMD models.

Bibliography

1. P. Flanders, *Parallel Algorithms for Basic Data Compression and Expansion on the ICL DAP*, Technical report cm85, ICL Systems Strategy Centre 1983
2. M. Gutzmann, *Algorithmenbasierte Untersuchung der Einführung mehrerer Befehlsströme am DAP*, Master's thesis, Universität Erlangen-Nürnberg IMMD VII 1988
3. R. Kern, *Automatische Generierung von Programmen aus RGL-Spezifikationen in SIMD-Normalform und maschinenunabhängige Interpretation des erzeugten Pseudocodes*, Master's thesis, Universität Erlangen-Nürnberg IMMD VII 1991
4. J. L. Madloch, *Syntaktische und Semantische Analyse von RGL Spezifikationen und Überführung in einen Abhängigkeitsgraphen*, Master's thesis, Universität Erlangen-Nürnberg IMMD VII 1991
5. D. Müller-Wichards, *An Algebraic Approach of Performance Analysis*, Parallel Computing in Science and Engineering, 4th Int. DFVLR Seminar on Foundations of Engineering Sciences, Berlin 1987, R. Dierstein et al., Eds., Springer-Verlag
6. D. Parkinson, *Performance Analysis in a 4096 Processor Environment*, Journal of Systems and Software 1 (11–15) 1986
7. L. Siegel and H. Siegel and P. Swain, *Performance Measures for Evaluating Algorithms for SIMD Machines*, IEEE Transactions on Software Engineering 8 (319–330), 4 1982
8. A. Strey, *Ein Vergleich von Verbindungsnetzwerken für große SIMD-Parallelrechner*, PhD thesis, Universität Erlangen-Nürnberg IMMD VII 1991
9. J. Stroczyk, *Automatische Transformation von RGL-Spezifikationen mit affinen Indexausdrücken in eine SIMD-Normalform*, Master's thesis, Universität Erlangen-Nürnberg IMMD VII 1991
10. K. Thalhofer, *Automatische Generierung von Programmen für SIMD-Parallelrechner aus Spezifikationen*, PhD thesis, Universität Erlangen-Nürnberg IMMD VII 1989
11. K. Thalhofer, *RGL—A Specification Language based on Recurrence Equations*, Tech. Rep. 8/89, Universität Erlangen-Nürnberg IMMD VII 1989
12. R. Upton and S. Tripathi, *On the Performance Evaluation of Fine-Grained SIMD Computer Architectures: An Analysis of the Connection Machine*, High Performance Computer Systems (129–139), E. Gelenbe, Ed., North-Holland, Amsterdam 1988

THE COMBINED APPROACH TO THE DISTRIBUTED COMPUTER SYSTEM SIMULATION

R.L.Smeliansky, Y.P.Kazakov, Y.V.Bakalov.

*Computer System Lab.,
Computer Mathematic and Cybernetic Dep.,
Moscow State University,
Leninsky Hill, Moscow, 119899, USSR.*

ABSTRACT

This paper presents the main principles of the combined approach to the distributed computer system simulation and its language basis. This approach is devoted to combine the different kinds of distributed computer system analysis on the base of common modeling scheme.

1.Introduction

Neither a single algorithmic analysis of the distributed systems nor a single quantitative analysis is complete because of some particularities of their behavior. We define these kinds of analysis through the differences of the methods that they use. The quantitative analysis uses the method of queueing theory, of the operational research theories etc. The main feature of these methods is that time is used as a metric value there. The algorithmic analysis uses the methods of automaton theory, the algorithmic theory (like Petry Net, Turing Machine, Markovian Algorithms etc.). The common feature of the algorithmic analysis methods is that time is not a metric value. Time has a form of event ordering there.

Each of the enumerated types of analysis uses its own form of the model description. Consequently, the user has to transform the model description from one form in another. There is no guaranties that the

model descriptions before and after these transformations are adequate. Also, there is no formal methods to ascertain the differences between these forms of the same model. As a result, this transformation is a source of mistakes.

The simulation modeling system which pretends to implement the combined approach has to meet the following requirements:

- this modeling system as algorithmic system must have a well known list of algorithmic solvable problems;
- this system must provide the possibilities of description of both the program behavior and the hardware operational characteristics to evaluate their quantitative indexes of performance;
- this system must simulate correctly two kinds of parallelism: concurrency and interleaving.
- this system must support the request-reply nature of interacting inherent of a computer system operating.

None of the existing simulation modeling system meets these requirements. The theoretical aspects of these problems have been investigated in details in ¹.

Basically, we will discuss a language aspects of combined approach here. However, before this discussing we have to the comment the second requirement. The point is that the description of a program behavior as a sequence of events is ambiguous (for more information about this ambiguity see ²). To overcome this ambiguity we introduce the notion of steps. Our conception is that a program process develops by step. A step is a pair that consist of the influence and the reaction. The influence is the receiving a message of certain type and a control from another process (about messages and their types see further). The reaction is the sending a message of certain type and passing control to another process.

We propose the specific modeling scheme called model with messages (M^2) as a basis of combined approach. This modeling scheme results from ³.

2. General description of modeling scheme

The model is a set of processes. The processes interact with each other, using the messages. The set of messages is partitioned into a finite number of equivalence classes, each is called a message type. Besides a type the message has other attributes, which will be discussed later.

Every process has the set of buffers to receive and to store the messages. This set must be described in the process. There are two kinds of buffers: internal and external. The process is able to read from and to write into its internal buffer. The process is able only to write into its external buffer. The internal buffer of a process and some external buffers of other processes are bound before execution of a model. So the process names other processes through their external buffers names.

Every process attempt to read a message from the empty buffer will suspend this process until the reception of the first message from this buffer. The process is also suspended, when it sends a message into a full buffer.

A process has the following structure:

```
<process> ::= process <name>
  ( <buffers description> )
  { <internal objects description>
    <process body> }
```

The buffers (internal as well as external) and the msg-variables are described in the buffers description section. Msg-variable is the local buffer for one message. The value of msg-variable has a structure with three fields: type, priority and delay. Msg-variables may be initialized when they are described. For example:

```
msg  a, buff = (CAT, 3.5, 1.0);
```

The ordering of messages in the internal buffer and its length must be defined in buffer description. In our system there are four kinds of the ordering: queue, stack, queue with the priorities and undetermined ordering. The length is denoted as a non-negative integer or * (non-limited length). This characteristics of the internal buffer

may be automatically modified before the algorithmic analyses. The characteristics of the external buffers are not supposed to be described, because the external buffers will be connected with the internal buffers. The internal buffers may be initialized with messages. The examples of a buffers description are the following:

```

buf  b1 (QUEUE,16), bbb (STACK,*);
buf  b2 (UNDET,16) = { (A,0,3), (BYE,2.3,7) } ;
stk  s1, hur, prpa;

```

The body of the process is formed from M^2 -statements. The syntax of the M^2 -statement is:

```

<M2-statement> ::= <msg-variable> = <message>; |
  send (<message>, <external buffer>); |
  receive (<internal buffer>); |
  complex (<code>); |
  while (<condition>) <M2-statement> |
  repeat <M2-statement>
    until (<condition>); |
  for (<integer>) <M2-statement> |
  if (<condition>) <M2-statement>
    else <M2-statement> |
  if (<condition>) <M2-statement> |
  switch (<var&>) {
    case <value&>: <list of M2-statements>
      ...
    default: <list of M2-statements> } |
  break; |
  continue; |
  stop; |
  { <list of M2-statements> } |
  <dummy statement>

```

```

<message> ::= <msg-variable> | <message-constant>

```

```

<message-constant> ::= (<type>, <delay>, <priority>) |
  (<type>, <delay>) | (type)

```

<type> ::= <string of letters and digits>

<delay> ::= <float>

<priority> ::= <float>

<list of M²-statements> ::=

**<M²-statement> | <list of M²-statements>
<M²-statement>**

The **send** statement is used to transmit the message into the external buffer. The **receive** statement is used to get the message out of the buffer.

The **complex** statement is used for two purposes: to project the model activity on model metric time and to simulate some aspects of the internal process activity. The **sleep** (<del>) statement serves for the first purpose. It increases model time on a <del> value. Also, model time can be altered by means the delay field of the last message, and in some other way.

The C language and the function, supporting the statistics and a random number generation, are used for the second purpose. These two purposes can be achieved by means the <code> part of the **complex** statement.

The M²-statements are executed step by step. The **while**, **repeat**, **for**, **if**, **switch**, **break**, **continue**, **stop** statements are used to control the order of statements executing, and their semantics are traditional.

The conditions in these statements are the logical expressions like a C language logical expressions. They may contain the internal variables, msg-variables and functions **type** and **numb**. The **type** (<buf>, <num>) function returns the type of the message with <num> number in the <buf> buffer. The **numb** (<buf>) function returns the quantity of messages in the <buf> buffer.

The body of process is the sequence of steps. Its syntax is:

<process body> ::= <list of steps>

<list of steps> ::=

```

<step> | <list of steps> <step>

<step> ::= <M2-statement receive> <reaction> |
    <process with parameters> |
    while (<condition>) { <list of steps> } |
    repeat <list of steps> until (<condition>); |
    for (<integer>) { <list of steps> } |
    if (<condition>) { <list of steps> } |
    if (<condition>) { <list of steps> }
        else { <list of steps> } |
    switch (<var&>) {
        case <value&>: <list of steps>
            ...
        default: <list of steps> } |
    break; |
    continue; |
    stop; |

<reaction> ::= <list of M2-statements, the execution of
which must
    always end with send statement, and its execution
doesn't
    contain any other send or any other receive
statements>

```

The step may include many **send** statements, but only one of them can be executed.

It is worthwhile to mention that there are the processes with the parameters in our system, which is used for the technological purposes. We aren't going to describe it here.

All processes start simultaneously and run asynchronously. The process ends, if it achieves the end of program, or if the **stop** statement is executed. The model finishes its work, if all the processes have been finished.

3. Model transformation

The simulation model is transformed to its form for algorithmic analysis by the follow way:

- the internal objects descriptions are excluded;
- the complex statements are excluded;
- all conditions (in conditional statements and cycles), which contains the internal objects, are changed to undetermined conditions.

It is proved that after such transformation we have the model in algorithmic analysis form, which belongs to following algorithmic class of models:

- Turing machine class (TM);
- Petri net class (PN), if model unlimited buffers are unordered, and `type` and `numb` functions are not used with unlimited buffers in the model;
- P/V system class (PV), if model unlimited buffers are unordered, `type` and `numb` functions are not used with unlimited buffers, and there is only one type of messages in the model;
- finite-state automaton class (FS), if there are no unlimited buffers in it.

The algorithmic analysis system is able to transform models from TM class to PN class and from PN class to FM class. The way of the transformation from TM class to PN class is:

- the orderings of all unlimited buffers are changed to undetermined;
- the conditions with functions `type` and `numb`, related from unlimited buffers, are changed to undetermined conditions.

The user of the system must realize, that the transformations can extend (SM->TM, TM->PN) or reduce (PN->FS) the behavior of his model. He can use such instruments of system as filter for selecting of histories, which are belong to real behavior. The possibility of reasonable transformation is real, because DCS are the finite-state machines at last.

So, the user of system has the following opportunities:

- to make a compromise between expressive power and solvability (to select useful class of models);

- to use the methods of analysis of more "weak" classes of models, then the selected class;
- to use methods of analysis of more "strong" classes of models, with the corresponding expenses on transformation.

4. Algorithmic analysis

The main task of the algorithmic analysis is to check the feasibility of such a properties of the model in which time is not used as a metric value.

After the source model transformation to the form close to Petri net, many auxiliary developments of the model may appear. These new developments are the result of using the internal test operation in transformed model. In order to drop them, special preconditions are used.

The algorithmic analysis system (AAS)'s input information is the source text of model and formulas of the properties to be checked and preconditions. A precondition is an expressions that provides the restriction of system's behavior in terms of dropping some developments or describing the order of some other developments.

Such traditional properties of systems as safety, fairness, liveness and absence of deadlocks ^{4,5} are the examples of properties that AAS can check.

The result of AAS work is information about feasibility of the properties specified by the user.

The syntax of source system model and transformations of it for preparing for algorithmic analysis have been described before in this article.

Special language of properties description is used to formulate the properties to be checked. The semantics of this language is close to the semantics of the Temporal logic language described in ⁵. However, it's impressive facilities are enlarged by the introducing of temporal operators if-then-else, while, case etc.

To formulate properties of system it is essential to introduce the concepts of process state and system state. The process state is

meant to be the name of current step of the process. System state is a vector of states of all the processes.

The properties to be checked and preconditions are described in special Properties Definition Language (PDL) in forms of expressions. These expressions consist of process state functions, operators of the first ordered logic and operators of the temporal logic. One can use macro variables in order to store in them some parts of expressions. These variables are defined in assignment statements of PDL. The value of a variable is right part of the assignment statement concerned as a character string. Process state functions are two-armed boolean functions of the form $P(<\text{process name}>, <\text{process state name}>)$. These functions value is true iff the process with specified name is in the specified state.

The example of an assignment statement is

Reading = P(reader,reads)

here macro variable Reading is defined. The right part defines a process state functions which is true iff the process reader is in state reads. It will substitute for every appearance of the macro variable.

The expression

if Reading then Future(Writing)

describes the fairness property, that means that if the system was (in the past) in the state where condition Reading was true, then in future condition Writing will be true. Here Reading and Writing are macro variables.

The expression

Always(not(P(Proc1,crit_sect)andP(proc2,crit_sect))

denote the safety property. The meaning of it is that processes proc1 and proc2 never enter the critical section at the same time. Here the

critical section is denoted by the state `chit_sect` of these processes.

Here is an example of typical precondition:

`Past(State1) and Now(user(Condition)) and Now(Fork) -> >`
`Prohibited(Way1) and Order(Way2,Way3)`

The meaning of this precondition is the following.

If the condition `State1` was true,
 and now the condition defined by user function `Condition` is true,
 and now the system is in the state where `Fork` is true, then in
 future the system will not be able to pass such a states in which
 condition `Way1` is true,

and the states with true `Way2` and `Way3` the system will be able to
 path only in specified order.

The following is a brief description of how AAS works.

Before the algorithmic analysis AAS transforms the system to the
 special form that is a partial ordered set of processes steps. The
 relation of partial order covers both the steps within the single
 process and to the steps of different processes. It has been proved
 that that one can represent this set as a forest of trees, where nodes
 represent processes steps and arcs represent the relation of partial
 order. AAS passes this structure in special way and checks specified
 properties. The pass process is controlled by preconditions.

5. Conclusion

We have shown the combined approach to distributed computer
 system simulation that provides both algorithmic and quantitative
 analysis of models. Special kind of model organization based on
 request/reply scheme and the process step notion have been proposed.
 In terms of algorithmic solvability this model is proved to be between
 Petri nets and P/V systems. It is also proved that the model can be
 represented in a form of partial ordered set of program steps. Special
 languages for model description and model properties description have

been shown by examples. Although we don't have enough space here, we find it worthwhile to mention the deep philosophical relation between request/reply scheme and object oriented approach.

Nowadays two implementations of modeling system have been produced: for single computer environment and for distributed computer environment.

REFERENCES

1. Smeliansky R.L. *The model of distributed computer system operating.* Vestnik Moscow University. Ser.N15,N3,1990.
2. Milner R. *Calculus of communicating systems.* LNCS V.92,1980.
3. Riddle W.E. *An approach to software system modeling and analysis.* Computer Languages 4, 49-66, 1979.
4. Owicki S., Lamport L. *Proving Liveness Properties of Concurrent Programs.* ACM Transactions on Programming Language and systems, Vol 4, No 3, July 1982.Pages 455-459.
5. E.Allen Emerson, Chin-Laung Lei *Modalities for Model Checking: Branching time logic strikes back.* Science of Computer programming 8(1987) 257-306.

SYNTHESIS OF MODELS FOR MULTIPROCESSOR SYSTEM EVALUATION

VICTOR KHABAS

*Computing Center, Siberian Division,
USSR Academy of Sciences, Novosibirsk, 630090, USSR*

ABSTRACT

Some aspects of synthesis of simulation models for bus-structured multiprocessor systems are considered. With a view to describe multiprocessor structures and the rules of their operation, a new mathematical object is introduced which is named Hyper Queuing Network.

Based on the Hyper Queuing Network presentation for multiprocessor systems with the global bus, an application software package has been developed for the synthesis of simulation models of multiprocessor systems and performance evaluation of such systems.

The conducted tests have demonstrated high efficiency of the proposed method. Synthesis of a simulation model, which used to take months or at least weeks, now takes a few minutes and depends on how fast the user responds to the computer.

1. Introduction

In recent 10 - 15 years the multiprocessor systems have become widely spread in processing highly-parallel data. Such systems are composed of processors, memories and an interconnection network.

The process of operation of multiprocessor systems is normally described by an alternate sequence of computation and synchronization periods. During the computation period the processors asynchronously process data. Characteristic of this period is that every processor performs uniform actions, i.e. generates the memory access for reading (writing) information, then processes it and prepares a new access. This is followed by the synchronization period.

Usually the behaviour of the system in the synchronization period may be easily predicted. It is known that the computer performance in the computation period is highly degraded due to the interference among processors during information exchanges. Such

conflicts take place when several processors simultaneously address to the same bus or memory module. Due to the complexity of analysis of processes, estimating the value of conflicts in the computation period is a problem.

It is known that simulation is the basic method for the performance evaluation of computer systems. Here time costs at the stage of the creation of respective models are considerable.

The aim of this publication is to acquaint the reader with the results of automating the process of creation of simulation models for the class of multiprocessor systems with the global bus. The developed methods have made it possible to synthesize the necessary model on the user's request in a few minutes in the interactive mode on IBM PC.

This paper is a continuation of the investigation begun in ^{1,2}.

2. ARCHITECTURE DESCRIPTION

It has been chosen to describe multiprocessor systems at the following level. They are defined as a set of processor elements (called processor hereafter) which interact with the memories through the buses. The interconnection network is a bottleneck for such systems.

Architectures of multiprocessor systems are specified by the symbols of basic modules shown in Fig.1.

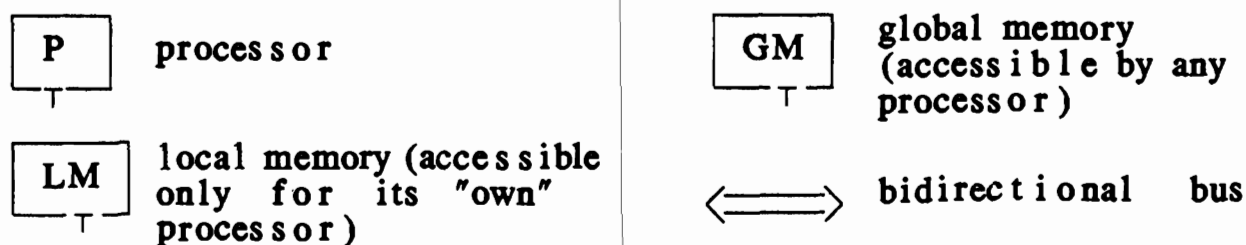


Figure 1. Basic modules

The system performance will be evaluated in the computation period. We will describe the status of the basic modules.

The processor has the following set of states:

off-line: the processor is preparing an access to memory or

is processing data;

waiting for a bus release;

interaction with a memory module (local or global): in this case the processor occupies all the buses necessary for organizing this interaction.

The processor executes the following repeated sequence of actions: off-line; addressing to a memory module and establishing a connection with it; interaction with the memory module.

The bus (memory module) has the following set of states:

not active: no processor has addressed to the bus (memory module);

active: the bus (memory module) is captured by a processor.

A bus (memory module) cannot be simultaneously captured by more than one processor. When two or more processors intend to catch the same bus (memory module), a conflict is arising.

3. SIMULATION METHODS

Analytical analysis and simulation are known to be the key methods employed for the performance evaluation of computer systems.

Analytical analysis is normally used in rough approximations of operating conditions of the systems. As a result, it has a restricted application field for a wide range of architectures.

Simulation has a wider and less specific application area since it makes possible to allow for any degree of detalization of a process being examined. Moreover, up-to-date simulation languages like GPSS and SIMULA contain efficient means for presenting the complex systems being simulated by queuing systems.

In using simulation models there are some common problems like parameterization and interpretation of results. However, in modeling multiprocessor systems, some specific problems arise. They are caused by the fact that every model is designed for a specific multiprocessor architecture.

As a result, even most insignificant changes in the structure of a system can bring about the need to develop another simulation

model because the present one can no longer be used.

Why does such a situation arise?

It is often attributed to the fact that the representation of multiprocessor systems by the queuing systems is impossible. Hence universal means for the description and simulation of discrete network systems cannot be applied. This necessitates the creation of a new simulation model every time for each architecture variant.

For example, let us consider a system with segmented memory depicted in Fig.2.

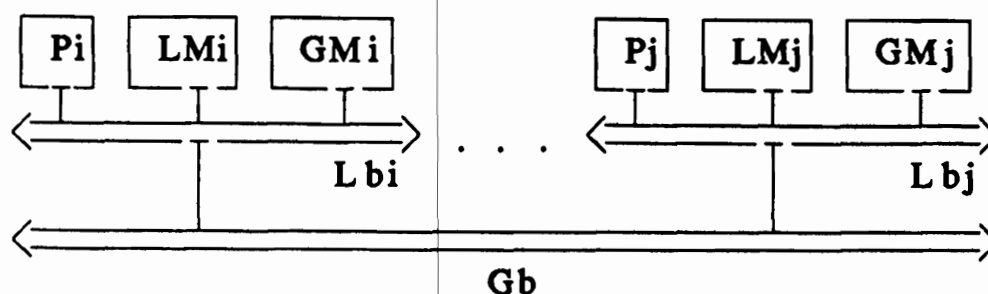


Figure 2. System with segmented memory.

Gb is a global bus, Lb is a local bus

In order to organize the interaction of the i -th processor with the j -th global memory, the processor has at first to capture the buses LBi, GB and LBj. If any of them has been earlier occupied, then the processor will wait until the bus becomes not active. It is only after all the necessary buses have been captured that the interaction follows. Note that the buses are captured successively in the direction from the processor to an appropriate memory module. Here $Lbi \rightarrow Gb \rightarrow Lbj$.

Clearly, the traditional representation of such an interaction by a set of queuing systems is impossible because the classical queuing theory does not support the methods for the description of preliminary capture of a group of service centers and the following simultaneous servicing of the message on them.

This interaction may obviously be program-realized. However it will be a particular realization in terms of a given architecture variant of the system.

Let us turn to the system depicted in Fig. 3 and pay attention to the interaction of the processor P_i with the memory module G_{mj} . Clearly, the program realization of the previous interaction has to be done again because one should take into account the new buses L_{bi2} and L_{bj2} , the order of their capture, time allowed for capture and disciplines of arbitration. As a result, it becomes necessary to create a new simulation model.

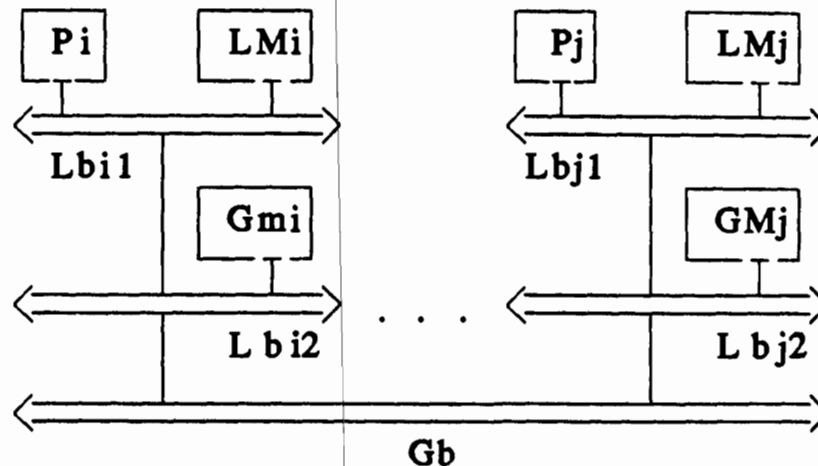


Figure 3. Second variant of multiprocessor system

4. AUTOMATION OF SIMULATION

There are no doubts as to the importance of automating the process of simulation of multiprocessor systems. The main emphasis here is on synthesizing simulation models on the user's request.

The following approach is proposed:

a new mathematical object - Hyper Queuing Network (HQN) - has been introduced;

using this object, methods for the representation of operation of multiprocessor systems have been developed.

In order to examine the approach in question for efficiency, a software package for the performance evaluation of multiprocessor systems with the global bus has been written and realized on IBM PC. The programs of the package are capable to synthesize necessary models on the user's request in the interactive mode and to simulate the

behaviour of the systems.

4.1. Hyper Queuing Network

A HQN contains a set of service centers $\mathcal{M}$ and a set of types of customers $\mathcal{R}$. The terms "service center" and "customer" have the same meaning as in the classical queuing theory ³. In addition, a set of hyper centers $\mathcal{Z}$ is introduced.

The function $f^{(*)}$ is defined on the set of hyper centers $\mathcal{Z}$ and with respect to a customer of the type $*$ ($*$ -customer hereafter). The range of $f^{(*)}$ includes all the subsets of the set of service centers $f^{(*)} : \mathcal{Z} \rightarrow 2^{\mathcal{M}}, * \in \mathcal{R}$.

The global route $\text{Glob}(*)$ of the $*$ -th customer is defined as a sequence of hyper centers from the domain of definition of the function $f^{(*)}$.

For every hyper center λ and $*$ -customer, for which $f^{(*)}(\lambda)$ has been specified, the local route $\text{Loc}(*, \lambda)$ is defined, $\lambda \in \mathcal{Z}$. A local route sets the order in which an $*$ -customer occupies the centers of the set of $f^{(*)}(\lambda)$.

The $*$ -th customer is serially serviced at the hyper centers of the global route $\text{Glob}(*)$. The servicing itself includes the preliminary capture of all centers of the set $f^{(*)}(\lambda)$ by the customer and subsequent simultaneous servicing at these centers.

Occupying a service center is a conventional process of servicing the customer (see ³) with the only difference being that the servicing channel is not released after the occupying is completed. The channel remains inaccessible for other customers, and the customer itself goes to the next service center of the hyper center, as prescribed by the local route.

After the customer has captured the channel in each service center of its local route, simultaneous servicing begins. After all the previously captured channels have been released, the customer moves to another hyper center on its global route.

Thus, the HQN allows to get a clear and easy description of interactions between processors and memories through buses. This can be illustrated by multiprocessor systems with the global bus which are

of practical interest.

The structure of a system of such a class may be presented as a set of processor nests which are "hanged" on the global bus. Let the processors of the system be numbered by integers. Then the i -processor nest includes all buses and memories which are accessible to the i -processor without its exit to the global bus and the processor itself.

Thus, for the system shown in Fig.2, the i -th nest contains P_i , L_{bi} , L_{mi} and G_{mi} and for the system shown in Fig.3 it contains P_i , L_{bi1} , L_{bi2} , L_{mi} and G_{mi} , respectively.

It follows from the definition of HQN that any processor-memory interaction will be represented by a hyper center, whose topology will contain an ordered sequence of service centers (local route), corresponding to the string: `<processor-bus-...-bus-memory module>`.

The off-line regime of a processor will be specified by a hyper center with the only service center.

4.2. Description of the software package

A software package of the model synthesis (Synthesizer) and the simulation (Simulator) for multiprocessor systems with the global bus has been written in PASCAL (IBM PC).

Synthesis is the process of creation of a data structure, which corresponds to the HQN representation of a system being simulated. Thus the creation of an appropriate data structure is equivalent to the construction of a simulation model itself. Then the data structure created in the Synthesizer is used as input data for the Simulator. The latter unit simulates the behaviour of a multiprocessor system.

It is further assumed that the class of systems being simulated are symmetrical, i.e. all processor nests contain an equal number of both buses and memories with similar performances. So, symmetry enables us to describe the operation of the entire system by the actions of one processor from any nest.

Consider the main functions of the Synthesizer unit.

DESCRIPTION OF PROCESSOR NEST The following menu is proposed to the user: `<processor-bus-...-bus-memory module-...-memory module>`,

here the buses and the memories belong to the same processor nest.

The menu is displayed on the screen with an excessive number of buses and memory modules. The necessary ones are selected. The menu is displayed every time in order to specify every interaction in the processor nest.

For example, in order to describe the off-line action of a processor, it is necessary to mark the first element in the corresponding string. If we wish to describe the interaction of the i -th processor with the module G_{mi} , elements are to be selected which correspond to <processor>, <bus> and <memory module>, respectively.

Clearly, a collection of such strings describes the trajectory of the global route within the processor nest. Local routes within each hyper center are specified by a succession in which the processor captures the elements.

Since the system is symmetrical, the specified interactions will be similarly described for a processor from any nest.

INTERACTIONS THROUGH GLOBAL BUS A menu with an excessive list of elements for describing the interaction between the processor and the memory module from an "alien" nest is displayed.

The menu consists of two lines. The upper line is <processor-bus-...-bus-global bus> and the lower line is <bus-...-bus-memory module-...-memory module>. Here the user also chooses the elements necessary for describing interaction through the global bus. After interaction has been specified, the menu is updated.

PARAMETERIZATION AND SWITCHINGS The order of switchings for a processor is specified, as well as kinds of distributions for each interaction. Out of the set of arbitration disciplines, the ones suitable to the appropriate buses are selected. The initial state of the model corresponds to the off-line action for all processors.

Let a description of the Simulator unit be given. The constructed HQN is simulated and statistical information is accumulated. The regenerative method of analysis is used to ensure the correct processing of statistical data ⁴. The regeneration cycle consists of intervals of active and non-active states of the global bus.

Simulation results are output in the user-suitable form. Major performances of the system include: utilization and workload of the processors, throughput of the buses, evaluation of the lengths of queues before buses and memories.

The created software package has been used to assess the performance of multiprocessor systems from ⁵. The time needed to create a program of the model on the user's request has been in the focus of attention, because the process of writing and debugging a simulation program is very time- and labour-consuming. It usually takes months or at least weeks while the simulation experiment lasts about 10 min. Tests have shown that the maximum time needed to construct any model is about 5 min.

5. Conclusion

A new mathematical object - Hyper Queuing Network - has been introduced. The representation of multiprocessor systems by HQN allows to get a simple and natural description of their structures and rules of operation.

A software package of synthesis and simulation of the behaviour of multiprocessor systems with the global bus has been developed. The control experiments show that the proposed method is highly efficient.

Future investigations will be directed towards synthesizing simulation models for multiprocessor systems in a more general form.

6. References

1. G.F.Verzakov and V.B.Khabas , Automation of Simulation of Multimicroprocessor Systems. - Novosibirsk, 1987. - 24 p. - (Preprint USSR Acad. of Sciences, Sib. Div., Comp. Center; N 723). (In Russian).
2. V.B.Khabas and E.A.Kujeleva, Simulation of Bus-Oriented Multiprocessor Systems . Tezisy Vsesojuznoj Nauchnoj Konferentsii "Modelirovanie Sistem Informatiki". Novosibirsk. 1990. P. 141-143. (In Russian).
3. L.Kleinrock, Queuing Systems. V. 1 (John Wiley & Sons, 1974).

4. M.A.Crane and A.J.Lemoine, An Introduction to the Regenerative Method for Simulation Analysis (Springer-Verlag, Berlin-Heidelberg-New York. 1977).

5. I.V.Molonov and R.N.Smeljanskij, Comparative Analysis Some Architectures of Multiprocessor Systems with the Global Bus. In: Programmnoje Oborudovanie i Voprosy Prinjatija Reshenij (Moscow State University, 1989. P. 66-80). (In Russian).

HIGHLY EFFICIENT ALGORITHMS FOR STATIC ANALYSIS OF PARALLEL PROGRAMS

N.N. MIRENKOV, S.A. SIMONOV

*Computing Center, Siberian Division,
USSR Academy of Sciences,
630090, Novosibirsk, USSR*

ABSTRACT

We suggest linear algorithms for the static analysis of deadlock-freedom and uniqueness of parallel programs in the PA(G)-language. The language constructs describe systems of sequential processes with collective schemes of interactions. The interaction statements make it possible to define communications, synchronization and ordering of information and control dependent subprocesses.

1. Introduction

Static analysis of parallel programs is a part of their debugging. There were many attempts to use static analysis for the item. Great part of the attempts are founded on extracting the set of critical states of programs and then either to prove that there are no unwaiting states (for example deadlock one) or to prove that undesirable states are unattainable (the control can't be at those states). For this kind of proving usually used: i) the method of theorem and program correctness proving (this is some analog of Floyd method in sequential programming)¹; ii) the method of parallel program behavior analysis with the help of its simulating by finite-state machine or network models^{2,3}; iii) calculation of communicative sequences (CCS)⁴; iv) analysis of partial ordered sets of events in parallel processes⁵ and v) various graph algorithms. The most methods for static analysis have exponential complexity.

Paper⁶ proposed the special languages PA(I) and PA(G) for the description of parallel algorithms and formulated a theorem establishing deadlock-freedom and uniqueness of parallel programs under the assumption that every loop terminates. In paper⁷ we proved

this theorem by describing polynomial algorithms for the analysis of features indicated for PA(I) programs. These algorithms are also suited to programs in PA(G), but the peculiarities of this language allow linear algorithms⁸.

In this paper, a brief description of the PA(G) language and the way it may be used is followed by the formulation of static deadlock-freedom and uniqueness analysis and an estimate of its complexity.

2. The PA(G) language

A parallel program (p-program) in the language PA(G) represents a set of interacting sequential processes each of which is executed with its data stored in a local memory. A process may be either starting or being initiated during execution. Each process contains conventional constructions and collective interaction statements (IS). These statements define some kind of communication and conditions of their execution. Let j be the name of a process using the statements discussed below. Then

BE (L, B, L_j) defines (BE stands for Broadcast Exchange) broadcasting the value of variable B from process L to all other processes (active with respect to interactions, see below) that assign it to the local variable L_j .

SE ($T, L_j, L_{j \pm 1}$) defines (SE stands for Shift Exchange) a shift of the value of variable L_j which is in the local memory of the process j to variable $L_{j \pm 1}$ of the process $j \pm 1$. The direction of the shift depends on the parameter T . For example, if $T = 'L'$ the shift is to the left; $T = 'R'$ - to the right, $T = 'LE'$ defines shift from even processes alone to the left, $T = 'RO'$ - of odd processes alone to the right. Cyclic shift (modulo N) is defined by adding the letter 'C'. $T = 'RC'$, for instance, defines a cyclic shift to the right to a process with a greater number; the last process is shifted to that having the smallest number.

GB (B, F_j, S_j', S_j'') defines (GB stands for the General Branch) a simultaneous change of the control flow in all processes. B

is a logical function; its arguments are F_j (in process i the name of the argument is F_i). If $B = \text{true}$ then all processes go to S_j' otherwise to S_j'' . There are two distinguishable cases of GB usage differing by their syntax. The first of them is GB used as a conditional statement. "Then" part opens with the statement S_j' , while "else" part opens with the statement S_j'' . The other case of GB use is concerned with the formation of the so-called generalized "while...do..." loops to be explained below.

GGB (*List*) defines (GGB stands for the general *goto*) the generalized unconditional jump of control in all processes in *List*. This statement is responsible for starting and eliminating of processes.

For a higher flexibility two statements are added that do not define interactions but influence them. These statements are $\delta:=0$ and $\delta:=1$. The former makes the execution process passive to collective interactions, while the latter brings it back to the active state.

The above set of interacting statements is a sufficient tool for describing or simulating any communication schemes. The schemes are called "collective" because the same statement is to be present in all active processes for the interaction to take place. If the process i does not participate in an interaction this should be explicitly shown by setting δ equal to 0. So δ is a kind of mask for the interaction. BE, SE, GB statements are executed if a corresponding statement has to be executed by all processes (except for passive ones). This fact means that synchronization and interaction are combined in the language. For synchronization alone one can use an empty statement (i.e. BE, SE, GB without parameters). Interactions are identified by a pair of IS labels: the first one of them identifies the statement in the text, while the second one identifies the interaction. Clearly, all first labels should be different unlike the second ones which may coincide if they define the same interaction on a separate execution path. The GGT statement is special: even if it is present only in one active process, it can be executed. So the GGT statement may at the moment of its execution interrupt other processes or generate a new one. Similarly, the execution of $\delta:=0$

($\delta:=1$) can at any moment make a process passive (active) to collective interactions. To provide for a less arbitrary execution of GGT, $\delta:=0$, and $\delta:=1$ statements as well as initiation and finish conditions (IFC) need be introduced into the language. IFC should always accompany the GGT, $\delta:=0$ and $\delta:=1$ statements. The moment of GGT appearing as well as $\delta:=0$ or $\delta:=1$ in any process is called an intermediate event of the PA(G) program. The syntax of IFC is like a special form of comment to the GGT, $\delta:=0$, $\delta:=1$ statements: *<initial condition>* ; *<finish condition>*. The conditions are logical expressions $P = P(R_1) \vee P(R_2) \vee \dots \vee P(R_h)$, $h > 0$. Here, in initial condition (IC), $P(R_l) = \text{true}$, $l=1, \dots, h$ if the interaction labeled by R_l has taken place, otherwise $P(R_l) = \text{false}$. In the finish condition (FC), $P(R_l) = \text{false}$ if the interaction labeled by R_l has taken place, otherwise $P(R_l) = \text{true}$. Execution of a statement having IFC may start only after both control reaches it and its IFC are held.

To conclude the description of the PA(G) language, let us consider the way to define loops. A loop containing no IS in its body may be written as an ordinary *while...do...* construction. These loops may be nested in other loops and *if...then...else...* constructions. We *a priori* assume the loops to terminate.

A loop containing IS must be a part of the so-called generalized *while...do* loops. A generalized loop consists of several loops each of which is executed in its own active process. Note that all processes that do not participate in the execution of a generalized loop must be passive at the moment. A representation of a generalized loop in a process is written with the help of GB. The head of the loop being

while GB (B, F_j , S_j' , S_j'') *do* ...,

where S_j' is the first statement of the loop body, and S_j'' is the statement immediately following the loop body. The repetition of a generalized loop is defined by an IS. Since the decision whether to continue or terminate a generalized loop is unique for all processes, a process synchronization take places after every iteration. It is also clear that the number of iterations in all processes is the same. If the loop body contains other IS's, they may communicate with other processes executing the loop, too. If the body of a generalized loop

contains another statement like $\delta:=0$, $\delta:=1$, GGT, it must be accompanied by an IFC which refers to the previous execution of an IS proceeding the statement (for initiation conditions) or following it (for finish conditions). It suffices to use a second label alone to specify an interaction. Processes not participating in the execution of a generalized loop are to be explicitly marked as passive throughout the execution. When the execution of the loop is completed all processes must be synchronized with the help of an additional, possibly empty, IS placed after the loop body. It must be used as an intermediate event in the initiation condition of the statement $\delta:=1$ in any passive process.

Consider a program for parallel sorting to illustrate the way PA(G) may be used. The known algorithm is based on a simultaneous permutation (ordering) of elements in pairs. Let us consider an array of n elements to be ordered. Let these elements be distributed over n processes in such a manner that the i -th element is stored in the i -th process in variable x . It is required to reorder the array so that the i -th (by value) element belongs to the i -th process. To accomplish this, no more than n parallel permutations are needed. Consider a simple example. We are to sort the array (4,8,3,5,1). First, we compare the elements in odd pairs: (4,8) and (3,5). Since the relative order in these pairs is correct, we carry out no permutations. Then we compare the elements in even pairs (8,3) and (5,1). Note that the first pair is degenerated and consists of only one element. Since the order in both pairs is not correct, we do a simultaneous rearrangement of the elements in the pairs and then return to the first step. The maximum of n parallel comparisons and permutations is required. The permutation may be written as a SE statement. Each process may be executed as follows:

```

p:=1;
while GB ( v, p, M1, M2 ) do
  M1 : y:=x ;
      SE ( 'RO', x, y ) ;
      if y > x
      then p:=1; x,y:=y,x
      else p:=0;
      SE ( 'LE', y, x ) ;
      SE ( 'RE', x, y ) ;

```

```

if y > x
then p:=1; x,y:=y,x
else skip;
SE ( 'LO', y, x );

```

M2 : . . .

Here the local variable x contains an element of the array to be ordered, y is a temporary variable. Each one of the parallel processes executes a generalized loop headed by

while GB (V , P , $M1$, $M2$).

The condition for the loop repetition is the disjunction (see symbol V) of all p in all processes, where $p=1$ implies that in the current iteration of the loop the elements have been permuted either in i and $i-1$ or in i and $i+1$. If no permutation has occurred, $p=0$. The statement $SE('RO',x,y)$ performs shifting right the value of variable x of all odd processes to variable y of all even processes, then all even processes select the smaller element among their own x and the received y , and shift it to the left with the help of $SE('LE',y,x)$. As all even number processes compare their elements, so do the odd processes but here the comparison result is always false. We can say therefore, that the odd processes are just waiting for the minimum value to be chosen in even processes. This shift-and-compare structure ensures permutation of elements in pairs. After all permutations in the odd pairs of all processes are completed, odd and even processes exchange their roles and do the ordering in even pairs. The first half of a loop body orders odd pairs and the second one even pairs. The loop terminates when no permutation occurs, i.e. all p are zero. Note that iterations are executed simultaneously by all processes. So, if we can prove the first iteration of a generalized loop to be deadlock-free, the next one is deadlock-free, too. This becomes obvious if we bear in mind that every path through the loop body has been explored. It follows from the first iteration deadlock-freedom and the syntactic correctness of a generalized loop that any finite number of iterations will not lead to a deadlock.

3. Algorithm for deadlock-freedom analysis

A theorem defining deadlock-freedom conditions⁶ says that for a PA(G) program to be deadlock-free it is sufficient that the order of intermediate events is the same in all processes for any realization, and among active processes executing a statement BE(1,...) there is a process named 1. If we are interested in a free interpretation, then the condition is both necessary and sufficient.

Succession of intermediate events may be interpreted as word in an alphabet of IS's with their second label. The word results from a particular execution of a program. Then the sufficient deadlock-freedom condition can be formulated as the coincidence of words in all processes of any execution. However, we need only to compare the successions having the same order execution of GB's. Coincidence of words is too strict a demand, since some processes may not take part in an interaction if they are marked as passive with the help of the statement $\delta:=0$. Intermediate events in these processes may be ordered in another way but this does not imply a deadlock in a PA(G) program. So, in defining the coincidence of successions we should take into account that the succession $\delta:=0 \dots \delta:=1$ coincides with any others beginning with an intermediate event used in IC of the $\delta:=0$ statement and finishing with an intermediate event specified in FC of the $\delta:=1$ statement.

Formally, the coincidence of intermediate event successions can be expressed in terms of the distribution equivalence concept defined below. Let successions of intermediate events be represented as IS-successions and their second labels. Then

Definition: Two successions of intermediate events $(L_{i_1}, L_{i_2}, \dots, L_{i_n})$ and $(M_{j_1}, M_{j_2}, \dots, M_{j_k})$ are called distribution equivalent if either i) they are the same ($n=k$, $L_{i_h} = M_{j_h}$ for all $h=1 \dots n$) or ii) they may be divided into subsuccessions so that corresponding subsuccession either coincide or one of the subsuccession in a non-coincident pair begins with the statement $\delta:=0$ and finishes with the statement $\delta:=1$

and does not contain other statements like $\delta:=0$ and $\delta:=1$.

The complexity of comparing algorithm is no more than $n+k$, i.e. the sum of succession length.

A complete algorithm for deadlock-freedom analysis may be looked at as a simultaneous step by step move along all branches of processes with the test of the resulting IS succession for distribution equivalence. The test means that if we come across a IS in a process, then all other active processes must contain either the IS too, or the statement $\delta:=0$. The use of a GGT may also lead to a deadlock if the following two conditions do not hold: first, processes eliminated by GGT must be pregenerated by another GGT, or be starting and second, new GGT-generated processes must have the same successions of intermediate events as other processes. We feel it proper to add some details to the description of the algorithm. Let the program under analysis consists of 1 processes and the maximum depth of *if* statement nesting is k . Then the way the algorithm works in this context is as follows:

```

while not end of program do
  begin
    if a condition statement occurs
      then the number of process
            successions looked up (n) increases to  $n+1$  ;
            /* the new succession correspond to the then-part
               and the current one - to the else-part of the
               statement */
    if the end of a conditional statement occurs
      then the number of successions looked-up decreases by 1;
            /* the then-part succession is finished */
    if a GB statement occurs
      then
        if all active processes contain the same statement;
          then the number of successions looked-up is doubled;
          else report about the possibility of a deadlock;
    if the end of a GB statement occurs
      then the number of successions decreases exactly by the
            number it has increased at the GB beginning ;
    if a GGT statement occurs
      then
        if this statement generates a new process
          then number of succession increases by 1
          else /* if the statement eliminates processes */
            if the process to be eliminated exists
              then take it out of consideration
            else /* the process does not exist */

```

```

        report about the possibility of a deadlock ;
    if an IS occurs
    then
        if all active processes contain the same statement;
        then
            if this is BE(1,...)
            then
                if there is a process with the name 1
                then do nothing
                else report about the possibility of a deadlock
            else report about the possibility of a deadlock;
        else do nothing
    take next statement ;
end.

```

A process may remain passive until an intermediate event appears indicated by the nearest $\delta:=1$ statement after $\delta:=0$.

The way the algorithm works is comparable with the flow of a liquid advancing simultaneously along the channels which have the form of a processes digraph, as the flow progresses. It checks the coincidence of barriers that come its way. Since the analysis is made under the assumption that every loop spreads on one iteration, the estimated complexity of the algorithm is $O(k \cdot l \cdot t)$, where t is the maximum length of succession, l is the number of processes, and k is the greatest depth of *if*-nesting in all processes. However, as every statement is observed only once, we can take L as the length of longest process instead of $k \cdot t$, and the total length of all processes instead of $l \cdot L$.

4. Algorithm for uniqueness analysis

An execution of the deadlock-freedom PA(G) program without $\delta:=0$, $\delta:=1$, GGT statement is unique because synchronization and communication are combined. Usage of the statements $\delta:=0$, $\delta:=1$ and GGT may result in ambiguity because not all the details of their execution can be anticipated. To avoid ambiguity, every time the statement is used appropriate IFC should be specified; IFC's imply that the statement may be executed any moment following the events mentioned in IC and prior to the ones mentioned in FC. If no other interactions occur between these two events the execution of $\delta:=0$, $\delta:=1$, or GGT will not lead to ambiguity. The solution is the same for a GGT, if it used

for generating new processes. If it is employed to eliminate processes, the requirement is that no operation on data takes place between the IS's from the IFC. So, the check for uniqueness is reduced to that for correctness of IFC for each $\delta:=0$, $\delta:=1$, and GGT statement. In other words, a pair of intermediate events is to be selected such that the first is used in IC and the second in FC, both lying on the same path of a parallel program execution and no other events or simple operations occurring between the IFC events.

Due to the structuredness of PA(G) programs the test for IFC correctness can be readily performed and can be done in parsing.

5. Example

To illustrate the way the described algorithms act, look at a p-program building the Pascal triangle or, to be more exact, an $(n \times l)$ array R, whose elements $R[i,j]$ are zero when $j > i$, and the remaining elements form the triangle sought. The triangle elements are calculated by the following recurrent formula:

$$R[1,j] = 1; \quad j = 1..n, \quad R[i,1] = 0; \quad i = 1..l,$$

$$R[i,j] = R[i-1,j] + R[i-1,j-1]; \quad i,j > 1.$$

In a program of 1 processes the array R is distributed so that each of its columns is located in a one-dimensional array r comprising n elements. Line i is to be stored in the element i of the local array r. Predicate P is used which means "not more than" test of variable i. The j-process text of the program looks like that:

```

      i := 1;
      if j = 1 then r[i] := 1 else r[i] := 0;
K1-M1 : while GB ( P, i, S1, S2 ) do
      S1 :   i := i+1;
           if j = i
           then r[i] := 1;
K2-M2 :           SE('R', r[i-1], h)
           else
           if j > i
           then
               δ := 0; | M1 ; M2
               r[i] := 0;
               δ := 1; | M2 ; M3
           else

```

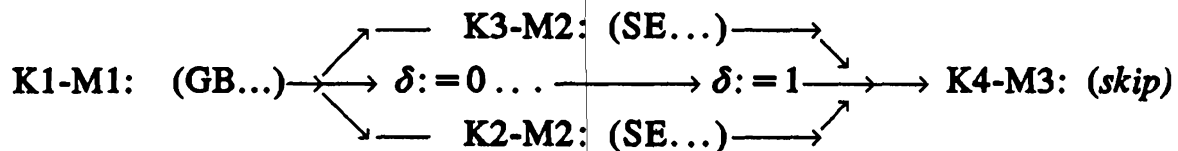

K3-M2 : SE('R', r[i-1], h);
 r[i] := r[i-1] + h

K4-M3 : skip

S2 :

In the generalized loop body there is only one IS - SE('R',r[i-1],h) which shifts the element R[i-1,j-1] used to calculate R[i,j] to the right. The shift is done from the local variable r[i-1] in the j-th process to the local variable h in the j+1-th process. Processes whose numbers are greater than i became passive prior to the execution of the IS labeled M2. Processes must recover their active state before a new iteration starts, i.e. before the statement "skip" labeled with M3 is activated.

It suffices to consider only one iteration of a generalized loop. The following successions of interactive statements are possible



As it is seen, all possible paths define distribution - equivalent successions of IS, and it follows from the processes being identical that the deadlock-free condition holds. The program is unique because no intermediate events may occur between the IFC events.

6. Conclusion

Peculiar of the PA(G) language which makes it useful for very large scale computations is the capability, first, to involve all (or about all) processes in every communication and, second, to describe synchronization and interaction in one statement. This gives us the double advantage of flexibility intrinsic to MIMD -languages and of *a priori* correctness, i.e. deadlock - freedom and uniqueness, inherent in SIMD -type languages. Clearly, these two contradictory properties are incompatible in one language directly. The PA(G) language is thus a compromise because it allows to write both identical and non-identical processes and to reorder interactions during execution.

Linear complexity of deadlock freedom and uniqueness analysis make the algorithms very workable. High effectiveness is achieved

because there is no need to spread iterations of loops and carry out step-by-step analysis of process texts to detect the properties in question. It is essential that the next step of the analysis can be made for a shorter 'new' parallel program whose deadlock freedom and uniqueness have been proved at the previous step. These algorithms as well as those from^{7,8} are based on process marking which makes it possible to order information and control dependent subprocesses. We believe that a similar approach can be used for Ada and Occam programs.

7. References

1. Levin G.M., Gries D. A proof technique for CSP // *Acta Informatica*. - 1981.- V.15.-P.281-302.
2. Apt K.R. A static analysis of CSP programs // *Lecture Notes in Computer Science*. - 1984. V.164 -P.1-17.
3. Taylor R.N. A general purpose algorithm for analysis concurrent programs // *Comm ACM*. - 1983. V.26, N5.- P.362-376.
4. Milner R. A calculus for communicating systems // *Lecture Notes in Computer Science*. - 1980.- V.92.
5. Pratt V.R. Modeling concurrency with partial orders // *International J. Parallel programming*.- 1987.- V15. -P.33-71.
6. Mirenkov N.N. (1987) Parallel algorithm and static analysis of parallel programs. // *Lecture Notes in Computer Science* - 1987.- V.269.
7. Mirenkov N.N., Simonov S.A. Polynomial algorithm for static analysis of parallel processes. //CONPAR 88, Ed. C.R.Jesshope and K.D.Reinartz. Cambridge University Press. British Computer Society 1989, pp.587-594.
8. Simonov S.A. Deadlock freedom and uniqueness analysis of parallel programs. *Proceedings of Parallel Computing in Engineering and Engineering Education*. UNESCO, Paris. Oct. 8-12, 1990, pp.267-272.

A PETRI NET ENTITY AS A FORMAL MODEL FOR LOTOS, A SPECIFICATION LANGUAGE FOR DISTRIBUTED AND CONCURRENT SYSTEMS

ANISIMOV N. A.

*Institute of Automation and Control Processes
Far East Division of the USSR Academy of Sciences
5 Radio Street, Vladivostok, 690032, USSR*

ABSTRACT

We suggest a formalism that is intended to implement a modular approach to the specification of complex distributed and concurrent systems. A new notion of an entity is introduced. It is defined as a Petri net together with a set of labellings, each labelling being treated as an access point intended for communication with other entities. A basic operation of parallel composition on entities is defined. Then we discuss modular aspects of LOTOS and show how they can be represented in terms of Petri net entity. For illustrative purposes we use a well-known alternating bit protocol.

1. Introduction

Specification languages play an important role in the software life cycle, especially for distributed and parallel systems. LOTOS¹⁰ is one of the most popular language for specification of distributed and concurrent information processing systems. It has been developed within ISO and mainly oriented for specification of protocols and services of computer networks. At present one can observe an intensive research activities concerning LOTOS⁷. There are numerous works on specification and verification techniques, methods of implementing LOTOS-specifications and conformance testing, various computer aided tools. Almost all protocols and services issued from ISO are accompanied with their LOTOS specifications. Thus LOTOS promises to become a very important tool for design of distributed and concurrent systems in the nearest future. A complete definition of LOTOS itself is contained in the ISO document¹⁰. A comprehensive introduction in LOTOS can be found in the paper³.

LOTOS is based on the process algebra that is slightly modified version of Milner's CCS¹² with some features of Hoare's CSP⁸ and other algebraic theories of concurrent processes. This component of LOTOS is used for the specification of behavioral aspects. Originally, an operational semantic of LOTOS is defined in the interleaving form with the aid of inference rules. But it is well recognized that an interleaving semantic is insufficient for many applications because it does not allow to represent a concurrency in an adequate way but only as a non-deterministic choice. In order to avoid this drawback considerable efforts have been done to equip CCS-like languages with "true-concurrency" semantics which is mainly based on Petri nets and related models¹⁴. Moreover, it is known a few attempts to equip LOTOS itself with Petri net semantics^{11,1}. Thus we can state the main advantage of Petri net semantic of LOTOS:

- a net-semantic of LOTOS allows to represent concurrency in the most adequate and natural way.

Moreover, other evident benefits of a net-theoretic approach to semantic definition are at hand:

- a visualization of LOTOS-specifications that is relied on a graphical form of Petri nets. This enhances understanding of LOTOS specifications, especially their dynamic properties since many practitioners are familiar with Petri nets;
- a possibility to exploit numerical models, methods, tools of Petri net theory⁵ in order to analyze LOTOS-specifications;
- a net-semantic provides an additional useful information for implementation of LOTOS-specifications since a net-semantic is a first step of synthesis of a system structure from its behaviour.

In the representation of LOTOS specifications by means of Petri nets there are a number of aspects. The one of these is a compositionality, i.e. a definition of LOTOS operators on the domain of Petri nets or related models. At present this aspect seems to be successively elaborated and, as a result, all LOTOS constructors have their counterparts in terms of Petri nets. Another aspect of a net-approach to semantic definition is a so called modularity, i.e. an ability to represent logical modules and compose more complex specifications. Unfortunately, the Petri net theory has a lack of modularity. We can mention only a few papers related to this subject^{6, 3}.

The aim of this paper is to cast a modularity into Petri nets in the form suitable for LOTOS specification. For this we suggest to use a notion of Petri net entity that has been introduced in the author's preprint². Here all definitions, however, will be done in slightly different way in order to account LOTOS constructors as far as possible. We will show how modular aspects of LOTOS can be treated in terms of Petri net entity. Note we don't consider net definitions of LOTOS operators (except parallel composition and hiding). We refer interested readers to numerous papers just mentioned.

2. Basic Notions

Let $A = \{a_1, a_2, \dots, a_m\}$ be a set. Let $\mathbb{N}$ be a set of all natural numbers and $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$. A multiset over a set A is a function $\mu: A \rightarrow \mathbb{N}_0$ that associates a non-negative integer to each element of A . Sometimes we will write a multiset over A as $\{n_1 a_1, n_2 a_2, \dots, n_m a_m\}$ where $n_i \in \mathbb{N}_0$ is a number of instances of $a_i \in A$ in the multiset. A set of all multisets over A will be denoted as μA .

2.1 Definition. A Place-Transition net (PT-net for short) is a tuple $N = \langle P, T, F, W, M_0 \rangle$ where

- (1) P is a finite set of places;
- (2) T is a finite set of transitions such that $P \cap T = \emptyset$;
- (3) $F \subseteq P \times T \cup T \times P$ is a flow relation;
- (4) $W: F \rightarrow \mathbb{N}$ is an arc weight function;
- (5) $M_0: P \rightarrow \mathbb{N}_0$ is an initial marking. □

Here we assume the capacity of all places of PT-nets is infinite, i.e. $K(p) = \infty$ for all $p \in P$. For any $x \in P \cup T$ we write $\bullet x = \{ny \mid (y, x) \in F, W(y, x) = n\}$ and $x^\bullet = \{ny \mid (x, y) \in F, W(x, y) = n\}$. Multisets $\bullet x$ and $x^\bullet$ are called pre-set and post-set of x , respectively. Functions $\bullet()$ and $()^\bullet$ can be naturally extended to the (multi) sets in such a way:

$$\bullet(n_1 t_1, \dots, n_m t_m) = n_1 \bullet t_1 \cup \dots \cup n_m \bullet t_m$$

Let Alph be a set of names and $\tau \in \text{Alph}$ a special distinguishable symbol that is associated with an unobservable action. We further define $\text{Act} = \mu\text{Alph} \cup \{\tau\}$.

2.2 Definition. A labelling of a PT-net $N = \langle P, T, F, W, M_0 \rangle$ is a tuple $\alpha = \langle \text{id}, \text{Alph}, \sigma \rangle$ where id is an identifier of the labelling, $\sigma: T \rightarrow \text{Act}$ is a labelling function. $\square$

This definition is an extension of a well-known classical definition of a labelling. For instance, each transition can be labelled by not a single label but a multiset of ones. A transitions labelled by τ are treated as invisible or internal actions. An identifier of a labelling allows us to distinguish various labelling of the same net.

Further we will need an operation of parallel composition of PT-nets. For this reason, first let us define a disjoint union of two PT-nets allowing to consider two nets to be a single one.

2.3 Definition. Let $N_1 = \langle P_1, T_1, F_1, W_1, M_{01} \rangle$ and $N_2 = \langle P_2, T_2, F_2, W_2, M_{02} \rangle$ be PT-nets with disjoint sets: $P_1 \cap P_2 = T_1 \cap T_2 = \emptyset$. Then disjoint union of these nets is a new PT-net $N = N_1 \cup N_2 = \langle P_1 \cup P_2, T_1 \cup T_2, F_1 \cup F_2, W_1 \cup W_2, M_{01} \cup M_{02} \rangle$. $\square$

Let α be a labelling of the net N_1 . Extend α onto the net $N_1 \cup N_2$ by defining the new labelling $\tilde{\alpha} = \langle \text{id}_\alpha, \text{Alph}_\alpha, \tilde{\sigma}_\alpha \rangle$ where $\tilde{\sigma}_\alpha(t) = \sigma_\alpha(t)$ when $t \in T_1$ and $\tilde{\sigma}_\alpha(t) = \tau$ when $t \notin T_1$.

Second define an unary operation of autocomposition in such a way.

2.4. Definition. Let N_1 be a net with two labelling $\alpha = \langle \text{id}_\alpha, \text{Alph}_\alpha, \sigma_\alpha \rangle$ and $\beta = \langle \text{id}_\beta, \text{Alph}_\beta, \sigma_\beta \rangle$ such that $\text{id}_\alpha = \text{id}_\beta$. Then autocomposition of the net N_1 with respect to α and β is defined to be the new net $N = \text{acom}_{\alpha\beta}(N_1)$ where

$$(1) \quad P = P_1, \quad M_0 = M_{01}$$

$$(2) \quad T = T_1 \cup T_{\text{syn}} - T_{\text{aut}} \text{ where}$$

$$T_{\text{syn}} = \{X \cup Y \mid X, Y \in T_1, \tau \in \sigma_\alpha(X) = \sigma_\beta(Y), \text{ the pair } (X, Y) \text{ is minimal and } X, Y \text{ are finite}\},$$

$$T_{\text{aut}} = \{t \in T_1 \mid \sigma_\alpha(t) \neq \tau \text{ or } \sigma_\beta(t) \neq \tau\}$$

$$\text{Here } \sigma(X) = \{\sigma(t) \mid t \in X\}$$

$$(3) \quad F = F_1 \cup F' - F'' \text{ and } W = W_1 \cup W' - W'' \text{ where } F', W' \text{ are determined from the relations } \bullet(X \cup Y) = \bullet X \cup \bullet Y, (X \cup Y)^\bullet = X^\bullet \cup Y^\bullet$$

$$F'' = \{(x, y) \mid x \in T_{\text{aut}} \text{ or } y \in T_{\text{aut}}\}, \quad W'' = \{(z, n) \mid z \in F''\} \quad \square$$

A pair (X, Y) is minimal if there exists no pair $(X', Y') : X' \subset X, Y' \subset Y$ or $X' \subset X, Y' \subset Y$ such that $\sigma_\alpha(X') = \sigma_\beta(Y')$.

In other words, the autocomposition does the following surgery on a net structure. The sets of places (and sequentially initial markings) remains the same. For the set of transitions T some extra transitions of synchronization, T_{syn} , are added. The extra transitions are constructed as multisets over T , i.e. $T_{\text{syn}} \subseteq \mu T$. The transitions which take part in synchronization T_{aut} are removed together with adjacent arcs forbidding an autonomous executing of these actions. Fig.1 shows the usage of this operation. Here only one new transition is added: $\{2t_1, t_3, 2t_5\}$ since $\sigma_\alpha(\{2t_1, t_3, 2t_5\}) = \sigma_\beta(\{2t_1, t_3, 2t_5\}) = \{4a, 4b\}$.

The claim of the finiteness of X, Y in the clause (2) of the Definition 2.4 is essential. For example, if we modify β such that $\sigma_\beta(t_5) = \{b, 2a\}$ then we can not build a synchronization transition with finite definition.

Let π be a labelling of the net N_1 . Extend π onto the net $N = \text{acom}_{\alpha\beta}(N_1)$ where α, β are also labellings of N_1 in such a way: $\tilde{\pi} = \langle \text{id}_\pi, \text{Alph}_\pi, \tilde{\sigma}_\pi \rangle$ where

$$\tilde{\sigma}_{\pi}(t) = \begin{cases} \sigma_{\pi}(t), & t \in T_1 \cap T \\ \sigma_{\pi}(X) \cup \sigma_{\pi}(Y), & (X \cup Y) \in T_{\text{syn}} \end{cases}$$

Now we are ready to define an operation of parallel composition. Let N_1 and N_2 be PT-nets with labelling α and β , respectively. Then a parallel composition of the nets with respect to labelling α and β is a new net $N = (N_1 \alpha \parallel_{\beta} N_2) = \text{acom}_{\alpha\beta}(N_1 \cup N_2)$. One can convince that this definition embodies an intuitive idea of parallel composition with the generalization onto multisets. For example, one can imagine the procedure depicted in Fig.1 as a parallel composition.

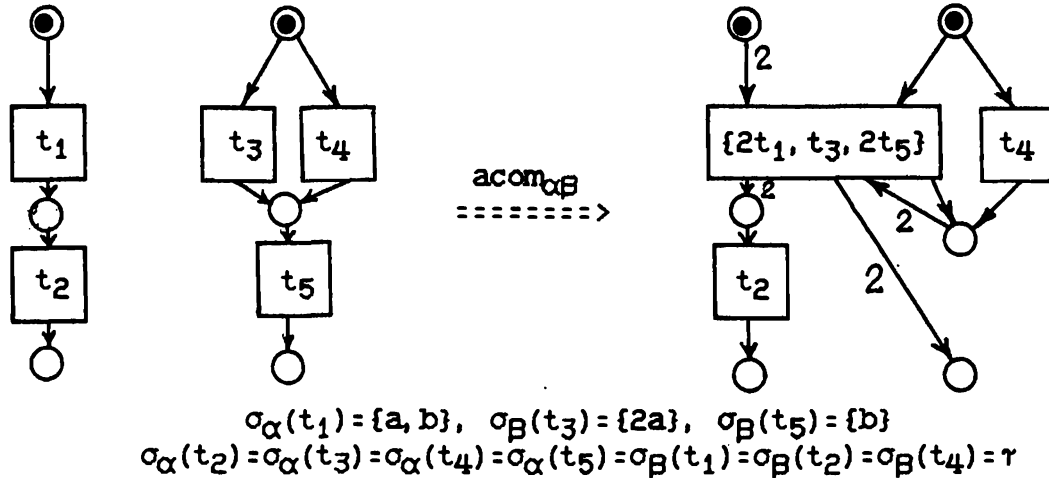


Fig.1. Autocomposition of nets.

We should noting that, in general, the calculation of the set T_{syn} is a very complex task. But we believe that many applications use only 1-1 or 1-n communications, i. e. in $X \cup Y$ either $|X|=1$ or $|Y|=1$.

3. A Formal Definition of Entity.

Now we are prepared to give a formal definition of entity. Note that term "entity" itself comes from the area of computer networks where it is used in the Reference model of OSI/ISO⁹.

3.1 Definition. An entity is a tuple $E = \langle N, \Gamma \rangle$ where

- (1) $N = \langle P, T, F, W, M_0 \rangle$ is a PT-net called a structure of the entity;
- (2) $\Gamma = \{\alpha_1, \dots, \alpha_n\}$ is a set of labelling where each labelling $\alpha_i = \langle id_i, \text{Alph}_i, \sigma_i \rangle$ is called an access point of the entity. An identifier id_i is called the name of the access point α_i . □

In other words, an entity is a PT-net equipped with a set of labelling. The net itself defines the structure of the entity and labelling are associated with its access points. One can communicate with an entity only via its access points. The observation of the behaviour of an entity is considered to be a particular, specific case of communication. It is clear that an entity can exhibit different behaviors at different points. A transition labelled by τ at a certain access point is considered to be invisible and, through it, one cannot communicate with the entity in this point. One and the same transition may be visible from two or more access points which may go under different names. Moreover, a transition may be visible from one access

point and invisible from another one. And, finally, a transition may be invisible from any point.

For this definition, we are obviously indifferent to the actual names of places and transitions (P and T) of a PT-net N. So we define an entity within its isomorphism class.

An entity E with access points $\alpha_1, \dots, \alpha_n$ will be denoted below by $E[\alpha_1, \dots, \alpha_n]$ and represented like a PT-net whose each transition is labelled by the tuple $\langle a_1, \dots, a_n \rangle \in \text{Act}_1 \times \dots \times \text{Act}_n$. By convention, entirely internal transitions, i.e. those labelled by $\langle \tau, \tau, \dots, \tau \rangle$, will be depicted without any labels. It is convenient to use a schematic representation of an entity as a box that has lines coming to it to represent access points.

Fig. 2 depicts schematic (a) and Petri net-like (b) representations of the entity which are corresponded to a very simple protocol. It has three access points. The point s is the service access point. It is defined on the alphabet $\text{Alph}_s = \{\text{DAT}\}$. The point p is the access point thorough which protocols commands, $\text{Alph}_p = \{\text{DT}, \text{ACK}^+, \text{ACK}^-\}$, can be executed. The point tm is intended for communication with a timer entity and defined on the alphabet $\text{Alph}_{tm} = \{\text{start}, \text{reset}, \text{expired}\}$.

Fig. 3 shows an example of timer entity. It has only one access point tm defined on the alphabet $\text{Alph} = \{\text{start}, \text{reset}, \text{expired}\}$.

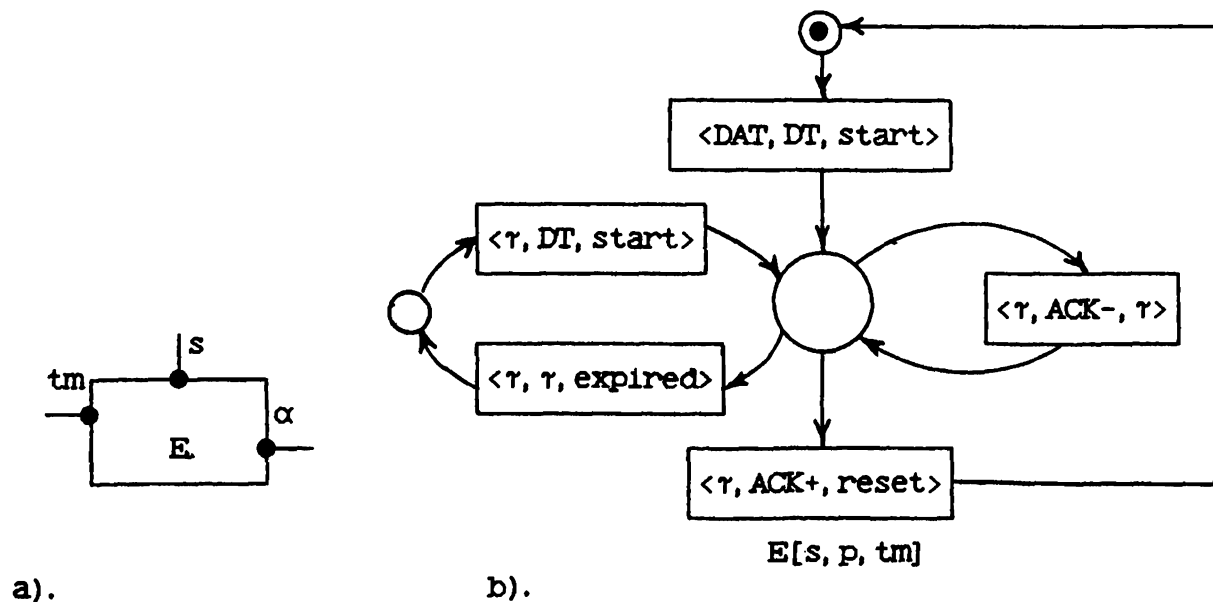


Fig. 2. An example of a Petri net entity.

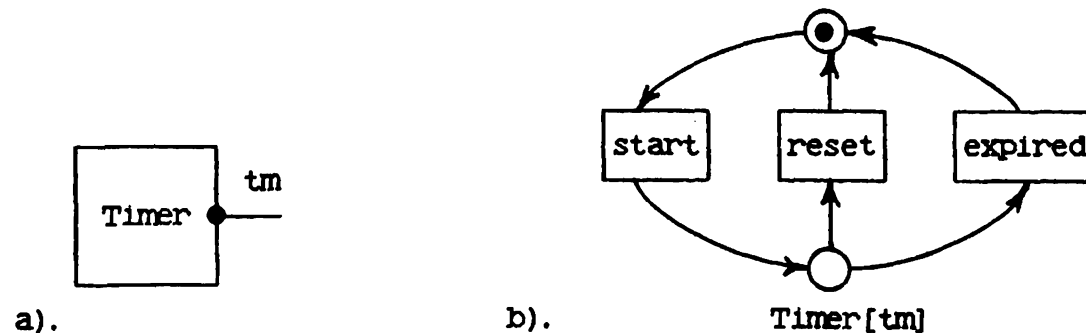


Fig. 3. A timer entity.

4. Operations on Entities

In this section we define two operations on entities which allow us to build complex systems from the simplest ones.

4.1 Definition. Let $E_1 = \langle N_1, \Gamma_1 \rangle$ and $E_2 = \langle N_2, \Gamma_2 \rangle$ be entities. Let $\alpha \in \Gamma_1$ and $\beta \in \Gamma_2$ be access points of these entities such that $id_\alpha = id_\beta = g$. Then a parallel composition of E_1 and E_2 with respect to g is defined to be an entity $E = E_1 \parallel [g] \parallel E_2$ where $N = \text{acom}_{\alpha\beta}(N_1 \cup N_2)$ and $\Gamma = \{\tilde{\pi} \mid \pi \in \Gamma_1 \cup \Gamma_2\}$. $\square$

Informally speaking, a parallel composition is a union of two entities with extra transitions added to implement an entity synchronization via access points α and β . Moreover, the labelling of the access points are united and extended onto the whole new entity. It should be noted that τ -transitions at α and β take no part in the communication.

Now it is not hard to extend the definition onto a more general version of the parallel composition with respect to a set of access points.

4.2 Definition. Let E_1 and E_2 be entities and $\{\alpha_1, \alpha_2, \dots, \alpha_n\} \subseteq \Gamma_1$ and $\{\beta_1, \dots, \beta_n\} \subseteq \Gamma_2$ such that $id_{\alpha_i} = id_{\beta_i} = g_i$ for all $i \in \{1, \dots, n\}$. Then a parallel composition of E_1 and E_2 w.r.t. $G = \{g_1, \dots, g_n\}$ is defined to be an entity $E = E_1 \parallel [g_1, \dots, g_n] \parallel E_2$ where $N = \text{acom}_{\alpha_1\beta_1}(\dots \text{acom}_{\alpha_n\beta_n}(N_1 \cup N_2))$, $\Gamma = \{\tilde{\pi} \mid \pi \in \Gamma_1 \cup \Gamma_2\}$. $\square$

The correctness of this operation is followed from the proposition whose proof is routine and omitted due to space limitations.

4.3 Proposition. Let N be a PT-net and $\alpha_1, \alpha_2, \beta_1, \beta_2$ its labelling. Then $\text{acom}_{\alpha_1\beta_1}(\text{acom}_{\alpha_2\beta_2}(N)) = \text{acom}_{\alpha_2\beta_2}(\text{acom}_{\alpha_1\beta_1}(N))$.

We also define a hide operation for entities that simply eliminates appropriate access points without changing a structure of an entity.

4.4 Definition. Let $E_1 = \langle N_1, \Gamma_1 \rangle$ be an entity and $\{\alpha_1, \dots, \alpha_n\} \subseteq \Gamma_1$. Then $E = \text{hide } id_1, \dots, id_n \text{ in } E_1 = \langle N_1, \Gamma_1 - \{\alpha_1, \dots, \alpha_n\} \rangle$. $\square$

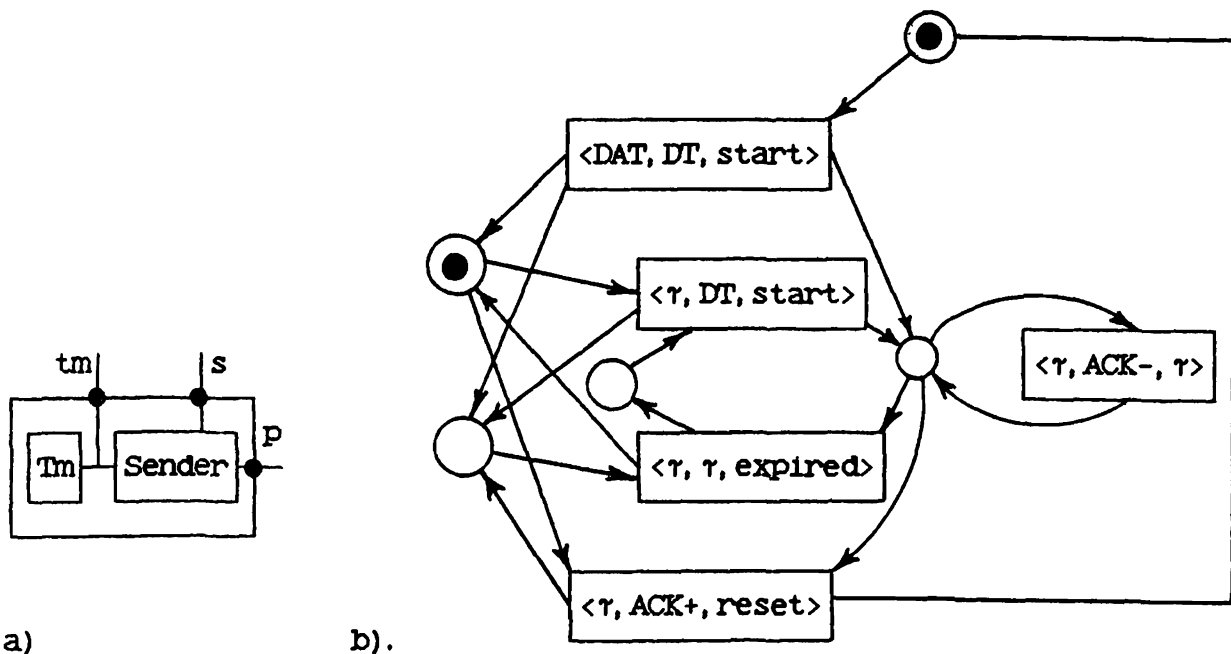


Fig. 4. Operation on entities.

In the Fig.4 we show an example of usage of the operation of parallel composition both in schematic and net-like forms.

5. Representing a modularity in LOTOS. An example of the alternating bit protocol.

The central modular construct in LOTOS is a process. A LOTOS specification is usually represented as a hierarchy of process definitions. We will assume familiarity with the main ideas of LOTOS. However let us briefly remember some features of LOTOS. Typical structure of process definition in simplified form can be represented as follows:

```

process Process_Name [ $g_1, \dots, g_n$ ] ( $x_1:t_1, \dots, x_m:t_m$ ): functionality :=
    behaviour expression
    where
        process definitions
    endproc

```

Each process is defined by unique name, Process_Name. [$g_1, \dots, g_n$] is a gate list through which process can interact with other processes. ($x_1:t_1, \dots, x_m:t_m$) is a parameter list where x_1 is a name of variable and t_1 is its sort. A *functionality* can be either exit or noexit keywords indicating whether process can successively terminated or not. Note that process definition may contain other process definitions forming a hierarchy of processes.

The behaviour expression specifies a temporal ordering of action executions using a set of LOTOS operators. The general structure of an action denotation has a form $g\alpha_1\alpha_2\dots\alpha_k$ where g is a gate name and attribute α_1 means either value declaration or variable declaration. The first one has the form "!"E where E is a value expression. The second one has the form "?x:t" where x is a name of variable and t is its sort identifier.

Suppose we have chosen a suitable Petri net semantic for LOTOS operators including a finite representation of processes with recursion. Note that in all the semantics each action is denoted as a net transition labelled by the action name.

Now we can represent a LOTOS-process by a Petri net entity in a very natural way. Indeed, each process can be represented by a single Petri net entity with 1-1 correspondence between access points and gates. A structure of the entity is built with the aid of net semantic from the behaviour expression, each transition being labelled by the corresponded action denotation. Parameters can be treated in terms of High-level Petri nets.

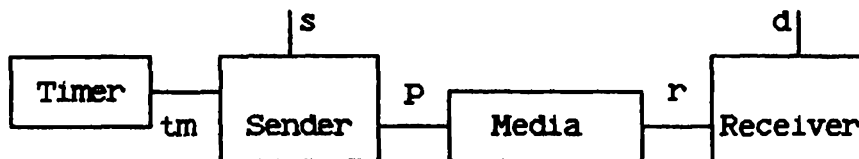


Fig. 5. Architecture of AB-protocol.

Let us illustrate this approach by well-known alternating bit protocol¹⁵ (AB-protocol, for short) with slight modifications. The architecture of AB-protocol is depicted in the Fig. 5. The AB-protocol

consists of four logical modules - sender, receiver, media and timer. AB-protocol should read data blocks from the user via the gate *s* and deliver them to the remote user via the gate *d*. A media can either transmit data block or lose some of them. To correct the errors AB-protocol uses a retransmission technique with the aid of timeout mechanism.

Thus we can conceive the AB-protocol as the parallel composition of four processes:

$Ab\_protocol[s, d] = \text{hide } p, r, tm \text{ in}$

$(Timer[tm] \mid [tm] \mid Sender[s, p, tm] \mid [p] \mid Media[p, r] \mid [r] \mid Receiver[r, d])$

Due to the space limitation we consider only sender and timer modules in details. Below LOTOS-specifications of the processes are presented.

```

process Sender[s, p, tm] (n:bool):noexit :=
  s?dat:msg; p!DT!n!dat;
  tm!start; Wait_Ack[p, tm] (n, dat)
  where
    process Wait_Ack[s, p, tm] (n:bool, dat:msg):noexit :=
      tm!expired; p!DT!n!dat; tm!start; Wait_Ack[s, p, tm] (n, dat)
    [] p!ACK!n; Wait_Ack[s, p, tm] (n, dat)
    [] p!ACK!n; tm!reset; Sender[s, p, tm] ( $\hat{n}$ )
    endproc (* Wait_Ack *)
  endproc (* Sender *)
process Timer[tm]:noexit :=
  tm!start; 1;
  ( tm!expired; Timer[tm]
    [] tm!reset; Timer[tm] )
endproc (* Timer *)

```

Sender is defined with the aid of the auxiliary process Wait_Ack in order to specify the part with a cyclic behaviour. Process Sender behaves as follows. First Sender can read data (of sort msg) from the user via gate *s*. After that it sends the protocol command "DT" with these data and sequence number *n* via the gate *p*. After that Sender turns on a timeout and wait for acknowledgement (instantiates the process Wait_Ack). If it receives an acknowledgment with the same sequence number via gate *p*, then it turns off timeout and comes back to initial state with another sequence number ($\hat{n}$). If timeout is expired then process retransmits previous protocol command and again starts a timeout.

A process Timer has only one gate *tm* and behaves as follows. Initially, it can be turned on by action *tm!start*. After some interval (internal action 1) it can be turned off by either the command *tm!reset* or expiring of the timeout.

One can easily convince that the entity of Fig. 6 does represent the LOTOS-specification of Sender (up to parameters). A representation of timer has been depicted in Fig. 3.

6. Discussion and concluding remarks.

One can observe that each transition of resulting Petri net entity we obtained from LOTOS-specification in previous section is visible at only one access point. This is due to that the LOTOS does not allow to

specify simultaneous executions of a number of actions at the different access points. Thus we loose some benefits of our approach. One way to avoid this is to add to LOTOS a new operation of simultaneous execution likewise in Milner's SOCS¹³. For this operation we will use the sign "|". Returning to example of AB-protocol, it seems reasonable to think of execution a sequence $s?dat:msg; p!DT!n!dat; tm!start$ as an execution of a single event $(s?dat:msg|p!DT!n!dat|tm!start)$. Now we can rewrite a sender specification:

```

process Sender[s, p, tm] (n:bool):noexit:=
  (s?dat:msg|p!DT!n!dat|tm!start); Wait_Ack[p, tm] (n, dat)
  where
    process Wait_Ack[s, p, tm] (n:bool, dat:msg):noexit:=
      tm!expired; (p!DT!n!dat|tm!start); Wait_Ack[s, p, tm] (n, dat)
    [] p!ACK!n; Wait_Ack[s, p, tm] (n, dat)
    [] (p!ACK!n|tm!reset); Sender[s, p, tm] (n)
    endproc (* Wait_Ack *)
  endproc (* Sender *)

```

One can convince that corresponded Petri net entity (up to parameters) is exactly the same as presented in Fig.2. Its composition with a timer entity is depicted in the Fig.4.

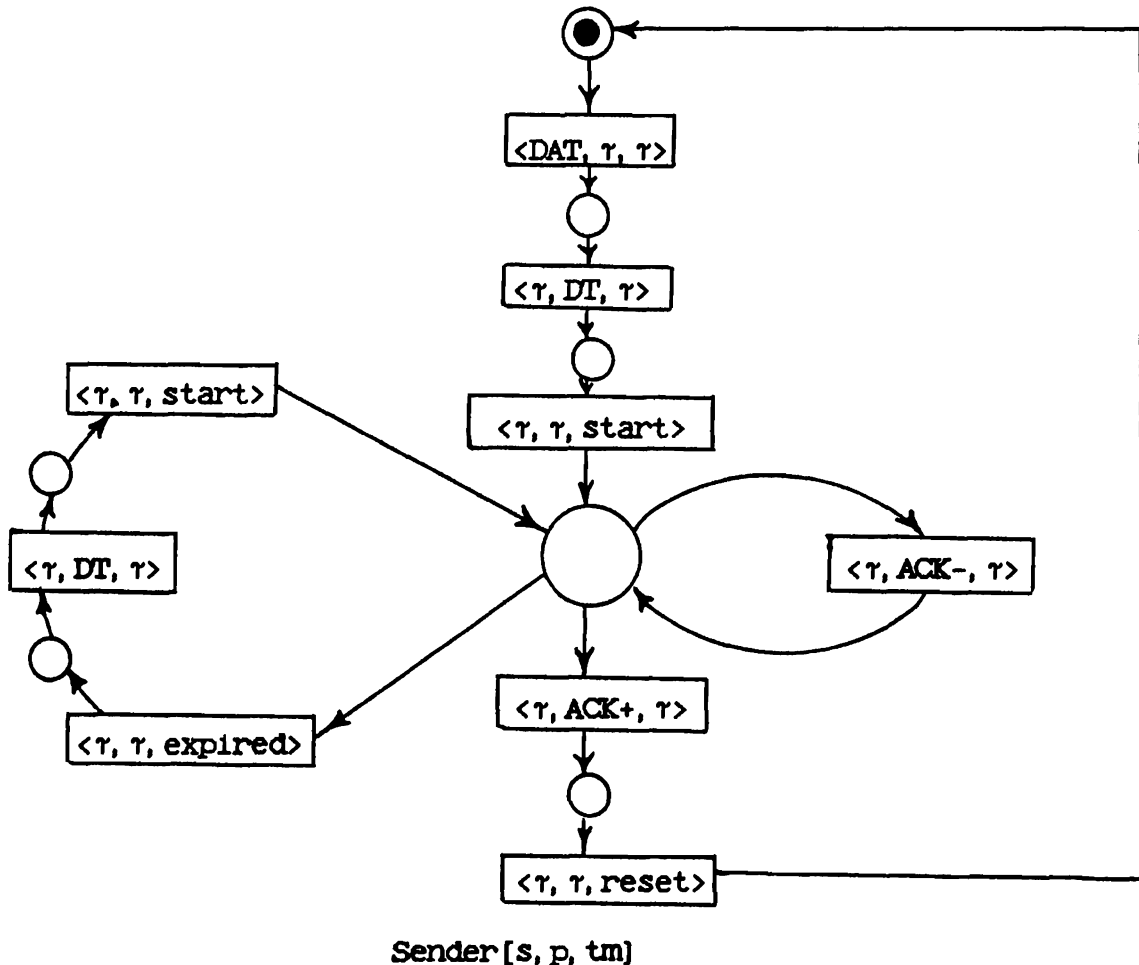


Fig. 6. Petri net entity of the Sender process.

In this paper we adopted some constraints on a specification style. In particular, we assumed that each transition had the following structure $g\alpha_1\alpha_2\ldots\alpha_n$ where g is a gate name, α_1 corresponds to an action name and $\alpha_2, \ldots, \alpha_n$ are parameters. But this constraint is very strong. There are a number of various specification styles¹⁶ that are not well-suited for a Petri net representation. Nevertheless, we believe that the idea of visualization of LOTOS specifications seems to be fruitful. From our opinion, one way to implement this idea is to develop a new specification style, which we will refer to as a visualization-oriented style, that should guarantee a successive translation LOTOS-specifications into Petri nets.

References*

1. N. A. Anisimov. Formal Model of LOTOS Based on Petri Net Theory, in *Proc of the Workshop on Computer Networks* (Alma-Ata, 1988) pp. 34-39 (in Russian).
2. N. A. Anisimov N. A. A Notion of Petri Net Entity for Communication Protocol Design. *Institute for Automation and Control Processes*, (Vladivostok, 1989).
3. B. Baumgarten. On Internal and External Characterization of PT-net Building Block Behaviour, *LNCS 340* (1989).
4. T. Bolognissi, Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Network and ISDN Systems 14* (1987) pp. 25-29.
5. W. Brauer, W. Reisig, G. Rozenberg (Eds) (1987), Petri Nets. Parts I and II. Proc. of an Advanced Course, *LNCS 254, 255* (1987).
6. L. A. Cherkasova, V. E. Kotov. Structured Nets. *LNCS 118* (1981).
7. P. H. J. Van Eijk, C. A. Vissers, M. Diaz M. (Eds) The Formal Description Technique LOTOS: Results of the ESPRIT/SEDOS Project, (1989).
8. C. A. R. Hoare. Communicating Sequential Processes. (Printice-Hall, 1985).
9. ISO 7498, Information Processing Systems - Open Systems Interconnection - Basic Reference Model (1983).
10. ISO 8807, Information Processing Systems - Open Systems Interconnection - "LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour" (1988).
11. S. Marchena, G. Leon. Transformation from LOTOS specifications to Galileo Nets. in: *Formal Description Techniques. Proc. of the 1st International Conference FORTE'88* (North-Holland, 1989) pp. 217-230
12. R. Milner. A Calculus for Communication Systems. *LNCS 92* (1980).
13. R. Milner. (1983) Calculi for Synchrony and Asynchrony. *Theoretical Computer Science 25* (1983) pp. 267-310.
14. E. -R. Olderog, U. Goltz, R. Van Glabbeek (Eds). Combining Compositionality and Concurrency - Summary of a GMD Workshop, Konigswinter, March 1988. *Arbeitspapiere der GMD 320* (1988).
15. A. S. Tanenbaum. Computer Networks. (Prentice-Hall, 1981).
16. C. Vissers, G. Scollo, M. Van Sinderen. Architecture and Specification Style in Formal Description of Distributed Systems. in: *Protocol Specification and Verification, VIII. Proc. of the IFIP WG 6.1 8th International Symposium*, Atlantic City (North-Holland, 1988).

* *LNCS* means *Lecture Notes in Computer Science*, Springer Verlag.

OPTIMAL CIRCULANT COMPUTER NETWORKS

E.A.MONAKHOVA

*Computing Center, Siberian Division of the USSR
Academy of Science, 630090, Novosibirsk, USSR*

ABSTRACT

In this paper a class of n -dimensional regular graphs called D_n -graphs or circulants is considered. Such graphs provide a practical interest as reliable interconnection networks for the multimodule supercomputer systems especially for the multitransputer systems. A solution of the problem of determining the best topologies to minimize the diameter (and, hence, the transmission delay) of the network is considered. A brief survey of the works of Russian researchers studying the circulants and several new results on the analytical representation of optimal circulants and on existence of optimal two-dimensional loop networks are presented.

1. Introduction

In this work we consider an important class of regular graphs known as D_n -(KAIS-)graphs¹⁻², circulant (cyclic) networks³⁻⁵, undirected star-polygon graphs⁶. Particular cases of these graphs are realized in ICL DAP, ILLIAC IV, MINIMAXS⁷, MICROS⁸ and are also considered in projects BSP, MPP, DIRMU, etc. Such interconnection network has been used for the organization of the high-speed multimodule memory in computer systems⁹. Recently the interest to the circulants has again animated due to appearance of the transputers^{4, 10-13}. Generalization of the considered class of networks was proposed by Monakhov²⁴.

In the general case a circulant (D_n -graph) is defined as graph $G(N; s_1, s_2, \dots, s_n)$ with N nodes labeled as $0, 1, 2, \dots, N-1$ having $i \pm s_1, i \pm s_2, \dots, i \pm s_n \pmod{N}$ nodes adjacent to each node i . The sequence $S = (s_i) \ (0 < s_1 < s_2 < \dots < s_n < (N+1)/2)$ is called the jump sequence. Degree $2n$ (n -dimension) of a node in an undirected graph G

is the number of edges incident to it.

The graph $G(N;1,s_2,\dots,s_n)$ when $s_1=1$ is particular case of the circulants. Such circulants known as loop networks have been studied in ⁹⁻¹¹. Fig. 1 illustrates the circulant graphs $G(15;3,5)$ with 15 nodes and $G(16;1,4)$ with 16 nodes and degree 4.

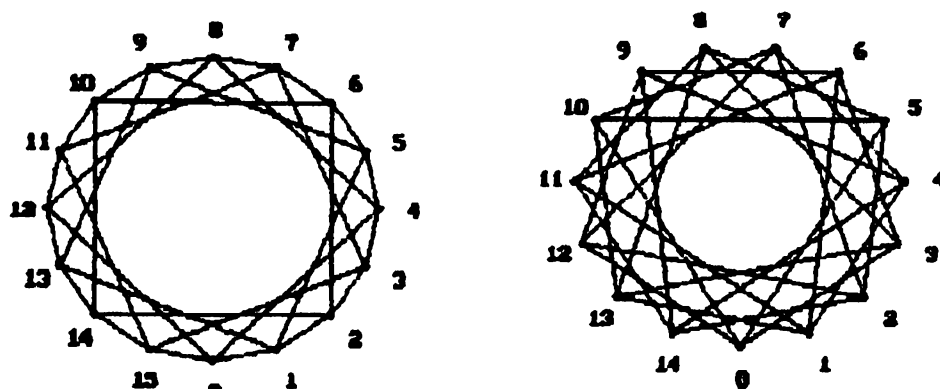


Figure 1

Synthesis of the optimal circulants is a certain problem of graph theoretical optimization representing the generalization (in the class of circulants) of the (d,k) -graph problem ¹⁴. This problem is in the search for a graph with the minimum of the diameter and the average diameter and maximum of the reliability and connectivity among all circulants having N nodes and dimension n (let this set be $T(N,n)$). The diameter of G is defined by $d = \max_{i,j} d_{ij}$, where d_{ij} is the length of the shortest path from the node i to the node j . The average diameter of G is defined as $\bar{d} = \sum_{i,j} d_{ij} / N^2$. Let $d(N) = \min_S \{d(G(N;S))\}$. Let for any graph $v \in T(N,n)$ $K_{n,m}$ denotes the number of nodes to be attained from node 0 by using at most m jump sizes and let $K_{n,m}^*$ be the upper bound for $K_{n,m}$. And let $L_{n,m} = K_{n,m} - K_{n,m-1}$, $L_{n,m}^*$ be the upper bound for $L_{n,m}$ on set $T(N,n)$. The values of $K_{n,m}^*$, $L_{n,m}^*$ for any n,m were determined ^{1,15} (see also ¹⁶). For $n=2$ $K_{2,m}^* = 2m^2 + 2m + 1$, $L_{2,m}^* = 4m$.

The graphs achieving the upper bounds are called the optimal graphs, namely, a graph $v \in T(N,n)$ is optimal if $L_{n,m} = L_{n,m}^*$ for any

$0 \leq m \leq d^* - 1$ and $L_{n,d^*} = N - K_{n,d^*}^* - 1$, where an optimal diameter d^* is given from the correlation $K_{n,d^*}^* - 1 < N \leq K_{n,d^*}^*$. As this is shown¹ the optimal graph has a minimum d and $\bar{d}$ and a maximum of the reliability and connectivity among all the graphs from $T(N,n)$. A graph G is nearly optimal if $d(G) = d(N)$.

The diameters of the optimal circulants are computed from the expression for $K_{n,m}^*$. The general dependence is in that a diameter is proportional to $\sqrt[n]{N}$. Wong and Coppersmith⁹ gave the lower bound for $d(N)$ in the case of $n=2$ $(\sqrt{2N}-3)/2$. In the case of $n=2$ the exact lower bound for $d(N)$ should be $d^*(N) = \lceil (\sqrt{2N-1}-1)/2 \rceil$, where $\lceil x \rceil$ denotes the smallest integer greater than or equal to x . This lower bound was pointed out first by Monakhova¹⁷, and later was also indicated by Boesch and Wang⁴.

2. Catalogue of optimal circulants

On the basis of the carried out investigations of circulants properties the reduced search algorithm was given in¹⁸ and this method has been applied for obtaining a catalogue of optimal circulants. The optimal graphs having $n=2,3,4,5$ and $N \leq 256$ were obtained. Fragments of the catalogue are shown in Tables 1,2. In Table 1 the optimal graphs are of degree 4, in Table 2 they are of degree 6. Different unequivalent descriptions are given for every N (see part 4). The complete catalogue is adduced in^{18,19}.

3. Analytical representation of optimal two- dimensional circulants

For the optimal circulant graphs with $n=2$ the author first found the formulas for the determining the values of s_0 and s_1 and diameters of such graphs^{17,20}.

Theorem 1. For any natural $N > 4$ the optimal circulant $G(N; s_0, s_1)$ exists and has a description in the form of

$$G(N; s, s+1), \text{ where } s = \lceil (\sqrt{2N-1}-1)/2 \rceil \quad (1)$$

The diameter of this graph is $d^* = \lceil (\sqrt{2N-1}-1)/2 \rceil$, where $\lfloor x \rfloor$ is the nearest integer to x .

Table 1

N	s_0	s_1	d
16	1	6	3
32	1	7	4
	1	12	
48	1	20	5
	2	9	
96	1	42	7
	3	20	

Table 2

N	s_0	s_1	s_2	d
16	1	2	6	2
	1	2	7	
	1	3	4	
	1	4	6	
	1	4	7	
32	1	6	7	
	32	1	4	
	1	4	10	
	1	4	13	
	1	6	9	
48	1	7	18	3
	1	8	11	
	1	9	14	
	2	8	9	
	2	8	9	
96	1	6	37	4
	1	8	42	
	1	11	18	
	1	12	17	
	1	27	38	

Routing and broadcasting algorithms for the optimal circulants of description in the form of $G(N; s, s+1)$ from Theorem 1 can be found in ²¹. There are shown that these algorithms have the constant complexity $O(C)$ independent of N .

In ¹⁸ it was proved the following theorems dealt with graphs of the form of $G(N; 1, s_2, \dots, s_n)$ known as loop networks.

Theorem 2. Let $G(N; 1, s_2, \dots, s_n)$ be the optimal circulant with N nodes. Then for all $i = \overline{2, n}$ $s_i \geq 2(d^*-1)$ where d^* is the optimal diameter.

Theorem 3. For any natural $m > 1$ the optimal circulants of $n=2$ have the descriptions in the form of $G(N; 1, 2m+1)$ for $N = 2m^2 + 2m + 1$ and in the

form of $G(N;1,2m-1)$ for $N=2m^2+2m-1$.

In addition to 16 infinite classes of values N for which optimal circulants can be found pointed out in ¹¹, we give some new such classes. It is not difficult to prove the following two theorems.

Theorem 4. Let $N=2d^2+d-2$ or $N=2d^2+d-3$ where $d>1$. Then, the network $G(N;1,2d-2)$ is optimal. Let $N=2d^2+d-5$, where $d>2$. Then, the network $G(N;1,2d+4)$ is optimal.

Theorem 5. The maximum value for N for which $G(N;1,s)$ for any $s \geq 2$ is optimal, equals

$$N_{\max} = \begin{cases} 9 & \text{if } s=2 \\ (s+1)^2/2+s & \text{if } s \text{ is odd} \\ (s+2)^2/2+s/2-1 & \text{if } s>2 \text{ is even} \end{cases}$$

4. Classes of equivalence of optimal circulants

In this part the problem is investigated how one may get other descriptions of a circulant if one of its descriptions is known.

Let us have some description of the circulant graph $G(N;S)=G(N;s_1,s_2,\dots,s_n)$. Multiply all s_i by the element t of leaden system of deductions (mod N) ²². As a new s_i' takes the residues of the division ts_i to N , if they are less than $[N/2]$ or take the additions of these residues to N , if they are greater than $[N/2]$.

The given transformation transferring all s_i into s_i' is called ³the equivalent transformation and the relation between sets S and S' and also between the graphs $G(N;S)$ and $G(N;S')$ - the relation of equivalence. If now t runs the leaden system of deductions (mod N) then all graphs of the form $G(N;S')$ given from $G(N;S)$ form the class of equivalence. Tables 1 and 2 adduce the values s_i for different classes of equivalence for the given N .

From the equivalence of graphs $G(N;S)$ and $G(N;S')$ follows their isomorphism ³. The reverse property, generally, does not hold. For example, $G(16;1,2,7)$ and $G(16;2,3,5)$ are isomorphic but unequivalent. For the circulants with the prime number of nodes the notions of equivalence and isomorphism coincide ⁶.

Taking into consideration the above-mentioned one may determine the value of s for the optimal loop networks of the form $G(N;1,s)$ using (1).

For example, the optimal graph with $N=36$ has $d^*=4$ and $s_0=4$, $s_1=5$. The leaden system of deductions (mod 36) is 1,5,7,11,13,17. Transferring the given jumps 4,5 into s_1', s_2' gives a new equivalent optimal graph $G(36;1,8)$.

But for some N 's this method cannot be used since the optimal graphs for description of the form $G(N;1,s)$ do not exist. Let $Q(N)$ denote the number of classes of equivalence of optimal two-dimensional circulants for the given N . We offer two open questions for further investigation: 1) obtain $Q(N)$ for any N ; 2) classify all N 's for which $Q(N)=1$. A solution of these questions gives the key to a problem of existence of optimal two-dimensional loop networks. Analysis of the catalogue of optimal circulants¹⁹ shows that $Q(N)$ has a tendency to regularity. From 16 infinite classes of the values N ¹¹ three classes, namely, $N=2t^2+3t+1$; $2t^2+4t+2$; $2t^2+5t+3$, have increasing regularity $Q(N)$ and a number of other classes of values N have $Q(N)=1$.

5. On existence of two-dimensional optimal loop networks

In¹¹ the problem of classification of those N 's for which two-dimensional optimal loop networks can be found is proposed. The results of theorem 1 and a notion of equivalence allow us to find some necessary and sufficient conditions of existence of the optimal loop networks. The proofs of theorems 6-9 do not adduce with the aim to reduce the size of this paper. Theorem 1 gives the following property: any positive integer $N>3$ can be represented in one of the intervals

$$2d^2+2 \leq N \leq 2(d+1)^2+1, \text{ where } d \text{ is a positive integer.} \quad (2)$$

Theorem 6. For any N from interval (2) for which $\gcd(N,d)=1$ or $\gcd(N,d+1)=1$ the optimal loop network of the form $G(N;1,s)$ exists.

Theorem 7. For any N from interval (2) for which $\gcd(N,d) \neq 1$, $\gcd(N,d+1) \neq 1$ and $Q(N)=1$, the optimal loop network of the form $G(N;1,s)$ does not exist.

Theorem 8. For $N=2d^2+2d$, where d is any positive integer, the optimal loop network of the form $G(N;1,s)$ does not exist. For $N=2d^2-2$, where d is an odd integer, the optimal loop network of the form $G(N;1,s)$ exists but if d is an even integer then such a graph does not exist.

Theorem 9. Let $d>2$ be a prime integer. Then for all $2(d-1)^2+2 \leq N \leq 2(d+1)^2+1$ (except $N=2d^2-2d; 2d^2+2d; 2d^2+4d$) the optimal loop network of the form $G(N;1,s)$ exists.

6. On existence of multidimensional optimal circulants

To generalize to the case of arbitrary n and N the analytical method of obtaining the optimal graphs is not completely applied so far, as it was shown in ¹⁸, the optimal circulants exist not for all values of N and n . For example, the optimal circulants with $n=3;4;5$ and $N \leq 100$ do not exist for $N=22-26, 51-54, 56-69; 34, 36-45, 47; 50, 52-68, 72-73$. But nearly optimal solutions exist for these cases.

In the field of synthesis of the optimal circulants the following questions remain unsolved: getting the necessary and sufficient conditions of existence of the optimal circulants of $n>2$ and revealing the classes of N and n for which it is possible to get optimal descriptions by means of analytical formulas.

For the general case of n the analytical solution is given ²⁰ for the circulants with diameter equal to 2.

Theorem 10. For any natural $n>1$ the optimal circulants with $2n+1 \leq N \leq 4n+1; 4n-1 \leq N \leq 8n-3; 6n-7 \leq N \leq 12n-15$ exist and have accordingly the following descriptions $G(N;1,2,\dots,n); G(N;n,n+1,\dots,2n-1); G(N;n-1,2n-2, 2n-1, \dots,3n-4)$. For any natural $n>3$ the optimal circulants with $8n-13 \leq N \leq 16n-29$ (except $N=16n-30$) exist and have the descriptions $G(N;4,8,\dots, 4n-12,4n-10,4n-9,4n-7)$.

The obtained optimal descriptions of circulants of an arbitrary degree have an interest property to keep the values of jumps s_i on the large interval of a change N . For example, the optimal circulants of $n=7$ and $43 \leq N \leq 81, N=83$ have the same description of the form $G(N;4,8,12,16,18,19,21)$.

The analogous property takes place for optimal graphs of dimension 2 from Theorem 1. From this theorem it follows that for all $2s^2 - 1 \leq N \leq 2s^2 + 4s + 3$ ($s=1,2,3,\dots$), $G(N;s,s+1)$ is the optimal circulant. The given property does not hold, for example, for chordal ring networks²³ which are the subgraphs of circulants and also for the optimal circulants of the descriptions different from those considered.

7. References

1. V.V.Korneev, Macrostructure of homogeneous computer systems. *Vychislitelnye sistemy*, vyp.60, (Novosibirsk, 1974) pp.17-34
2. V.A.Vorobiev, Simplest structures of homogeneous computer systems. *Vychislitelnye sistemy*, vyp.60, (Novosibirsk, 1974,) pp.35-49
3. B.Elspas, J.Turner, Graphs with circulant adjacency matrices. *J. Combin. Theory*, 9 (1970) pp.297-307.
4. F.T.Boesch, J.-F.Wang, Reliable circulant networks with minimum transmission delay. *IEEE Trans. Circuits Syst.*, CAS-32 (1985) pp.1286-1291.
5. H.A.David, Enumeration of cyclic graphs and cyclic designs. *J. Combin. Theory*. 13 (1972) pp.303-308.
6. J.Turner, Point-symmetric graphs with a prime number of points. *J. Combin. Theory*. 3 (1967) pp.136-145.
7. N.N.Mirenkov, *Parallel programming for multimodule computer systems*. (Radio i svyas, Moscow, 1989) 320p.
8. V.G.Khoroshevsky, V.V.Korneev et al. Computer system with programmable structure MICROS- tool for industrial control. In *VIII International Conference COMCONTROL '87*. (Moscow,1987) pp.174-177.
9. C.K.Wong, Don Coppersmith, A combinatorial problem related to multimodule memory organization. *J.Assoc. Comput. Mach.* 21 (1974) pp.392-402.
10. R.F.Browne, R.M.Hodgson, Symmetric degree-four chordal ring networks. *IEEE Proceedings*, 137-E (1990) pp.310-318

11. D.-Z.Du et al. A combinatorial problem related to distributed loop networks. *Networks*, 20 (1990) pp.173-180.
12. M.-S.Chen, K.G.Shin, D.D.Kandlur, Addressing, routing and broadcasting in hexagonal mesh multiprocessors. *IEEE Trans. on computers*, 39 (1990) pp.10-18.
13. M.Schinder, New architectures keep pace with throughput needs. *Electronic Design*. No5 (1981) pp.97-106.
14. G.Memmi, Y.Raillard, Some new results about the (d,k) graph problem. *IEEE Trans. Comput.* 31 (1982) pp.784-791.
15. C.K.Wong, T.W.Maddocks, A generalized Pascal's triangle. *Fibonacci Quart.*, 13 (1975) pp.134-136.
16. V.V.Korneev, *Architecture of computer systems with programmable structure*. (Nauka, Novosibirsk, 1985) 166p.
17. E.A.Monakhova, Analytical representation of optimal two-dimensional D_n -structures of homogeneous computer systems. *Vychislitelnye sistemy*, vyp.90, (Novosibirsk, 1981) pp.81-91.
18. E.A.Monakhova, Synthesis of optimal D_n -structures. *Vychislitelnye sistemy*, vyp.80, (Novosibirsk, 1979) pp.18-35.
19. E.A.Monakhova, Organization of macrostructures and intercomputer interactions in computer systems with programmable structure. *Ph.D. thesis*, (Leningrad Electrotech. Institute, 1988).
20. E.A.Monakhova, Optimal KAIS-structures of in-plant homogeneous computer systems. *Elektron. modelirovanie*, No3, (1985) pp.30-34.
21. E.A.Monakhova, Algorithms of interprocessor exchanges and reconfiguration of interconnection networks in computer systems with programmable structure. *Vychislitelnye sistemy*, vyp.94, (Novosibirsk, 1982) pp.81-102.
22. I.M.Vinogradov, *Principles of theory of numbers*. (Nauka, Moscow, 1972) 167p.
23. B.Arden, H.Lee, Analysis of chordal ring networks. *IEEE Trans. Comput.* 30 (1981) pp.291-295.
24. O.G.Monakhov, Parametric description of structures of homogeneous computer systems. *Vychislitelnye sistemy*, vyp.80, (Novosibirsk, 1979) pp.3-17.

Parallel isotropic access to tree-like data structures

Reiner Creutzburg

University of Karlsruhe
 Faculty of Informatics
 Institute of Algorithms
 and Cognitive Systems
 P. O. Box 6980
 D-7500 Karlsruhe 1
 Federal Republic of Germany
 (phone: +49-721-608 4325)
 (fax: +49-721-696893)

Lutz Andrews

Deutsches Textilforschungszentrum NW
 Frankenring 2
 D-4150 Krefeld 1
 Federal Republic of Germany

Abstract

This note deals with recent results of a series of papers on the problem of parallel access to trees and is motivated by the research work on parallel memory design for trees. Known results on parallel access to complete q -ary subtrees and extended q -ary subtrees are summarized. Then, new results on the parallel access to other tree-like data structures are presented.

1 Introduction

One of the most significant challenges in parallel computers is the parallel memory organization. In order to obtain great performance, data in parallel memory have to be accessed at the highest throughput. Therefore data must be processed in parallel.

Classical SIMD computers perform vector processing [3-4,15,19-21,25,30]. A lot of research work has been done in designing parallel memories to access arrays [2-4,10-11,13-15,19-26,28,30].

In general parallel memories are designed for obtaining conflict-free access to arrangements of cells that belong to a specified set of data templates. In case of a single template T the smallest number of memory modules needed obviously is $N = |T|$ (the size of T), but more modules may be required. In general the problem arises of finding the smallest number of memory modules for storing data so as to have conflict-free access for a set of templates of interest.

Trees are another important data structure in computer science [18]. It is an interesting problem to design parallel memories to access trees [1, 2, 5-9, 11-13, 15, 22, 23, 27, 29, 30]. We choose the terminology given in [17] to describe a tree. Consider a labelled q -ary tree such that the immediate successors of the node x are $qx + 1, qx + 2, \dots, qx + q$, and the label of the root is 0. The level of a node is defined by initially letting the root be at level 1. The level of every node is one more than the level of its immediate predecessor. The height t of a tree is defined as the maximum level of any node in the tree.

Parallel access to a subtree means the conflict-free access to all the nodes of the subtree. Whenever a set of data items must be retrieved but is not available from distinct memory modules and hence cannot be retrieved in one cycle we speak of a "conflict".

We will always consider the case of **isotropic** storage of the nodes of a tree-like data structure. We remark that there also exists semi-isotropic and non-isotropic storage schemes [12-15].

In recent papers the parallel access to

- complete binary subtrees [1,5,6,11-13,15,22,23,27,29,30],
- complete q -ary subtrees [1,5,6,15,30],

was investigated.

For possible application in parallel logic programming the complete extended binary subtrees are of particular interest. We studied

- complete left-extended binary trees,
- complete right-extended binary trees,
- complete general-extended binary trees,

in detail [8,9]. Then, the results were generalized in [1,7] for the case of

- complete left-extended q -ary trees,
- complete right-extended q -ary trees,
- complete general-extended q -ary trees,

and are described in detail in [1].

The aim of this paper is to summarize and classify known results and to present new results for other tree-like data structures.

2 Module assignment functions

If we speak about parallel storage of general data structures and parallel access the module assignment functions and address functions play an important role. In [15] module assignment functions and address functions for general data structures are described. We will concentrate here only on module assignment functions. We consider now module assignment functions for q -ary trees.

A module assignment function S for q -ary trees is a mapping from the set of the labels of a complete q -ary tree to the N memory modules. We denote the set of indices of memory modules by

$$E_N = \{0, 1, \dots, N - 1\}.$$

In the following we will always consider the case of **isotropic** storage [15] to tree-like data structures. This assumption means if two arbitrary nodes x_1 and x_2 are stored in the same memory module

$$S(x_1) = S(x_2) \bmod N$$

then their successor nodes

$$qx_1 + 1, qx_1 + 2, \dots, qx_1 + q; \quad qx_2 + 1, qx_2 + 2, \dots, qx_2 + q,$$

respectively, are stored in the same memory modules

$$S(qx_1 + 1) = S(qx_2 + 1); S(qx_1 + 2) = S(qx_2 + 2); \dots;$$

$$S(qx_1 + q) = S(qx_2 + q) \bmod N,$$

respectively.

A module assignment function is called **conflict-free** with respect to a tree, if for two nodes x_1 and x_2 ($x_1 \neq x_2$)

$$S(x_1) \neq S(x_2) \bmod N.$$

2.1 Recursively linear module assignment functions

A special recursively linear module assignment function S for binary trees was introduced in [27] by the following recurrence equations

$$\left. \begin{aligned} S(0) &= 0 \\ S(2x + 1) &= aS(x) + 1 \\ S(2x + 2) &= aS(x) + 2 \end{aligned} \right\} \bmod N \quad (1)$$

where a is an integer with $0 < a < N$.

A generalized form of (2.1) for q -ary trees was given in [5,6]

$$\left. \begin{aligned} S(0) &= 0 \\ S(qx + 1) &= aS(x) + 1 \\ S(qx + 2) &= aS(x) + 2 \\ \dots &\dots \\ S(qx + q) &= aS(x) + q \end{aligned} \right\} \bmod N \quad (2)$$

where a is an integer with $0 < a < N$.

The general recursively linear module assignment function S was introduced in [5,6,15] and has the following form:

$$\left. \begin{aligned} S(0) &= 0 \\ S(qx + 1) &= a_1S(x) + b_1 \\ S(qx + 2) &= a_2S(x) + b_2 \\ \dots &\dots \\ S(qx + q) &= a_qS(x) + b_q \end{aligned} \right\} \bmod N \quad (3)$$

where a_i and b_i are integers with $0 < a_i, b_i < N$ for $i = 1, \dots, q$.

3 Known results

In [1,5-7,15] the following results for isotropic subtree access are given and proved:

Theorem 1 *A parallel conflict-free access to all the nodes of an arbitrary complete q -ary subtree of height t of a q -ary tree ($q > 1, t > 1$) with a special recursively linear module assignment function S is possible, if the number of memory modules N is equal to*

$$N = q^{t-2}(q(t-1) + 1).$$

A minimization of N according to (3) is possible. Several examples are given in [1,15].

Theorem 2 *The minimal number N of needed memory modules for the parallel conflict-free access to general extended q -ary subtrees of height t is equal to*

$$N = qt.$$

The minimal number N of needed memory modules for the parallel conflict-free access to complete left- or right extended q -ary subtrees is equal to

$$N = (q-1)t + 1.$$

The related module assignment functions are given in [1,7,15].

Theorem 3 *A parallel conflict-free access to all t layers of a complete q -ary subtree of height t ($t > 0$) of a complete q -ary ($q > 0$) tree is possible with*

$$N = q^{t-1} \tag{4}$$

memory modules using a special recursively linear module assignment function S . The number N given in (4) is minimal.

Theorem 4 *A parallel conflict-free access to all the nodes of an arbitrary layer of the length l ($l > 0$) and to a layer of length smaller than l in each level h ($h > 0$) of a complete q -ary subtree ($q > 0$) is possible with*

$$N = l$$

memory modules using a special recursively linear module assignment function S .

4 New results

In several previous papers [1,5-9,11-13,15] we considered the parallel access to complete q -ary subtrees of height t and to complete general extended q -ary subtrees of height t of a complete q -ary tree.

Now we deal with the simultaneous parallel conflict-free access to complete q -ary subtrees and to complete left-, right- or general extended q -ary subtrees of a complete q -ary tree with N memory modules by using the same module assignment function [1].

Furthermore, we deal with the access to complete q -ary subtrees of height $t - 1$ and to the last level t of a complete q -ary subtree of height t .

4.1 Simultaneous parallel access to complete q -ary subtrees and to complete-extended q -ary subtrees

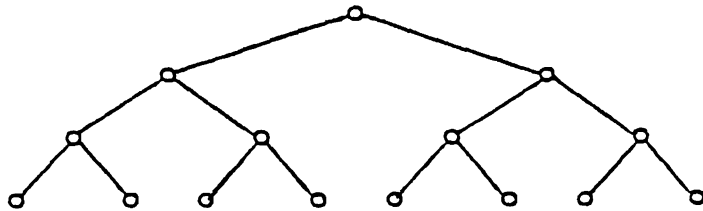
We use the following notations in this chapter

t_t height of a complete q -ary subtree,

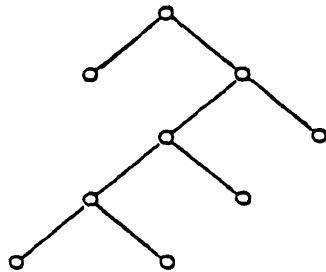
t_c height of a complete general extended q -ary subtree,

t_{lc} height of a complete left extended q -ary subtree,

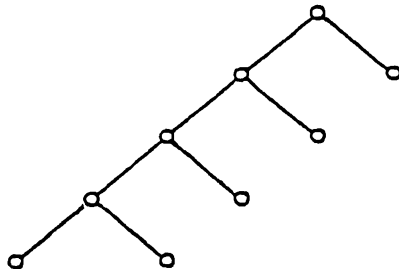
as shown in the next figures.



$t_t = 4$ height of a complete binary tree



$t_c = 5$ height of a complete general extended binary tree



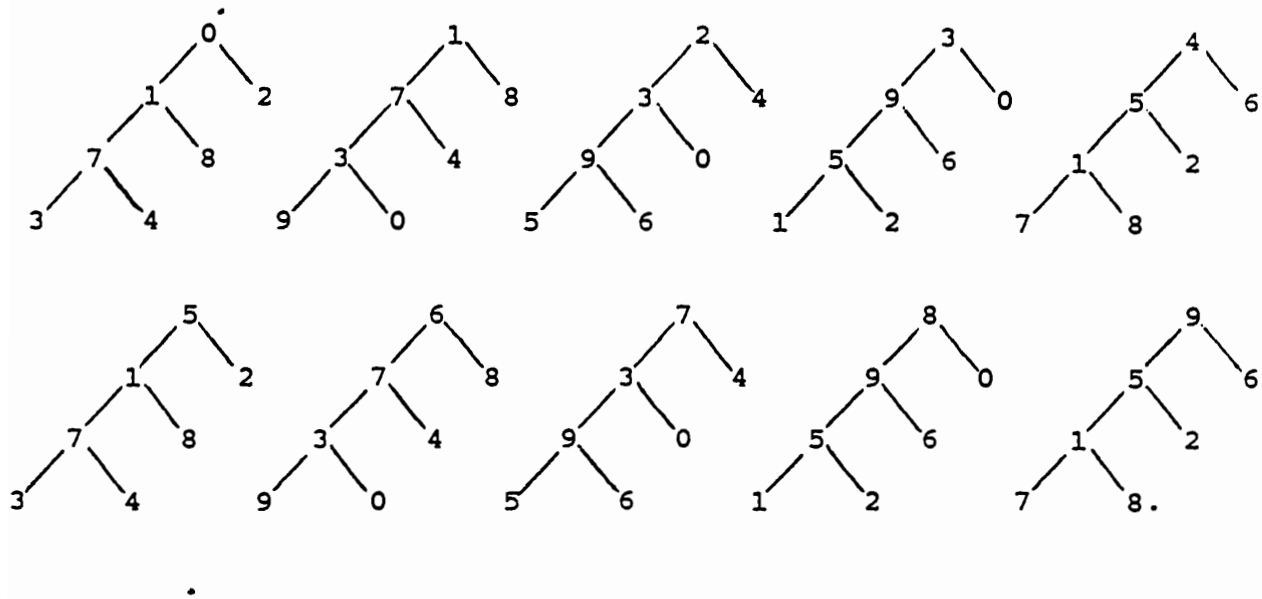
$t_{lc} = 5$ height of a complete left extended binary tree

The following example is illustrative.

Example 1 We consider the access to a complete binary tree of height $t_t = 3$ consisting of 7 nodes with $N = 10$ memory modules. We use the memory module assignment function given already in [27]:

$$\left. \begin{aligned} S(0) &= 0 \\ S(2x+1) &= 6S(x)+1 \\ S(2x+2) &= 6S(x)+2 \end{aligned} \right\} \text{mod } 10. \quad (5)$$

From Theorem 1 it is known that a conflict-free access to the 7 nodes of all 10 subtrees of height $t_t = 3$ is possible. Furthermore, this function allows the conflict-free parallel access to all the 7 nodes of an arbitrary left extended binary subtree of height $t_{lc} = 4$. The relating 10 subtrees are shown in the following figure



Now we generalize the results of Theorems 1-2 in the following Theorem [1].

Theorem 5 A parallel conflict-free access to all the $(q^{t_t} - 1)/(q - 1)$ nodes of an arbitrary complete q -ary subtree of height t_t of a complete q -ary tree ($q > 0$, $t_t > 1$), to all the

$q(t_c - 1) + 1$ nodes of an arbitrary complete general extended q -ary subtree of height $t_c = t_t + p + 1$

and to all the

$q(t_{lc} - 1) + 1$ nodes of an arbitrary complete left extended q -ary subtree of height $t_{lc} = q(t_t + p - 1) + 2$

with a special recursively linear module assignment function S according to (2) is possible, if the number of memory modules N in (2) is equal to

$$N_p = q^{t_t-2}(q(t_t + p) + 1) \quad (6)$$

with $p \in \{-1, 0, 1, 2, 3, \dots\}$ and if the parameter a in (2) satisfies the condition

$$a = \begin{cases} \begin{cases} 1 \pmod{q(2+p)+1} & \text{in the case } t_t = 2, \\ 0 \pmod{q} & \end{cases} \\ \begin{cases} 1 \pmod{q(3+p)+1} & \text{in the case } t_t = 3, \\ q \pmod{q^{t_t-2}} & \end{cases} \\ 1 \pmod{q(t_t+p)+1} & \text{in the case } t_t > 3. \end{cases} \quad (7)$$

Remark 1 For $p = -1$ we obtain from (6) that

$$N_{-1} = q^{t_t-2}(q(t_t - 1) + 1).$$

This case is the evidence of Theorem 1 and we have $t_c = t_t$ and $t_{lc} = q(t_t - 2) + 2$.

The proof is given in [1].

The Table 1 lists a lot of special cases of Theorem 5 concerning parallel access to an arbitrary complete q -ary subtree of height t_t and to an arbitrary complete left- or general extended q -ary subtree of height t_{lc} and t_c , resp., of a complete q -ary tree [1].

4.2 Parallel access to a complete q -ary subtree of height $t-1$ and to the last layer of level t of a q -ary subtree of height t

Now we consider the parallel (linear) access to all nodes of a complete q -ary subtree of height $t - 1$ and to all nodes of the last layer t of a complete q -ary subtree of height t . Note that in this case we can access all nodes of an arbitrary q -ary subtree of height t in two steps. The number of needed memory modules for the conflict-free parallel access is (in general) smaller than the number of memory modules which we need in the case that we conflict-free access to all nodes of a complete q -ary subtree of height t in only one step. We mention that we can conflict-free access to all these nodes of a complete q -ary subtree of height t in two steps (see above) by using the same module assignment function.

We deal with the following example:

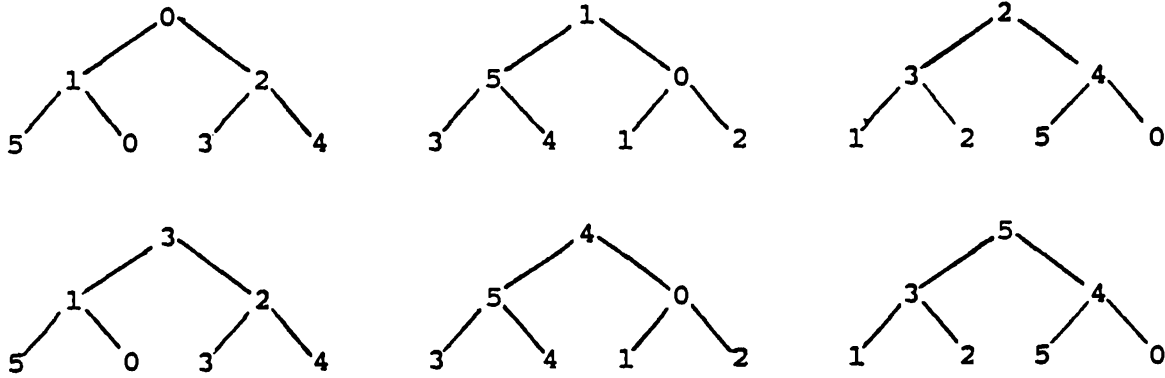
Example 2 Consider the access to the 7 nodes of the complete binary subtree of height $t = 3$ of the complete binary tree. We can conflict-free access to the 3 nodes of the complete binary subtree of height $t = 2$ and to the 4 nodes of the 3rd layer of the complete binary subtree of height $t = 3$ with $N = 6$ memory modules if we use the following module assignment function S :

$$\left. \begin{aligned} S(0) &= 0 \\ S(2x+1) &= 4S(x) + 1 \\ S(2x+2) &= 4S(x) + 2 \end{aligned} \right\} \pmod{6}. \quad (8)$$

q=2	q=3	q=4	q=5	q=6	q=7	q=8
7 nodes $N_{-1}=10$ $t_{1c}=4; t_c=3$ $N_0=14$ $t_{1c}=6; t_c=4$ $N_1=18$ $t_{1c}=8; t_c=5$ 15 nodes $N_{-1}=28$ $t_{1c}=6; t_c=4$ $N_0=36$ $t_{1c}=8; t_c=5$ $N_1=44$ $t_{1c}=10; t_c=6$ 31 nodes $N_{-1}=72$ $t_{1c}=8; t_c=5$ $N_0=88$ $t_{1c}=10; t_c=6$ $N_1=104$ $t_{1c}=12; t_c=7$	13 nodes $N_{-1}=21$ $t_{1c}=5; t_c=3$ $N_0=30$ $t_{1c}=8; t_c=4$ $N_1=39$ $t_{1c}=11; t_c=5$ 40 nodes $N_{-1}=90$ $t_{1c}=8; t_c=4$ $N_0=117$ $t_{1c}=11; t_c=5$ $N_1=144$ $t_{1c}=14; t_c=6$ 121 nodes $N_{-1}=351$ $t_{1c}=11; t_c=5$ $N_0=432$ $t_{1c}=14; t_c=6$ $N_1=513$ $t_{1c}=17; t_c=7$	21 nodes $N_{-1}=36$ $t_{1c}=6; t_c=3$ $N_0=52$ $t_{1c}=10; t_c=4$ $N_1=68$ $t_{1c}=14; t_c=5$ 85 nodes $N_{-1}=208$ $t_{1c}=10; t_c=4$ $N_0=272$ $t_{1c}=14; t_c=5$ $N_1=336$ $t_{1c}=18; t_c=6$ 341 nodes $N_{-1}=1088$ $t_{1c}=14; t_c=5$ $N_0=1344$ $t_{1c}=18; t_c=6$ $N_1=1600$ $t_{1c}=22; t_c=7$	31 nodes $N_{-1}=55$ $t_{1c}=7; t_c=3$ $N_0=80$ $t_{1c}=12; t_c=4$ $N_1=105$ $t_{1c}=17; t_c=5$ 156 nodes $N_{-1}=400$ $t_{1c}=12; t_c=4$ $N_0=525$ $t_{1c}=17; t_c=5$ $N_1=650$ $t_{1c}=22; t_c=6$ 781 nodes $N_{-1}=2625$ $t_{1c}=17; t_c=5$ $N_0=3250$ $t_{1c}=22; t_c=6$ $N_1=3875$ $t_{1c}=27; t_c=7$	43 nodes $N_{-1}=78$ $t_{1c}=8; t_c=3$ $N_0=114$ $t_{1c}=14; t_c=4$ $N_1=150$ $t_{1c}=20; t_c=5$ 259 nodes $N_{-1}=684$ $t_{1c}=14; t_c=4$ $N_0=900$ $t_{1c}=20; t_c=5$ $N_1=1116$ $t_{1c}=26; t_c=6$ 1555 nodes $N_{-1}=5400$ $t_{1c}=20; t_c=5$ $N_0=6696$ $t_{1c}=26; t_c=6$ $N_1=7992$ $t_{1c}=32; t_c=7$	57 nodes $N_{-1}=105$ $t_{1c}=9; t_c=3$ $N_0=154$ $t_{1c}=16; t_c=4$ $N_1=203$ $t_{1c}=23; t_c=5$ 400 nodes $N_{-1}=1078$ $t_{1c}=16; t_c=4$ $N_0=1421$ $t_{1c}=23; t_c=5$ $N_1=1764$ $t_{1c}=30; t_c=6$ 2801 nodes $N_{-1}=9947$ $t_{1c}=23; t_c=5$ $N_0=12348$ $t_{1c}=30; t_c=6$ $N_1=14749$ $t_{1c}=37; t_c=7$	73 nodes $N_{-1}=136$ $t_{1c}=10; t_c=3$ $N_0=200$ $t_{1c}=18; t_c=4$ $N_1=264$ $t_{1c}=26; t_c=5$ 585 nodes $N_{-1}=1600$ $t_{1c}=18; t_c=4$ $N_0=2112$ $t_{1c}=26; t_c=5$ $N_1=2624$ $t_{1c}=34; t_c=6$ 4681 nodes $N_{-1}=16896$ $t_{1c}=26; t_c=5$ $N_0=20992$ $t_{1c}=34; t_c=6$ $N_1=25088$ $t_{1c}=42; t_c=7$

Table 1. Some special cases concerning parallel access to an arbitrary complete q-ary tree of height t_t and to an arbitrary complete left- or general extended q-ary tree of heights t_{1c} and t_c , resp., of a complete q-ary tree that can be derived from Theorem 5.

The 6 relating subtrees are the following:



Remark 2 It is easy to show that the number $N = 6$ of needed memory modules for the conflict-free access to the 7 nodes of a complete binary subtree of height $t = 3$ in two steps (see above) is the minimal number of necessary memory modules not only for linear module assignment functions.

Theorem 6 In a complete q -ary subtree of height t ($t > 2$) of a complete q -ary tree ($q > 1$) the parallel access to all the $(q^{t-1} - 1)/(q - 1)$ nodes of a complete q -ary subtree of height $t - 1$ of the complete q -ary tree and to all the q^{t-1} nodes of the last layer t is possible with

$$N = q^{t-2}(q(t - 2) + 1) \quad (9)$$

memory modules using the recursively linear module assignment function S :

$$\left. \begin{array}{lcl} S(0) & = & 0 \\ S(qx + 1) & = & aS(x) + 1 \\ S(qx + 2) & = & aS(x) + 2 \\ \dots & & \dots \\ S(qx + q) & = & aS(x) + q \end{array} \right\} \text{mod } q^{t-2}(q(t - 2) + 1) \quad (10)$$

with

$$a = \begin{cases} 0 \text{ mod } q & \text{in the case } t = 3, \\ 1 \text{ mod } q + 1 \\ q \text{ mod } q^{t-2} \\ 1 \text{ mod } q(t - 2) + 1 \end{cases} \text{in the case } t > 3. \quad (11)$$

The proof is given in [1].

The Table 2 lists some special cases concerning parallel access that can be derived from Theorem 6.

With the help of a computer we found the minimal number of needed memory modules in some cases for binary trees. They are presented in the following Table 3. Note that these numbers of memory modules are minimal if we use recursively

linear module assignment functions of the type (3) for binary trees.

t	1-ary tree	binary tree	ternary tree	quad-tree	...	7-ary tree	oct-tree
3	3 nodes N = 2 modules	7 nodes N = 6 modules	13 nodes N = 12 modules	21 nodes N = 20 modules	...	57 nodes N = 56 modules	73 nodes N = 72 modules
4	4 nodes N = 3 modules	15 nodes N = 20 modules	40 nodes N = 63 modules	85 nodes N = 144 modules	...	400 nodes N = 735 modules	585 nodes N = 1088 modules
5	5 nodes N = 4 modules	31 nodes N = 56 modules	121 nodes N = 270 modules	341 nodes N = 832 modules	...	2801 nodes N = 7546 modules	
6	6 nodes N = 5 modules	63 nodes N = 144 modules	364 nodes N = 1053 modules		...		

Table 2. List of necessary numbers N of memory modules for parallel, recursively linear, complete q -ary subtree access of height $t - 1$ of a complete q -ary tree and to all nodes of the last level t of the complete q -ary subtree of height t of a complete q -ary tree that can be derived from Theorem 6.

t	3	4	5	6
q=2	7 nodes N = 6 modules	15 nodes N = 20 modules	31 nodes N = 42 modules	63 nodes N = 108 modules

Table 3.

References

- [1] ANDREWS, L.: *On parallel isotropic conflict-free access to tree-like data structures*. Diploma work, ZKI Berlin 1989.
- [2] ANISIFOROV, A.: *Parallel subtree access in dyadic memories*. Diss. (Leningrad 1989).
- [3] BATCHER, K. E.: *STARAN parallel processor system hardware*. In: Proc. Fall Joint Computer Conf. AFIPS Conf., AFIPS Press, 43, 1974, pp. 405-410.
- [4] BUDNIK, - KUCK, D. J.: *The organization and use of parallel memories*. IEEE Trans. Comput., C-20, 1971, pp. 1566-1569.
- [5] CREUTZBURG, R.: *Parallel optimal subtree access with recursively linear memory function*. In: PARCELLA'86 (Eds.: T. Legendi, D. Parkinson, R. Vollmar, G. Wolf) Akademie-Verlag: Berlin 1986, pp. 203-209.
- [6] CREUTZBURG, R.: *Parallel linear conflict-free subtree access*. Proc. Internat. Workshop Parallel Algorithms Architectures (Suhl 1987), (Eds.: A. Albrecht, H. Jung, K.

- Mehlhorn) Akademie-Verlag: Berlin 1987, pp. 89-96.
- [7] CREUTZBURG, R.: *Parallel conflict-free optimal access to complete extended q-ary trees*. In: PARCELLA'88 (Eds.: G. Wolf, T. Legendi, U. Schendel) Akademie-Verlag: Berlin 1988, pp. 248-255.
 - [8] CREUTZBURG, R.: *Parallel conflict-free access to extended binary trees*. Proc. IMYCS (Smolenice 1988), pp. 115 - 121.
 - [9] CREUTZBURG, R. - ANDREWS, L.: *Optimal parallel conflict-free access to extended binary trees*. In: Recent Issues in Pattern Analysis and Recognition (Eds.: V. Cantoni, R. Creutzburg, S. Levialdi, G. Wolf) LNCS 399, Springer: New York 1989, pp. 214-225.
 - [10] GÖSSEL, M. - REBEL, B.: *Parallel memory with recursive address computation*. In: Parallel Computing (Ed.: M. Feilmeier) Elsevier: London 1984, pp. 515-520.
 - [11] GÖSSEL, M. - REBEL, B.: *Data structures and parallel memory*. In: PARCELLA'86 (Eds.: T. Legendi, D. Parkinson, R. Vollmar, G. Wolf) Akademie-Verlag: Berlin 1986, pp. 49-60.
 - [12] GÖSSEL, M. - REBEL, B.: *Memories for parallel subtree access*. In: Parallel Algorithms and Architectures (Eds.: A. Albrecht, H. Jung, K. Mehlhorn) (Suhl 1987) Akademie-Verlag: Berlin 1987, pp. 122-130.
 - [13] GÖSSEL, M. - REBEL, B.: *Parallel memories for picture processing (in Russian)*. Invest. Akad. Nauk., Techniceskaja Kibernetika. No. 2 (1988), pp. 198-206.
 - [14] GÖSSEL, M. - REBEL, B.: *Parallel access to rectangles*. In: Recent Issues in Pattern Analysis and Recognition (Eds.: V. Cantoni, R. Creutzburg, S. Levialdi, G. Wolf) LNCS 399, Springer: New York 1989, pp. 201-213.
 - [15] GÖSSEL, M., REBEL, B., CREUTZBURG, R.: *Speicherarchitektur und Parallelzugriff*. Akademie-Verlag: Berlin 1989., also: *Memory Architecture and Parallel Access*. (English version in print)
 - [16] HOCKNEY, R. W. - JESSHOPE, C. R.: *Parallel Computers*. Hilger: Bristol, 1981.
 - [17] HOROWITZ, E. - SAHNI, S.: *Fundamentals of Data Structures*. Computer Science Press: Woodland Hills (Ca.) 1976.
 - [18] KNUTH, D. E.: *The Art of Computer Programming, Fundamental Algorithms*. Addison-Wesley: Reading (MA) 1968.
 - [19] KUCK, D. J. - STOKES, R. A.: *The BURROUGHS scientific processor*. IEEE Trans. Comput., C-31, 1982, pp. 363-376.
 - [20] LAWRIE, D. H.: *Access and alignment, in an array processor*. IEEE Trans. Comput., C-24, 1975, pp. 1145-1155.
 - [21] LAWRIE, D. H. - VORA, C. R.: *The prime memory system for array access*. IEEE Trans. Comput., C-31, 1982, pp. 435-442.

- [22] METLITZKI, E. A.: *Data structures and parallel memory organization based on dyadic storage schemes*. In: Recent Issues in Pattern Analysis and Recognition (Eds.: V. Cantoni, R. Creutzburg, S. Levialdi, G. Wolf) LNCS 399, Springer: New York 1989, pp. 189-200.
- [23] METLITZKI, E. A. - KAVERSNEV, V. V. : *Parallel Memory Systems (in Russian)*. Leningrad 1989
- [24] REBEL, B. - GÖSSEL, M.: *Ein paralleler Speicher*. Research Report, Central Institute Cybernetics and Information Processes. Berlin, November 1982.
- [25] REBEL, B. - GÖSSEL, M.: *On conflict-free parallel memory access*. Computers and Artificial Intelligence, 2, 1983, pp. 395-401.
- [26] SHAPIRO, H. D.: *Theoretical limitations on the use of parallel memories*. Univ. Illinois, Dept. Comp. Sci., Report No. 75-776, December 1975.
- [27] SHIRAKAWA, H.: *On a parallel memory to access trees*. In: Memoirs of Research Institute of Science and Engineering, Ritsumeikan Univ. Kyoto, Japan No.46 (1987) pp. 57 - 62. (as report 1984)
- [28] WIJSHOFF, H. A. G. - VAN LEEUWEN, J.: *The structure of periodic storage schemes for parallel memories*. IEEE Trans. Comput. C-34 (1985), pp. 501-505
- [29] WIJSHOFF, H. A. G.: *Storing trees into parallel memories*. In: Parallel Computing 85 (Eds.: M. Feilmeier, J. Joubert, U. Schendel) Elsevier: Amsterdam 1986, pp. 253-261.
- [30] WIJSHOFF, H. A. G.: *Data Organization In Parallel Computers*. Kluwer: Amsterdam 1989

PARALLEL ALGORITHM FOR MAPPING OF PROGRAM GRAPHS INTO PARALLEL COMPUTERS

O.G. MONAKHOV

*Computing Centre, Siberian Division of the USSR
Academy of Science, 630090, Novosibirsk, USSR*

ABSTRACT

A problem of mapping of the information graph of a complex algorithm into the graph of the interprocessor network of the parallel computer system is considered. The parallel distributed algorithm is presented for optimal or suboptimal solution of the mapping problem, the objective functions for mapping and necessary conditions of optimal mapping is developed.

1. Introduction

There is considered a problem of mapping of the information graph of a complex algorithm into the graph of the interprocessor network of the parallel computer system (CS), which consists of elementary processors (PE) with local memory, interconnected by a programmable communication network. The mapping problem¹⁻³ is known as NP-complete problem and it is reasonable to develop a parallel algorithm for solving the problem. In this paper the parallel distributed algorithm is presented for optimal or suboptimal solution of the mapping problem, the objective functions for mapping and necessary conditions of optimal mapping is developed. It is shown by simulation that the proposed algorithm is faster as compared to the sequential centralized algorithm.

2. Optimal mapping problem

Let a model of a parallel program be the graph $G_p = (M, E_p)$, where M is a set of modules (processes), E_p is a set of edges, representing information connections between modules. Let t_i be defined as weight

of the module $i \in M$, representing the execution time (or the number of computational steps) of the module i . Let v_{ij} be defined as weight of the edge $(i,j) \in E_p$, representing the number of information units passed from module i to module j .

A model of multiprocessor system with distributed memory is an undirected graph $G_s = (P, E_s)$ representing the network topology (structure) of the system, where P is a set of PE, and edges E_s represent interconnection links between PE.

The distance between nodes i and j of the graph $G_s(G_p)$ is denoted as $d_{ij}(r_{ij})$. The neighbourhood of the node i with the radius $\rho \geq 1$ is the set $L_\rho(i) = \{j \in M \mid r_{ij} \leq \rho\}$. Let $D(i) = L_1(i)$.

Let $\varphi: M \rightarrow P$ is the mapping of an information graph of the parallel program G_p into the structure G_s of CS. Let the mapping φ be represented by vector $X = \{x_{ik}; i \in M, k \in P\}$, where $x_{ik} = 1$ if $\varphi(i) = k$ and $x_{ik} = 0$ if $\varphi(i) \neq k$.

Let the quality of the mapping parallel program graph into structure of CS for the given vector X is described by functional: $F(X) = F_E(X) + F_C(X)$, where $F_E(X)$ represents computational cost (the overall module execution time of parallel program on system or the load balancing of PE for given X) and $F_C(X)$ represent interprocessor communication cost (the overall interaction time between modules, which are distributed in different PE, or the distance between adjacent modules of the program graph on CS structure for given X). The optimal mapping problem of parallel program graph into CS structure consists in optimization of the functional $F(X)$ by means of the parallel distributed algorithm.

3. Cost functionals of mapping

Now let us describe two examples of the objective cost functions which represent the computational load balance of PE and communication cost for the given mapping X .

$$F_1(X) = \sum_{k=1}^n \left[\sum_{i=1}^m x_{ik} - Mx_k \right]^2 + \sum_{p=1}^n \sum_{i=1}^m \sum_{j \in D(i)} d_{kp} x_{ik} x_{jp},$$

where $Mx_k = \sum_{p \in L_\rho(i)} \sum_{i=1}^m x_{ip} / |L_\rho(k)|$.

The first term in this expression describes deviation of PE_k load in the neighbourhood of PE_k with the radius $\rho \geq 1$, the second term describes the distance between PE_k and processors, which contain the adjacent modules to the modules embedded into PE_k , $1 \leq k \leq n$.

$$F_2(X) = \sum_{k=1}^n \sum_{i=1}^m (t_i x_{ik} + \sum_{p=1}^n \sum_{j \in D(i)} c_{ij}(v_{ij}) d_{kp} x_{ik} x_{jp}),$$

where $c_{ij}(v_{ij})$ is the time needed to transfer v_{ij} data units between modules i and j , when they are located in the neighbouring PE. The first term in this expression describes the overall module execution time, the second term describes the overall interprocessor communication time, $m = |M|$, $n = |P|$ and $\sum_{k=1}^n x_{ik} = 1$.

Thus, the optimization of the mapping φ consists in minimizing the nonlinear function $F(X)$ with linear restrictions and integer variables. Let Z denotes this task. The optimal solution of the task Z will be found by the distributed difference method ⁴.

4. Conditions for optimal mapping

Let $\Delta F(x_{ik}) = F(x_{ik} + 1) - F(x_{ik})$ be a difference operator, $i \in M$, $k \in P$.

Let $P^+ = \{k \in P \mid \exists i \in M: x_{ik} > 0\}$, $P_i^+ = \{k \in P \mid x_{ik} > 0\}$, $P_i^0 = \{k \in P \mid x_{ik} = 0\}$.

Analogous notations are introduced for the set M . Let $d = \max_{k, p \in P} d_{kp}$.

Theorem. Each following condition is the necessary condition for the optimal solution X of the task Z :

a) for every $i \in M$, $k \in P^+$ and $p \in L_\rho(k) \cap P_i^0$ the following must be carried out with $\rho \geq 1$: $\Delta F(x_{ip}) \geq \Delta F(x_{ik} - 1)$;

b) for every $i \in M_k^+$, $k \in P^+$ and $j \in L_\rho(i) \cap M_k^0$ the following must be carried out with $\rho \geq 1$: $\Delta F(x_{jk}) \geq \Delta F(x_{ik} - 1)$;

c) for every $i, j \in M$, $k \in P_i^+$ and $p \in L_\rho(k) \cap P_j^+$ the following must be carried out with $\rho \geq 1$: $\Delta F(x_{ip}) + \Delta F(x_{jk}) \geq \Delta F(x_{jp} - 1) + \Delta F(x_{ik} - 1)$.

The necessary conditions are sufficient, if the function $F(X)$ are convex and $\rho=d$.

The proof of the theorem is analogous to the proof of Theorem 1⁴.

5. Parallel distributed algorithm for mapping optimization

The presented conditions are constructive and allow us to design the following parallel distributed algorithm for has been proposed optimal and suboptimal solutions of the mapping problem. Each processor $k \in P$ executes an iterative process which uses the local information from the neighborhood $L_\rho(k)$ and performs of the following actions:

1) The necessary conditions a) and c) are verified with the given radius ρ in the processor $k \in P$ for each module $i \in M$ allocated in the processor.

2) If the condition a) is not carried out for the module i in the processor $p \in L_\rho(k) \cap P_i^0$, then the vector X is corrected by transmitting the module i (or only its descriptors and pointers) from processor k to p : $x_{ik} := 0$; $x_{ip} := 1$.

3) If the condition c) is not carried out, then the processors k and p exchange the modules i and j (or only their descriptors and pointers) between themselves: $x_{ik} := 0$; $x_{ip} := 1$; $x_{jp} := 0$; $x_{jk} := 1$.

These actions are executed at each modification of a quantity or a set of modules in the processor k .

The convergence of the algorithm to a local optimum is demonstrated and, to global optimum - if sufficient conditions from theorem are fulfilled.

The important part of the proposed algorithm if considering load balancing only was investigated by simulation on IBM PC/AT. It was shown that the uniform distribution of load is reached $O(n)$ times faster than the sequential centralized algorithm. Fig.1 shows the number of iterations (K) of load balancing part of algorithms as a function of PE number (N) in CS. Fig.2 shows the deviation the of PE load from mean system load (F), which provided by allocation algorithms, as a function of PE number in CS. The dotted lines

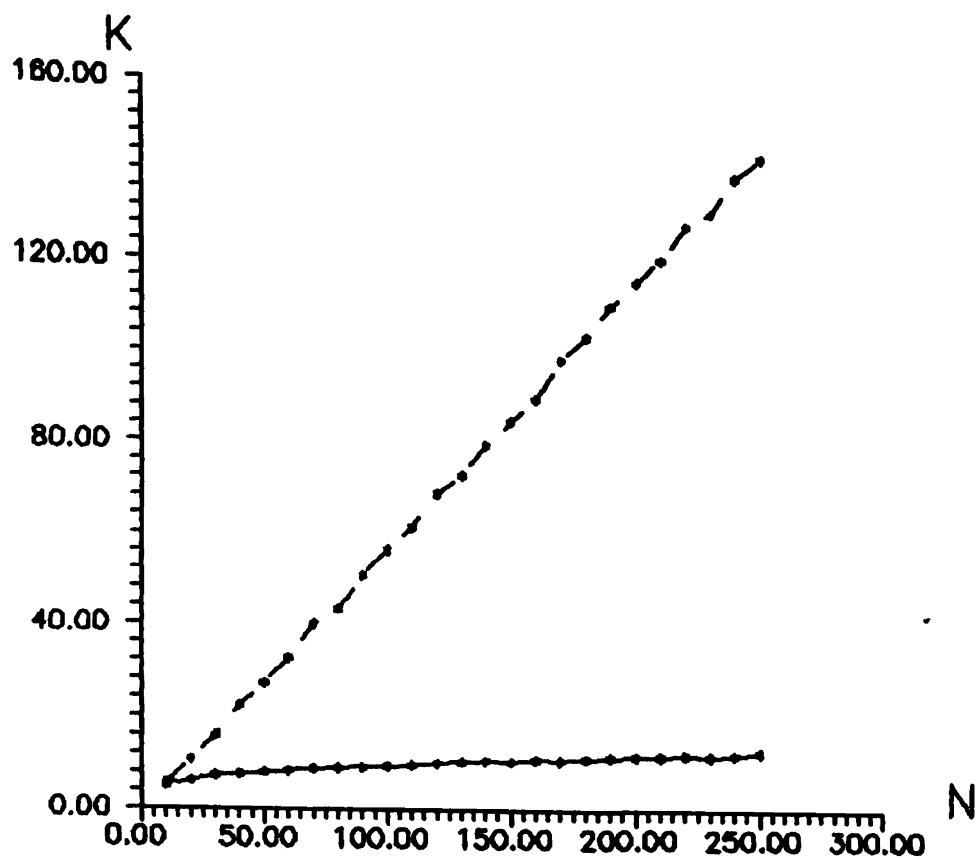


Fig.1

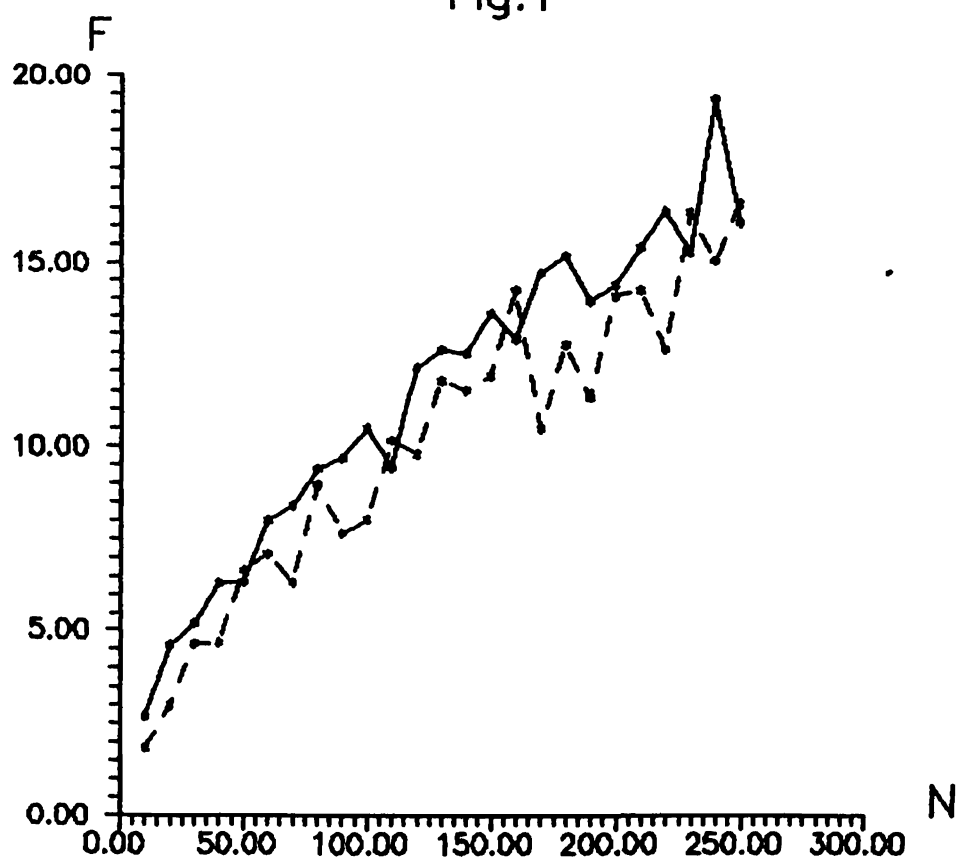


Fig.2

correspond to sequential centralized algorithm with cost function:

$$F_1(X) = \sum_{k=1}^n [(\sum_{i=1}^m x_{ik} - Mx_k)^2],$$

and solid lines correspond to parallel distributed algorithms with value of neighbourhood $|L_{\rho}(k)| = 2\sqrt{n}$.

Thus, the proposed parallel distributed algorithm is efficient and produces a optimal or good suboptimal solution of the mapping problem. The algorithm is based on the local information used to minimize the total cost function. It suits well for application in distributed operation system of CS with reconfigurable structure.

6. References

1. N.N.Mirenkov, *Parallel programming for multimodule computer systems*, (Radio i svyas, Moscow, 1989), 320p.
2. F.Berman, L.Snyder, On mapping parallel algorithms in parallel architectures.-*J. Parallel Distrib. Comput.*, 4 (1987) pp.439-458.
3. D.Fernandez-Baca, Allocating modules to processors in a distributed system.-*IEEE Trans. Software Eng.*, 15 (1989) pp.1427-1436.
4. O.G.Monakhov, Optimization of the distributed resource control in CS with programmable structure.-*Programmirovanie*, No.5, (1985) pp.94-105.

DYNAMIC PARALLELISM IN TRANSPUTER SYSTEMS

A. V. Anisimov

Department of Mathematical Informatics,
Kiev State University, Vladimirskaia, 64, Kiev-17, USSR.

Yu. E. Boreisha

Department of Mathematical Informatics,
Kiev State University, Vladimirskaia, 64, Kiev-17, USSR.

ABSTRACT

The family of languages implemented on transputer systems (Parallel C, Parallel Pascal and Parallel Fortran) is extended by new program tools providing dynamic parallelism. These tools enable the user to create parallel processes and reconfigure communication links between them dynamically. These tools form powerful addition to existing methods for parallel programming such as Black Boxes, Parallel Execution Threads and Processor Farms. It is called PARCS - Parallel Asynchronous Recursively Controlled Systems. The PARCS-tools provide the opportunity of parallel programming in such a style as in ADA language on the base of any algorithmic language. So the user who is adopted to his favourite languages does not need to learn new ones. Just use our extension of a chosen language.

1. Introduction

Parallel systems are characterized by parallel functioning and interconnections of their parts. In dynamic parallel systems with changable structure objects and communication links can be created and annihilated dynamically. For this parallel systems special programming means are necessary. An interesting set of such means was sugge-

sted in the programming language ADA. In parallel programming languages implemented on transputers (Occam, Parallel C, Parallel Fortran, Parallel Pascal) there are no means provided necessary dynamic changes in processes. The main attention in them is paid to static concurrency. Very simple means of dynamic parallelism are provided by special constructions of Parallel Execution Threads and Processor Farms.

The programming system PARCS (Parallel Asynchronous Recursively Controlled Systems) which was designed at the Kiev State University (USSR) is the set of programming means supporting design and implementation of parallel algorithms by means of extensions of basic programming languages (such as C, Pascal, Fortran, Modula-2). The PARCS system combines very powerful algorithmic conceptions - recursion and concurrency and give adequate descriptions of dynamic interconnections in processes with changable communication links. Kernels of this processes are given in basic chosen languages. For the CSP-model of communication (which is the basis of communication procedures of ADA and Occam) PARCS-tools provide dynamic creating and changing parallel processes.

The programming system PARCS supports PARCS technology of programming. The main conception which is exploited in PARCS is the introduction of so called Controlling Spaces. A Controlling Space can be viewed as a changable graph describing communication and logic structure of a solving problem. The PARCS programming system provides programming means which describe construction of a Controlling Space, storing and transportation of information by means of this Controlling Space, loading of the Controlling Space by algorithmic modules.

Proposed PARCS-tools are intended for programming of

parallel systems with dynamically changable structures and designed according to the PARCS-technology of programming. PARCS-technology has been developed by authors and is devoted to solving problems of static, dynamic and structural parallel programming.

PARCS-tools (in the proposed first version) are implemented on the transputer ring with any number of transputers (but less then 256). Resource allocation and routing problems are solved automatically with the help of monitors of dynamic mapping. The maximum number of concurrently executed processes is restricted by the number of transputers in the ring (in the proposed version for N-transputers it is equal to $(N-1)*255$, but it isn't a principal limitation). Next versions with the higher performance characteristics will be followed.

Many applied problems connected with design of dynamically changable graphs, parallel algorithms for computer systems with dynamically changable structure, algorithms of artificial intelligence and etc. can be solved in better way using transputers and our developed tools.

2. PARCS-tools

The essence of using PARCS-tools is the following (with Parallel C as an example). In the developed parallel system the final number of basic algorithmes is chosen. They are represented in the form of procedures in the using algorithmic language. On the base of these procedures parallel processes could be created by means of PARCS-tools during the work of a parallel program. The number of parallel processes is limited only by the memory size of transputers. A parallel process finishes its work when it has completed the execution of its corresponding procedure.

Each process has an unique name which it receives from

the system during the creation. For creating parallel processes the following function is used:

```
point setp(t)
    int t;
```

Parameter "t" defines the base procedure (procedure must be reenterant) on the pattern of which the next parallel process will be created. This function returns an unique name of the created process (type of a point) which is used for the access to this process.

Among all base procedures the main is chosen. The main parallel process is created on the base of this procedure automatically. This process starts the work of the whole parallel system.

The header of each base procedure has the following form:

```
#include "p.h" /* PARCS header */
proc_name (myp)
    point myp;
```

Through the parameter "myp" the parallel process which is created on the base of this procedure will receive its own name automatically.

Communication of parallel processes is realized by means of a message exchange. Message sending is realized by the following function:

```
sendp (p, ch, msg)
    point p; int ch; char *msg;
```

where parameter "p" defines the name of the target process, "ch" - the entry point number in it, "msg" - the sent message. Message sending is realized without delay. The only necessary condition for correct sending is the existence of a corresponding process with the name p.

Type point is realized as

```
#define point int
```

Dynamic reconfiguration of communicating links in the system is realized by means of processes names transferring with the help of parameters during the message exchange.

Message receiving is realized with delay by the following function:

```
getp (ch, msg)
int ch; char *msg;
```

where parameter "ch" defines the entry point number in which message is waited and will be received in the variable "msg". The part of the set of PARCS-functions has service purposes: for example - endp() - correct finishing of the process work; printp(msg) char *msg - printing the message "msg"; reportp() - printing the current state of the whole system, nump() - returns the number of transputers in the system (is used only in the root process).

The set of functions which realize exchange of integer numbers, integer sets and integer arrays is used by means of the following header:

```
#include "pi.h"
```

For example, the function:

```
sendp_i (p, ch, w)
point p; int ch; int w;
```

is proposed for sending integer value "w" to the process "p" in entry "ch".

For receiving in entry "ch" an integer value the following function is used:

```
getp_i (ch, w)
int ch; int w;
```

Analogous opportunities for exchange of float numbers, float sets and float arrays are realized by means of the header

```
#include "pf.h"
```

3. Creation and execution of PARCS-programs

For the execution of program with PARCS-tools it is necessary to create next files:

- file mtask.c with main procedure declaration root;
- file wtasks.c with all work procedures for base algorithms;
- file defwt.h with definition of work procedures (see appendice);
- file spwt.h with specification of work procedures (see appendice).

4. References

1. *Parallel C. User Guide* (3L Ltd., 1988).

Appendice: Example of multiplication of long numbers

It is necessary to execute value of $a \cdot b$, where a and b consist of n digits, and processor can multiplies numbers consisting from q digits (for example, $n = 16$, $q = 4$).

Numbers a and b are distributed from left to right on parts with q digits in each:

$$a = a_3 a_2 a_1 a_0, \quad b = b_3 b_2 b_1 b_0,$$

where any from numbers a_i, b_i ($i=0, k$) contains $\leq q$ digits ($k=n/q-1$). The result of multiplication is calculated as shown below:

$$\begin{aligned} a * b = & (a_0 * b_0) + \\ & (a_0 * b_1 + a_1 * b_0) * 10^{**q} + \\ & (a_0 * b_2 + a_1 * b_1 + a_2 * b_0) * 10^{** (2 * q)} + \\ & (a_0 * b_3 + a_1 * b_2 + a_2 * b_1 + a_3 * b_0) * 10^{** (3 * q)} + \\ & (a_1 * b_3 + a_2 * b_2 + a_3 * b_1) * 10^{** (4 * q)} + \\ & (a_2 * b_3 + a_3 * b_2) * 10^{** (5 * q)} + \\ & (a_3 * b_3) * 10^{** (6 * q)} \end{aligned}$$

From this record the recursive and parallel algorithm can be derived. The logical structure of this algorithm is


```

#include <par.h>
#include "p.h"
#include "pi.h"
void root(myp) point myp;
{ int q,d,u,a,b,i,n;
  char buf_msg[20];
  point p;
  /* Enter the values of q,d,u */
  p=setp(1);
  sendp_i4(p,1,q,d,u,myp);
  for(i=d; i<=u; i++)
  { /* Enter the next values of a,b */
    sendp_i2(p,2,a,b); }
  for(i=0; i<2*u+1; i++)
  { getp_i(1,n);
    /* Print the value of n */ }
}

```

```

/* wtasks.c */
#include "p.h"
#include "pi.h"
void mast(myp) point myp;
{ int q,d,u,a,b,s,n,i;
  char buf_msg[30];
  point l,r,out;
  printp("Mast begin the work");
  s=0;
  getp_i4(1,q,d,u,out);
  if(d<u)
  { r=setp(3);
    sendp_i5(r,1,q,d,u-1,myp,out);
    l=setp(2);
    sendp_i4(l,1,q,d+1,u,out); }
}

```



```

for( i=d; i<=u; i++)
{ getp_i2(2,a,b);
  s=s+a*b;
  if( i<u)
  { sendp_i(r,2,a);
    sendp_i(l,2,b); }
  if( i>d)
  { sendp_i(r,3,b);
    sendp_i(l,3,a); } }
if(d==u)
{ sendp_i(out,1,s); }
else
{ getp_i(4,n);
  s=s+n;
  n=s%power(10,q);
  sendp_i(out,1,n);
  n=(s-n)/power(10,q);
  sendp_i(l,4,n); }
endp();
}

void left(myp) point myp;
{ int q,d,u,a,b,s,n,i;
  char buf_msg[30];
  point l,out;
  printp("Left begin the work");
  s=0;
  getp_i4(1,q,d,u,out);
  if(d<u)
  { l=setp(2);
    sendp_i4(l,1,q,d+1,u,out); }
  for( i=d; i<=u; i++)
  { getp_i(2,b);

```

```

    getp_i(3,a);
    s=s+a*b;
    if(i<u)
    { sendp_i(1,2,b); }
    if(i>d)
    { sendp_i(1,3,a); } }
getp_i(4,n);
s=s+n;
if(d==u)
{ sendp_i(out,1,s); }
else
{ n=s%power(10,q);
  sendp_i(out,1,n);
  n=(s-n)/power(10,q);
  sendp_i(1,4,n); }
endp();
}

```

```

void right(myp) point myp;
{ int q,d,u,a,b,s,n,i;
  char buf_msg[30];
  point r,out,f;
/* It is similar to left */
}

```

```

/* defwt.h */

```

```

extern void mast();extern void left();extern void right();

```

```

/* spwt.h */

```

```

task_sp(1,mast,100); /* workspace - 100 words */
task_sp(2,left,100); task_sp(3,right,100);

```

HINTERCUBE: THE PROJECT OF A NEW GENERATION MULTIPROCESSOR SYSTEM

S.A.RAKOV, A.A.SHVEIN, and V.A.SIGNAEVSKII
Department of Computer Systems
International Centre on Informatics and Electronics
Presnensky val 19, Moscow, 123557, USSR

ABSTRACT

Advanced computer technology and parallel multiprocessor architecture allow designing of high-performance general-purpose computer systems with high I/O bandwidth and processor performance balanced with memory access time. These computer systems will provide modularity, extendability, and high performance/cost value. HINTERCUBE is an investigation of computer system architecture suited to meet these requirements.

1. Introduction

High-performance computer systems are taking the most promising market sector. Extreme growth can be observed in parallel computers and computer systems. Computer market experts predict 48% of total computers in use up to 1992 to be parallel¹. The growing production of desk-top personal supercomputers is awaited to reach \$4.5 billion of total revenue in 1991, and further growth \$1 billion per year.

The HINTERCUBE project was started in the International Centre on Informatics and Electronics in 1990. The aim of the project was to develop implementation techniques of general-purpose high-performance parallel computer systems for a wide range of applications.

The most significant features of this new generation system assumed to be as follows:

a. The new system is to be fitted to general-purpose application, provided to solve a plenty of problems in those areas, where super-computers are too expensive, while conventional computer systems, including PCs, minis, and mainframes are of insufficient computing power.

b. The new computer system must provide flexible extendability, so as to create application-dependent workstations upon a unified technological base with a wide scope of parameters:

- 1 to 1000 MFLOPs performance,
- 4 to 2000 Mbytes of main memory,
- 0.4 to 1000 Gbytes of mass memory.

c. The architecture of the new generation computer system is to be developed in such a manner as to run the conventional and advanced software with a minimum of modification, though taking major advantage of the special tools for parallel processing.

d. The system must have fault-tolerance and diagnostic capabilities, providing localization of the fault modules, while the program is running on the others.

We discuss below some source keypoints taken into account during the first stage of the project, and present some technical solutions of the multiprocessor architecture being now under development.

2. SOME TRENDS IN GENERAL-PURPOSE COMPUTER ARCHITECTURE

We can see two major trends in general-purpose computer architecture.

The first one is forced by extremely fast improvements in semiconductor technology. The leading classes of products in this area are the microprocessor chips and the memory chips. Due to a significant increase in microprocessor performance, memory capacity, and cost-effective manufacturing, nowadays PCs are widely used in conventional computations instead of recent minis and mainframes.

The most important result of this PC expansion in computer world is an enormous amount of the PC software products. When developing a new general-purpose computer architecture, the portability of these products should be provided.

As to PC architecture, it is today's implementation of the classical von Neuman's computational model. Some new developments in the PC architecture deal either with a microprocessor level (such as RISC-processors), or with an interface level. These architectural improvements do not affect the entity of the classical computational model. So the most valuable increase in technical parameters for PCs is based on technological development.

The most popular microprocessor chips used in PC design are those of Intel 80x86 family provided with top-down software compatibility.

The second trend in computer architecture development is to achieve higher performance using parallel computation on multiple processors.

Current semiconductor technology level enables a conventional processor to be implemented on a single chip, thus providing good opportunities for creating a general-purpose high-performance multiprocessor system.

Several designs of multiprocessor computer systems have been accomplished in the last few years. Among the most successful systems of this type were the Intel iPSC/2 based on 80386 chip² and the Inmos transputer systems on several general-purpose computer platforms, including IBM-compatible PC³.

This type of computers allows to get significant increase in performance/cost value due to low-cost standard hardware and cost-effective implementation of conventional sequential software.

3. PARALLEL COMPUTATION METHODS

There are different aspects of parallel computation methods depending on hardware solutions and application requirements.

The first kind of parallel computation is based on simultaneous evaluation of similar operations. It is commonly used in vector and array processors. These are likely to benefit significantly in special limited application areas with a high proportion of array and vector operations.

The other kind of parallelism is known as communicating sequential processes⁴. It is more appropriate to be a good theoretical model for a general-purpose multiprocessor system. Meanwhile, a true general-purpose system, being efficient in any application area, should employ both kinds of hardware-specific parallelism.

As to application-specific aspects of parallel computation, there exist two approaches to its classification. The first one is by granularity defined as a measure of the amount of computation in relation to the amount of communication of results between parallel computations⁵. According to this classification, fine-grained and coarse-grained parallelism are known.

The second application-specific approach is by method of solution which includes three subdivisions:

algorithmic, geometric and farm parallelism. The algorithmic parallelism demands the solution algorithm to be divided into modules which can execute concurrently. In case of the geometric parallelism, the problem space is divided in such a way that different sections (usually of equal size) can be computed concurrently. As far as the farm parallelism is employed, the problem space is divided into independent sections, and the solution is reached by the farmer processor sending work to a farm of worker processors as they get ready to compute.

All these kinds of parallelism could be successfully exploited by a general-purpose multiprocessor system with an appropriate design of processing nodes, interprocessor communications, and i/o interfaces.

4. PROCESSING NODES IN THE MULTIPROCESSOR SYSTEM

At the first stage of the HINTERCUBE research project, a plenty of microprocessor families were considered as possible candidates to be used in processing nodes. To make a proper choice of a microprocessor, the following characteristics were investigated: a microprocessor performance on general-purpose and special instructions, efficient processor-memory interface, interprocessor communication capabilities, and PC-architecture compatibility.

From the performance point of view, the most promising microprocessor currently available is Intel 80860⁶. It has the general-purpose instruction RISC-processor, the 64-bit floating-point processor with pipelined instructions suitable for vector and array computations, the 3-D graphics processing module, and the memory management module, all implemented on a single chip. Due to local parallelism, 128-bit internal data bus, and 64-bit external data bus, its peak performance at 40 MHz cycle reaches 80 MFLOPs for single-precision and 60 MFLOPs for double-precision instructions. On-chip cash memory (4K bytes for instructions and 8K bytes for data) together with on-chip memory address translation buffer, provide high real performance and Intel 386/486 virtual memory format compatibility.

Together with sufficient local memory, 1860 microprocessor is a good base for creating a very powerful computing node for application solutions with intensive use of matrix and vector operations and fine-grained parallelism. Further increase of computing power can be

achieved due to 64-bit data exchange between processing nodes in multiprocessor system.

The other types of parallel applications with relatively small amount of inter-processor data transfers, may employ another microprocessor, that is the INMOS transputer. It consists of a 32-bit general-purpose processor and on-chip inter-processor links providing an ability to connect any number of transputers in arbitrary networks.

Recent implementation of the Transtech TTM100 transputer module⁷ demonstrated that combining 1860 computing power and transputer flexible communicating opportunities is a good solution to design an appropriate processing node for a high-performance general-purpose multiprocessor.

5. INTER-PROCESSOR COMMUNICATIONS

Inter-processor communication was the second valuable aspect of a multiprocessor architecture design of the HINTERCUBE project. Real computer system performance depends in much degree on the amount of computational resources spent for inter-processor exchange. Since this amount is very different for various applications, the problem causes more difficulties in the case of a general-purpose multi-processor architecture.

Depending on hardware resources expenditure four major communication net types can be distinguished. The complexity of each type is a function of the number n of communicating nodes.

The first class is of complexity n^2 . Its typical example is a full commutator. The second class is of complexity $n \log_2 n$, such as hypercube communication system. The third class is of complexity n with a mesh communication net as an example. And the fourth class is of complexity c , not depending of n , such as bus connection.

For the first three classes of communication nets, the peak bandwidth grows proportionally to n , though the real bandwidth occurs to be lower because of conflicts. As for the fourth class assuming bus inter-connection, with initially high bus bandwidth, it is the most effective communication system until the total computation performance becomes so large with growing n that the bus bandwidth tends to be a bottleneck.

A reasonable solution is to create a multi-level bus communication net. It has real benefit in the case when

the whole application task can be divided into several subtasks with low-intensive exchanges between each other. In this case, the first level bus connections between the microprocessor and the local memory inside the single processing node have a minimal length and a maximal bandwidth. The second-level bus connections unite several processing nodes into a processing module. The third-level bus connections integrate processing modules into a multi-module system, etc. As a rule, next level connections tend to be longer and slower, and perform exchanges between relatively large weak-bound subtasks.

In our case using i860-based processing nodes, when the number of processing nodes in the system is small enough (up to 16), 64-bit multi-level data bus is a reasonable way to provide extendability of computing power required by fine-grained parallel applications. But the applications employing other type of parallelism may require more processing nodes. For this purpose another level of extendability is to be provided using a communication net of higher complexity, such as $\log n$ -complexity hypercube connections.

6. INPUT-OUTPUT INTERFACE

The extreme growing of the total computing performance in multi-processor systems requires an appropriate increase in input-output bandwidth. Otherwise for i/o-intensive applications, the benefit of multi-computing power will be significantly lowered because of i/o bottleneck.

This problem can be solved by the choice of high-bandwidth i/o interface, providing concurrent i/o processing on different computing nodes and the system extendability at the input-output level. On the other side, in general-purpose multi-processor systems standard interfaces are desirable to provide using a wide range of i/o devices.

In the HINTERCUBE project about twenty conventional and future interfaces were considered. For the first version of a system, SCSI-2 bus allowing 10 to 40 Mbytes/sec is planned. Depending on the application requirements, each processing module consisting of up to 16 nodes can be provided with 4 i/o buses.

7. The Summary of HINTERCUBE Architecture

A wide range of general-purpose high-performance multi-processor computer systems having various performance, memory volume, and i/o bandwidth can be constructed from a limited number of hardware modules.

The processing node is based on the i860 microprocessor containing 8 to 32 Mbytes of local/shared memory, and two interfaces. The first interface connects the processing module with i/o adapter, and the second one provides integration of processing nodes into the first-level multi-processor module.

The first-level multi-processor module consists of up to 4 processing nodes connected with the 320 Mbytes/sec 64-bit first-level system bus, the i/o bus adapter, and the second-level system bus adapter or a next-level transputer/hypercube adapter. The first-level system bus enables an access for any processing node to the local/shared memory of any other processing node in the same first-level multi-processor module.

The second-level multi-processor module consists of 2 to 4 first-level multiprocessor modules connected with the second-level system bus. This bus is similar to the first-level system bus and provides for any processing node an access to the local/shared memory of any other processing node inside this second-level multi-processor module. The maximum number of processing nodes in a second-level module is 16 with up to 512 Mbytes of memory, its peak performance is up to 1.92 GIPS and the peak i/o bandwidth is up to 640 Mbytes/sec.

Higher-level multiprocessor systems assumed to be constructed through transputer-like links to get a hypercube connection. The hypercube nodes are first-level multi-processor modules. A total number of processing nodes in a maximum hypercube configuration is 1024 with up to 32 Gbytes of memory and 120 GIPS total peak performance.

8. Conclusions

In this paper, we have discussed main initial concepts used at the first stage of the HITNERCUBE project. Our aim was to develop a cost-effective general-purpose multiprocessor architecture, which could be used in various applications with different parallel computing methods. Detailed technical solutions and software design strategy are now under development.

9. References

1. L.Curran, *ED*, No.11(1989) p.41-47.
2. D.Moody, in *Evaluating Supercomputers. Strategies for exploiting, evaluating and benchmarking computers with advanced architecture* (UNICOM, London, 1988) p.55-59.
3. *Transputer Development and iq Systems Databook* (Inmos Ltd.,Trowbridge,1989).
4. C.A.R.Hoar, *Communicating Sequential Processes* (Prentice-Hall,London,1985).
5. R.J.Best, *SERC/DTI Transputer Initiative* (Dec.1990 - Jan.1991) p.39-49.
6. *i860 64-bit Microprocessor. Advance Information* (Intel Corp.,1989).
7. S.J.Bradshaw, *TTM100: An i860 Processing Unit for Multi-Processing Systems* (Transtech Device Ltd., High Wycombe, 1990).

Parallel Programming Technology And Resource Management In Multiprocessor Systems

L.K.Babenko, O.B.Makarevich, A.G.Chefranov

Scientific Research Institute of Multiprocessor Computing Systems, 347928, Taganrog, USSR

Abstract

Parallel programming technology based on ideas of macrotasking level of parallelism using Fortran extensions and problems of resource management in multiprocessor system like ES-2703 are discussed in the paper. Users tasks are represented in the form of control graph with vertices corresponding to subtasks and arcs corresponding to the ways of control flow between subtasks. Tools for convenient description of control graph and efficient algorithm for scheduling ready for execution subtasks are given.

Introduction

Multiprocessor systems (MS) resource management problem while using macrotasking level [1,2] is one of the most hard problems of parallel computations. Effectiveness of its solving depends on many factors main of which are the following:

- 1) programming language parallelism definition tools;
- 2) ready for solving tasks scheduling algorithm.

Suggestions on realization of these two questions providing effective resource management for MS like ES-2703 [3] are given here. Structure of work is the following. Information about ES-2703 MS architecture is given in the first part; the second part is devoted to describing of Fortran using technology for parallel programming and resource managing on programmers level for ES-2703 like MS; in the third part resource management on operating systems level is discussed. The last part is summary of the work.

1. ES-2703 Multiprocessor system architecture

MS ES-2703 is created in the Scientific Research Institute of multiprocessor computing systems, Taganrog; this system may be classified as MIMD-system. MS ES-2703 can work with computers of ES class and IBM PC compatible computers which play a role of a host machine for MS. Host prepares programs for MS execution (editing, compiling and so on). MS ES-2703 itself consists of central control unit (CCU) and solving field (SF), the last is the set of up to 256 identical macroprocessors (MAP). CCU and inter-MAPs are linked by common control and information bus, inter-MAP communications are maintained by full-accessible matrix switching system. Technical characteristics for one model of the family of ES-2703 like MS are given below:

- number of MAPs - 96
- number of input-output channels - 32
- maximal performance MAP - 12 MFLOPs
- throughout of matrix switch - 512 Mbytes/s
- system memory - 256 Mbytes RAM
- data formats - 32, 64 bits
- hardware basis - large scale integrated circuits of 1834 series,
16x16 one bit crossbar switch LSI circuit of
1509 RG1 series.

Structure of the system is given in Fig.1. Exchange controller (EC) controls informational exchanges between MAPs. Input-output is performed via input-output channels under control of input-output controllers (IOC) and file server (FS).

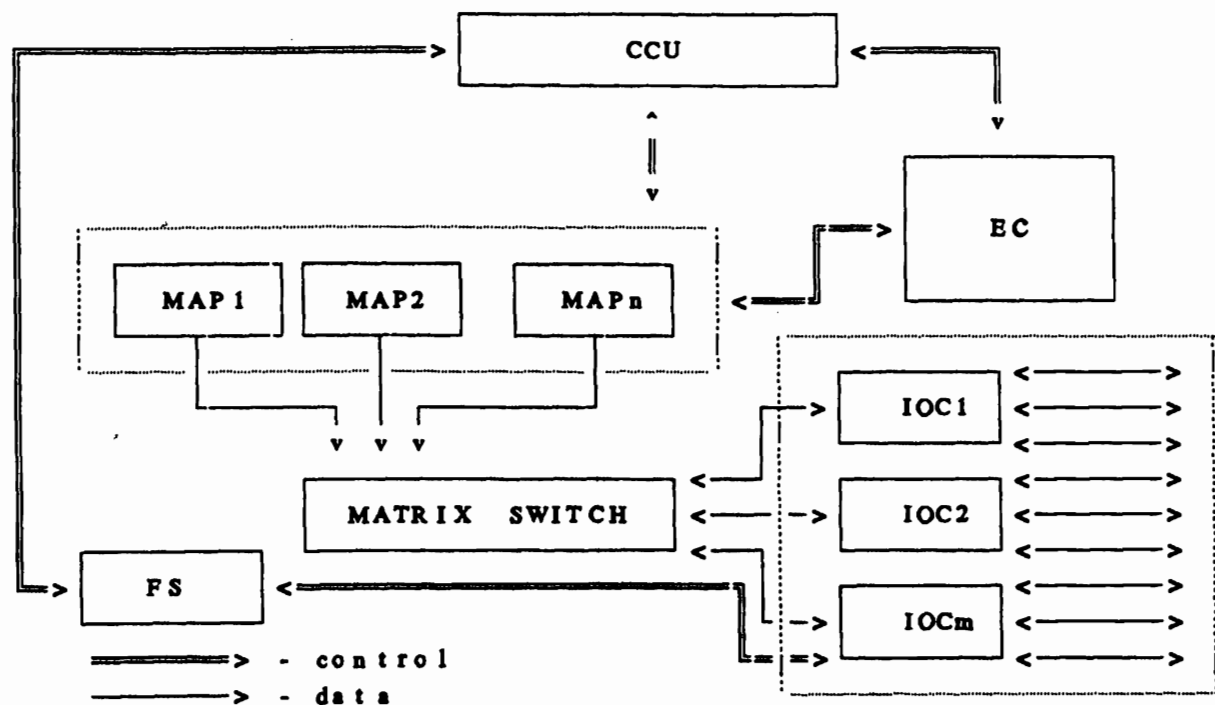


Fig.1. System structure

Task executing goes under control of MS operating system. MS operating system is resident in CCU and manages CCU and SF resources. Each MAP and CCU are processors with instruction and data memories. Local CCU data memory is at the same time global system data memory to which MAPs can have access via common information bus. MS in general case executes more than one task at a time. Each task is a set of one control module (CM) executed by CCU and a number of processing modules executed by MAPs. PM can be executed simultaneously by several MAPs; the necessary number of MAPs is determined at the calling PM moment. All MAPs occupied by PM get one and the same programme but different data sets. Each MAP operates in uniprogramming mode and CCU in multirogramming mode using time-sharing. CM's aim is to organize computations on the PMs level according to the requirements of the whole task. For the sake of this aim CCU has readiness conditions for each PM. Readiness of the PM for execution is determined by passing to it control from its predecessors in the task control graph. Let, for example, the task is represented by task control graph from Fig 2.

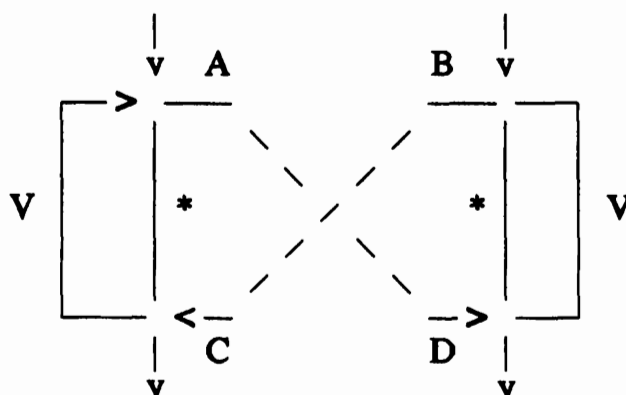


Fig.2. Example of task control graph

Conjunctive and disjunctive input (output) arcs of the graph are labeled by characters *, V respectively. Conjunctiveness (disjunctiveness) of input means that control must be accepted from each (any) of the input arcs for each realization. Similarly, conjunctiveness (disjunctiveness) of output means that in every realization control must be given in all (any) of the output directions. As it can be seen from Fig.2., part of the PMs are ready for execution at start moment (A,B); part of modules becomes ready when getting control from their predecessors (C,D); execution completion of the part of modules (C,D) may signal about task completion. If we denote by logical expression the fact of transmitting control from module M1 to module M2 (this expression has truth value "true" from the moment of transmitting control from M1 to M2 until M2 completion time moment and truth value "false" in other moments) then readiness condition for module C is the following

$$(A \rightarrow C) \& (B \rightarrow C) \& (C \rightarrow A)$$

The last conjunctive member of this expression appears due to the obvious remark that module can become ready not only in the case of getting control from its predecessors (input readiness) but also if it did not yet give control to its successors or if they got control from this module and have already been completed (output readiness). Thus to become ready module must be input ready and output ready.

Modules, being ready at the beginning of executing, are represented in the control graph by vertices labelled by input arrows (arcs without radiating them vertices). It is accounted that execution of the task is terminated if control is given in the direction of all output arrows (arcs without absorbing them vertices). For the sake of standartization it may be introduced standart fictitious vertice named Final, readiness condition for which is transmission of control along all the output arrows; readiness state of this module means terminations of the whole task. Thus for example in Fig.2 fact of terminating of the task is determined by truth value of the following logical expression

$$(C \rightarrow \text{Final}) \ \& \ (D \rightarrow \text{Final})$$

Readiness conditions of PMs are checked by CU operating in CCU on the basis of information about control transmission between PMs. This information enters CCU in the following way. All MAPs occupied by any PM work under control of one and the same programme, which has instructions for determining the PMs successors. As long as each MAP has its own data, decisions obtained in different MAPs of this group may be also different, therefore the following strategy is used. Each MAP occupied by PM, when processing has been completed, prepares information for transmitting to CCU and sets up completion flag. This information includes control information and processing results. When CCU detects that all MAPs of the module set up completion flags it begins to accept control information and processing results from MAPs. Control information is accepted only from MAP with logical number 0 in the group of MAPs occupied by respective PM and is fixed in special operating system control tables.

Let us now discuss how such interactions of CM and PMs with the help of programming language based on Fortran-77 can be represented.

2. Technology of using Fortran for parallel programming and resource management

Programmes for CM and PM are written in the Fortran. Each of these modules is similar to main programme of the usual Fortran; standartly prepared subroutines may be used by modules in usual manner. However modules have heads which determine the type of the module (control or processing), modules name and its parameters. Heads of CU and PM have respectively the following form

```
control  <CU's name>
process  <PM's name> (<parameter list>),
```

where <parameter list> is as usual a list of formal parameters divided by commas. It is possible to use in the roles of parameters names of variables, arrays and modules. Arrays as in the usual Fortran are described by instructions pointing type and DIMENSION. All variables and arrays described in module are located in the data memory of each MAP occupied this module. Arrays sizes can be given by expression depending on parameters passed to module (for example, on the processors number required by PM). Attribute of transmitting may be placed accross colon after parameter; it may have one of the following values - in, inout, out. Attribute value in means that corresponding parameter value enters module and does not come back, inout means entering and returning, out means not entering but returning. Default attributes value is in. For example, programmes part given in Fig.3.

```
execute C(A,B.mas:inout.R1,R2)
module A,B
dimension mas (R1,R2)
```

Fig.3. Part of processing module

contains the head of module with the name C which has five parameters: two parameters (A,B) have attribute value inout, last two parameters have attribute value in. First two parameters in this list are modules names, parameter mas is two-dimensional array, parameters R1, R2 are simple variables.

PMs are transmitted for execution with the help of system procedure domodule which is called by CM in a following way:

```
call domodule(processors number, module name,parameters list).
```

The first parameter of domodule determines required for PMs execution number of processors, PM's name is given by the second parameter, parameter list is the list of actual parameters for PM. Actual parameter may be variable, array or module name and also it may be array element, expression, part of array. Part of array is given by following way:

```
array name(triplet list)
```

where triplet list is a list of divided by comma triplets each of them has form:

```
start: finish [:step]
```

where start, finish and step (default value is 1) determine variance interval for corresponding array index while forming part of array and they have sense similar to parameters of usual Fortran do cycle. For example,note

```
mas(1:10,6:10)
```

denotes part of array mas containing elements with the following indices

```
(1,6) ... (1,10)
(2,6) ... (2,10)
...
(10,6) ... (10,10)
```

When actual parameter is array or part of array it may be followed across colon by the mark of individual mailing of its elements among the MAPs. This mark is embraced in brackets list of expressions, number of which is equal to dimension of transmitting array and each expression determines the size of individual block given to each MAP along corresponding directions. This mark may be omitted and then it is accounted that array or its part is fully transferred to each MAP of the PM. For example, executing of the following CM

```
dimension AM(100,100)
```

```
p=5
```

```
call domodule(p,C,A,B,AM:(100/p,100),100/p,100)
```

causes allocation of $p=5$ processors to module C; each of these processors gets block of array AM of the size 20×100 and processors with logical number i gets rows of array with number $20i, 20i+1, \dots, 20i+19, i=0, 1, \dots, p-1$. In order to share data among MAPs of PMs correctly it is necessary to have definitions of the PM's formal parameters for CM. For this purpose specifications of the used PMs are placed before head of the CM in the following form

```
<parameters definitions>
```

```
end
```

for each PM. Parameters list and parameters definitions written above must be the same as in the appropriate PM. When PM is called with the help of domodule test on correctness of the call, addresses of parameters in MAP is performed and size of transferred block are calculated and operating system prepares itself for transmitting them in both directions (CCU-MAP, MAP-CCU) taking into account the attributes of transmitting parameters. Thus the structure of CU realizing control of PMs is organized so as it is presented by graph in Fig.2, is the following:

```
specification A(...)
```

```
parameters definitions
```

```
end
```

```
specification B(...)
```

```
parameters definitions
```

```
end
```

```
specification C(...)
```

```
parameters definitions
```

```
end
```

```
specification D(...)
```

```
parameters definitions
```

```
end
```

```
control CM
```

```
global arrays and variables definitions
```

```
call domodule(p1,A,...)
```

```
call domodule(p2,B,...)
```

```
2 if(A→C.AND.B→C.AND..NOT.C→A) call domodule(p3,C,...)
```

```
if(A→D.AND.B→D.AND..NOT.D→B) call domodule(p4,D,...)
```



```

    if( $C \rightarrow A$ .AND.. $\text{NOT}.A \rightarrow C$ .AND.. $\text{NOT}.A \rightarrow D$ ) call domodule(p1,A,...)
    if( $D \rightarrow B$ .AND.. $\text{NOT}.B \rightarrow C$ .AND.. $\text{NOT}.B \rightarrow D$ ) call domodule(p2,B,...)
    if( $C \rightarrow \text{Final}$ .AND.. $D \rightarrow \text{Final}$ ) goto 1
    goto 2
1 ...
end

```

It must be noted that procedure domodule only tells operating system about readiness of the appropriate PM. Problems of scheduling ready PMs and their allocation to processors are solved by operating system and sequence of PMs starting on MAPs may be else than sequence of PMs transitions in the readiness state.

In order to have possibility of using one and the same PM in several tasks the names of PMs-possible successors of given PM must be transmitted to it as parameters. Extension of usual instruction stop in the form stop list of PMs actual successors is used for pointing out actual successors of the given PM.

For example in Fig.2 concluding part of the programme for PM named C may look like as

```

    if (termination condition) stop Final
    stop A

```

Problem of organizing interMAP communications are solved by calling system functions described in [4].

3. Resource management of solving field on the operating system level

Used here parallel programming technology assumes permanent existence during task solving its CM in CCU which organizes execution of the task PMs evaluating PM readiness conditions and transmitted them into readiness state with the help of system procedure domodule.

PM completed the work gives control to the subset of its successors, its results are unloaded in CCU data memory and occupied by PM processors become free. Such strategy of resource managing provides avoidance of deadlocks [5]. It was shown in [6,7] that algorithms of the class FBL are sufficiently effective for scheduling both independent and partially ordered tasks. The essence of these algorithms is allocation of fit tasks on free processors. It is accounted that task fits if its requirement to processors is not greater than free processors number. Decision about allocating tasks on the processors are made during task solving whenever any processor becomes free. Operating system realizes the most simple algorithm of the class FBL-algorithm FFBL. It may be represented as follows.

Algorithm FFBL

1. Determine set $\{ P_i \}$ of free processors and its cardinal number μ .
2. Scan the list of ready modules to find such module that its requirement for processors is not greater than μ .
3. If there were not found such modules then stop.
4. Allocate selected module on required number of free processors.

5. Go to #1.

This procedure must be fulfilled each time whenever any processor becomes free and at the beginning of systems work.

CONCLUSION

Thus in the presented work parallel programming technology based on macrotasking and algorithms for resource managing in MS like ES-2703 are considered. Extension of Fortran-77 programming language is used for parallel programming. Chosen here strategy of solving field resource management provides deadlock avoidance and is quite simple in realization. At present time the work on Fortran compiler and operating system design is carried out.

REFERENCES

1. Horoshevsky V.G. Computer systems work engineering analysis.(Radio i Svyaz, 1987), Moscow, 256 p. (in Russian).
2. Guzzi M.D., Padua D.A., Hoeflinger J.R., Lawrie D.H. Cedar Fortran and other vector and parallel Fortran dialects. - Supercomput'88 Proc., Orlando, Flo, Nov. 14-18, 1988, p.114-121.
3. Kalyaev A.V., Makarevich O.B., Nikolaev I.A., Babenko L.K. - Electronnaya vychislitel'naya tekhnika (Computers technology), 1988, v.2, Moscow (in Russian).
4. Babenko L.K., Makarevich O.B., Matveyeva L.N. Principles of computing in the in-line multiprocessor systems. - Lvov: Institute of the UkrSSR Academy of Sciences, 1988, 64 p. (pre-print N 16-88, in Russian).
5. Haberman A.N. Prevention of system deadlock.- Comm.ACM, 1969, v.12. N7, p.373-377.
6. Bakenrot V.J., Chefranov A.G. - Izvestija AN SSSR. Tehnicheskaya kibernetika (Proceedings of the USSR Academy of Sciences. Technical cybernetics), 1985, N4, p.135-148 (in Russian).
7. Chefranov A.G. - Izvestija AN SSSR. Tehnicheskaya kibernetika (Proceeding of the USSR Academy of Sciences. Technical cybernetics), 1985, N6, p.76-78 (in Russian).