Lecture Notes in Computer Science 732

Arndt Bode Mario Dal Cin (Eds.)

Parallel Computer Architectures

Theory, Hardware, Software, Applications

Springer-Verlag Berlin Heidelberg GmbH

Lecture Notes in Computer Science 732
Edited by G. Goos and J. Hartmanis

Advisory Board: W. Brauer D. Gries J. Stoer

www.manaraa.com

Arndt Bode Mario Dal Cin (Eds.)

Parallel Computer Architectures

Theory, Hardware, Software, Applications

Springer-Verlag Berlin Heidelberg GmbH

www.manaraa.com

Series Editors

Gerhard Goos
Universität Karlsruhe
Postfach 69 80
Vincenz-Priessnitz-Straße 1
D-76131 Karlsruhe, Germany

Juris Hartmanis
Cornell University
Department of Computer Science
4130 Upson Hall
Ithaca, NY 14853, USA

Volume Editors

Arndt Bode
Institut für Informatik, TU München
Arcisstr. 21, D-80333 München, Germany

Mario Dal Cin
Institut für Mathematische Maschinen und Datenverarbeitung
Lehrstuhl für Informatik III
Martensstr. 3, D-91058 Erlangen, Germany

CR Subject Classification (1991): C.1-4, D.1, D.3-4, F.1.3

ISBN 978-3-540-57307-4 ISBN 978-3-662-21577-7 (eBook)
DOI 10.1007/978-3-662-21577-7

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag Berlin Heidelberg GmbH
Violations are liable for prosecution under the German Copyright Law.

© Springer-Verlag Berlin Heidelberg 1993
Originally published by Springer-Verlag Berlin Heidelberg New York in 1993

Typesetting: Camera-ready by author

45/3140-543210 - Printed on acid-free paper

www.manaraa.com

Preface

Parallel computer architectures are now going to real applications! This fact is demonstrated by the large number of application areas covered in this book (see section on applications of parallel computer architectures). The applications range from image analysis to quantum mechanics and data bases. Still, the use of parallel architectures poses serious problems and requires the development of new techniques and tools.

This book is a collection of best papers presented at the first workshop on two major research activities at the Universität Erlangen-Nürnberg and Technische Universität München. At both universities, more than 100 researchers are working in the field of multiprocessor systems and network configurations and methods and tools for parallel systems. Indeed, the German Science Foundation (Deutsche Forschungsgemeinschaft) has been sponsoring the projects under grant numbers SFB 182 and SFB 342. Research grants in the form of a Sonderforschungsbereich are given to selected German Universities in portions of three years following a thoroughful reviewing process. The overall duration of such a research grant is restricted to 12 years. The initiative at Erlangen-Nürnberg was started in 1987 and has been headed since this time by Prof. Dr. H. Wedekind. Work at TU-München began in 1990, head of this initiative is Prof. Dr. A. Bode. The authors of this book are grateful to the Deutsche Forschungsgemeinschaft for its continuing support in the field of research on parallel processing.

The first section of the book is devoted to hardware aspects of parallel systems. Here, a number of basic problems has to be solved. Latency and bandwidths of interconnection networks are a bottleneck for parallel process communication. Optoelectronic media, discussed in this section, could change this fact. The scalability of parallel hardware is demonstrated with the multiprocessor system MEMSY based on the concept of distributed shared memory. Scalable parallel systems need fault tolerance mechanisms to garantee reliable system behaviour even in the presence of defects in parts of the system. An approach to fault tolerance for scalable parallel systems is discussed in this section.

The next section is devoted to performance aspects of parallel systems. Analytical models for performance prediction are presented as well as a new hardware monitor system together with the evaluation software.

Tools for the automatic parallelization of existing applications are a dream, but not yet reality for the user of parallel systems. Different aspects for automatic treatment of parallel applications are covered in the next section on architectures and tools for parallelization. Dynamic load balancing is an application transparent mechanism of the operating system to guarantee equal load on the elements of a multiprocessor system. Randomized shared memory is one possible implementation of a virtual shared memory based on distributed memory hardware.

www.manaraa.com

Finally, optimizing tools for superscalar, superpipelined and VLIW(very long instruction word)-architectures already cover automatic parallelization on the basis of individual machine instructions.

The section on modelling techniques groups a number of articles on different aspects of object oriented distributed systems. A distribution language, memory management and the support for types, classes and inheritance are covered. Formal description techniques are based on Petri nets and algebraic specification.

Finally, the section on applications covers knowledge based image analysis, different parallel algorithms for CAD tools for VLSI design, a parallel sorting algorithm for parallel data bases, quantum mechanics algorithms, the solution of partial differential equations, and the solution of a Navier Stokes solver based on multigrid techniques for fluid dynamic applications.

Erlangen and München, March 1993

Arndt Bode
Chairman SFB 342

Hartmut Wedekind
Chairman SFB 182

Mario Dal Cin
SFB 182

www.manaraa.com

Contents

Hardware Aspects of Multiprocessor Systems

Optoelectronic Interconnections
J. Schwider, N. Streibl, K. Zürl (Universität Erlangen-Nürnberg) 1

MEMSY – A Modular Expandable Multiprocessor System
F. Hofmann, M. Dal Cin, A. Grygier, H. Hessenauer, U. Hildebrand,
C.-U. Linster, T. Thiel, S. Turowski (Universität Erlangen-Nürnberg) 15

Fault Tolerance in Distributed Shared Memory Multiprocessors
M. Dal Cin, A. Grygier, H. Hessenauer, U. Hildebrand, J. Hönig,
W. Hohl, E. Michel, A. Pataricza (Universität Erlangen-Nürnberg) 31

Performance of Parallel Systems

Optimal Multiprogramming Control for Parallel Computations
E. Jessen, W. Ertel, Ch. Suttner (Technische Universität München)......... 49

The Distributed Hardware Monitor ZM4 and its Interface to MEMSY
R. Hofmann (Universität Erlangen-Nürnberg) 66

Graph Models for Performance Evaluation of Parallel Programs
F. Hartleb (Universität Erlangen-Nürnberg)............................... 80

Architectures and Tools for Parallelization

Load Management on Multiprocessor Systems
Th. Ludwig (Technische Universität München) 87

Randomized Shared Memory - Concept and Efficiency of a Scalable
Shared Memory Scheme
H. Hellwagner (Siemens AG, München) 102

Methods for Exploitation of Fine-Grained Parallelism
G. Böckle, Ch. Störmann, I. Wildgruber (Siemens AG, München).......... 118

Modelling Techniques

Causality Based Proof of a Distributed Shared Memory System
D. Gomm, E. Kindler (Technische Universität München) 133

Object- and Memory-Management Architecture
- A Concept for Open, Object-Oriented Operating Systems -
J. Kleinöder (Universität Erlangen-Nürnberg) 150

www.manaraa.com

An Orthogonal Distribution Language for Uniform Object-Oriented Languages
M. Fäustle (Universität Erlangen-Nürnberg) ... 166

Towards the Implementation of a Uniform Object Model
F. Hauck (Universität Erlangen-Nürnberg) ... 180

FOCUS: A Formal Design Method for Distributed Systems
F. Dederichs, C. Dendorfer, R. Weber
(Technische Universität München) ... 190

Applications of Parallel Systems

Parallelism in a Semantic Network for Image Understanding
V. Fischer, H. Niemann (Universität Erlangen-Nürnberg) ... 203

Architectures for Parallel Slicing Enumeration in VLSI Layout
H. Spruth, F. Johannes (Technische Universität München) ... 219

Application of Fault Parallelism to the Automatic Test Pattern Generation for Sequential Circuits
P. Krauss, K. Antreich (Technische Universität München) ... 234

Parallel Sorting of Large Data Volumes on Distributed Memory Multiprocessors
M. Pawlowski, R. Bayer (Technische Universität München) ... 246

Quantum Mechanical Programs for Distributed Systems: Strategies and Results
H. Früchtl, P. Otto (Universität Erlangen-Nürnberg) ... 265

On the Parallel Solution of 3D PDEs on a Network of Workstations and on Vector Computers
M. Griebel, W. Huber, T. Störtkuhl, C. Zenger
(Technische Universität München) ... 276

Numerical Simulation of Complex Fluid Flows on MIMD Computers
M. Perić, M. Schäfer, E. Schreck
(Universität Erlangen-Nürnberg) ... 292

Index ... 307

www.manaraa.com

Optoelectronic Interconnections

Johannes Schwider, Norbert Streibl, Konrad Zürl
Physikalisches Institut der Universität Erlangen-Nürnberg
Staudtstr. 7, D-8520 Erlangen, Germany.

1 Bandwidth

The overall performance of data processing machines can be increased in two ways: (i) by using faster system clocks and (ii) by using parallel systems consisting of a multitude of interconnected processing elements. In the near future central aims of information technology are the development of teraflop (10^{12} floating point operations per second) supercomputers and switching networks for telecommunications with terabit bandwidth.

With an acceleration of the system clock alone both of these aims cannot be achieved. A data processing system contains three basic functions: (i) active combining and switching of data, (ii) passive transport of data and (iii) storage of data (which often is implemented by flip-flops, that is by active devices). In quantum-electronics the fastest components are resonant tunneling diodes with a response, measureable for example by nonlinear frequency mixing, beyond 1 THz [Sol 83]. These frequencies belong already to the far infrared region of the electromagnetic spectrum. The simplest circuit, a ring oscillator consisting of two connected active devices in a loop, runs at about 300 GHz [Bro 88]. Today somewhat more complex integrated circuits in GaAs-technology are in research with on the order of 30 GBit/s bandwidth. Still more complex high-speed components, for example multiplexers and demultiplexers for fiber optical links with some 10 GBit/s are commercial. Modern digital telephone exchanges handle data with several hundred MBit/s. The characteristic data rate of a personal computer, determined by the time required for memory access, is on the order of 10 MBit/s. Consequently, one observes the trend summarized in table 1: Although ultrafast devices do exist, complex systems are necessarily slow.

The reason are the fundamental electromagnetic properties of electrical interconnections at high frequencies. If the wavelength of the electromagnetic radiation (associated with the frequency content of the signals to be transported) and the length of the line have similar order of magnitude, a vast variety of problems arises. Efficient screening is required to prevent crosstalk through radiation. Standing waves, reflections and echoes occur unless all lines are correctly terminated (which by the way is expensive in terms of energy). Impedance matching and 'microwave design rules' are required for splitting and joining signal lines. As a consequence, the fastest standard bus system (FutureBus II) supports today only 100 MBit/s per line.

On the other hand, optics is good at communicating informations. Recently, optoelectronical interconnections are an area of active research [Hase 84, Good

www.manaraa.com

Table 1: Complexity and speed of electronic systems

System	bandwidth		number of parts
resonant tunneling devices	3	THz	1
ring oscillator	0.3	THz	2
microwave IC (GaAs)	0.03	THz	several
telecommunications, supercomputer	0.000 3	THz	many
personal computer	0.000 03	THz	cheap

84, Kos 85, Berg 87, Sto 87, Cha 91, Die 92, Bac 92, Par 92]. Optical beams can freely cross through each other without interaction. Optics supports parallel interconnections, either through fiber bundles or through imaging systems. Optocouplers are widely used for isolation and to prevent ground loops. Finally, with state of the art optoelectronic devices the heat dissipation at the beginning and the end of an interconnection can be very small, in contrast to the electronic line drivers, that must be large in order to move necessarily large currents. Thus, the basic impedance matching problem is alleviated by the use of optical interconnects [Mil 89]. Because performance or packing density (or both) in modern electronics are limited by heat dissipation, the use of optoelectronics should yield definite advantages.

Over long distances and at high data rates optical fibers have already supplanted electrical connections. The basic question of the research in optoelectronic interconnections is: how short can optical interconnections be and still offer advantages over electronics? It seem fairly clear, that high performance systems will use optics for connections between modules and boards, lateron maybe even between hybrids and chips. On the other hand, an all-optical computer looks today as a worthwhile but maybe elusive aim of basic research.

2 Parallelism

Basically, parallel interconnections between a number of participants may have different dimensionality: Fibres (or wires) are one-dimensional connections. A printed circuit board, the wiring of gates on a chip or integrated optics offer essentially two-dimensional interconnections. The number of bonding pads at the edge of a chip scales with the linear dimension, the number of gates with the area. Therefore, quasiplanar interconnections cause a bottleneck in a complex system because much fewer connections are available than active devices. A good way out are three-dimensional interconnections through the 3D space above the plane, where the active devices are located. Then the number of interconnections and the number of devices scale essentially in the same way with the area. A numerical example is worthwhile: If each channel requires 300 μm space and a chip or a connector has a overall size of 1 cm, then with two-dimenensional connections we obtain a parallelism of 32 channels/cm, but with three-dimensional connections 32 x 32 = 1024 channels/cm^2. Hence, three-dimensional interconnections support highly parallel systems.

المنارة للاستشارات

www.manaraa.com

Table 2: Parallel interconnection topologies

connection type	#sources	#receivers	example	schematic
ordered point to point	1	1	wire bundle	
random point to point	1	1	switch fabric	
broadcasting, fan–out	1	N	clock distrib.	
fan–in	N	1	interrupt	
bus	N	N	bus system	
reconfigurable	N	N	telephone	

Optical communications is used across long distances, but in these applications usually only one line is necessary. The shorter the distance the higher is the required parallelism. Between subsystems and modules in a computer the interconnections are provided by a bus, which has on the order of 100 parallel lines. Modern chip packages have several hundred pins, hence optical chip to chip interconnections are worthwhile only if a parallelism of on the order of 1000 lines is provided. Optical gate to gate interconnections become interesting only if some 10^4 – 10^6 gates can be 'wired'.

As the degree of parallelism is increased, the data rate of the individual lines decreases: whereas a single fiber optical communications line might run at 20 GBit/s, a bus system should run at the systems clock rate, that is on the order of several 100 MBit/s in a high performance system. Otherwise different clocks must be used within a single subsystem for computing and outside communications, which is not practical in many cases. Also the cost for time multiplexing in terms of gate delays, space, heat dissipation and — last but not least — money is not negligible.

Another important feature that may serve to classify interconnection systems is topology: As shown in table 2 there are a number of different parallel interconnection topologies which are increasingly difficult to implement. The simplest approach are ordered parallel point to point connections. The electronic implementation is a number of parallel wires, optically they may be implemented by a fiber bundle, a fiber ribbon cable or by an imaging system that images an array of light sources onto a receiver array.

Somewhat more complicated are permutation elements, that is random point to point interconections. They allow to change the geometrical order in a bundle of parallel connections. Such permutations are required in many algorithms and therefore in many special purpose machines, for example in sorting and searching and therefore in switching networks and telephone exchanges (packet switch), in fast data transformations such as the fast Fourier transform and therefore in signal processors.

www.manaraa.com

Multipoint interconnection may allow (i) fan-out, i.e. broadcasting of a signal from one source to several receivers, or (ii) fan-in, i.e. the listening of one receiver into the signals transmitted by several receivers, or (iii) the combination of fan-out and fan-in, i.e. the sharing of a common communications line by several participants such as a bus line. Finally, and most complicated, there are reconfigurable interconnections, where the 'wiring' of the participants can be changed. A crossbar or a telephone exchange are examples of such a connection.

For all of these topologies optical implementations have been proposed in the past, see for example [Hase 84, Good 84, Kos 85, Berg 87, Sto 87, Cha 91, Die 92, Bac 92, Par 92], some of which will be presented in the following.

3 Optical backplane

Optoelectronic interconnections between boards (distance x up to 1 m) and integrated circuits (distance x on the order of 1 cm) based on a light guiding plate have been widely studied (for example: Hase 84, Brenn 88, Herr 89, Jah 90, Haum 90, Cha 91, Par 92, Stre 93). They have been proposed to serve as 'optical backplane' or 'optical printed circuit board'. Fig. 1 shows the basic optical setup for one single communication channel, which may be replicated for parallel (multichannel) interconnections.

A thick plate of glass or polymer material is used to guide the optical signals. It may be considered as an extremely multimodal waveguide or simply as free

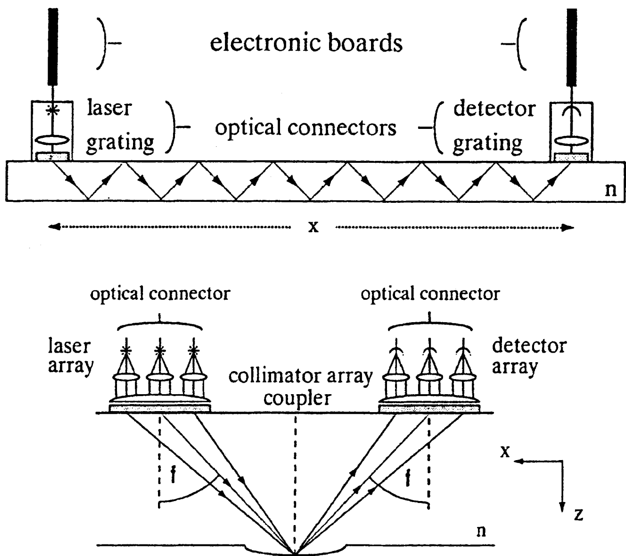


Figure 1: *(a) Principle of a light guiding plate. (b) a curved mirror between the optical connectors on the light guiding plate images the transmitters onto the receivers.*

www.manaraa.com

space 'filled' with transparent material. Its advantage in comparison to free space communication is that it acts as a mechanically well defined base plate for the 'optical connectors' and at the same time protects the optical signals from external disturbances such as dust, air turbulence etc. Its advantage compared to single mode waveguides is that the required tolerances and the alignment problems are much less severe – at least as long as the detectors for the optical signals are not too small.

A collimated beam from a semiconductor laser located on the optical connector is deflected by a grating by a large angle φ, coupled into the light guiding plate, propagates towards the receiver via multiple reflections, is coupled out of the plate by another grating at the second optical connector and detected by the receiver. Holographically recorded volume phase gratings in dichromated gelatine were reported to have excellent coupling efficiency (loss on the order of less than 0.5 dB per coupler for slanted gratings with 45° deflection angle and 740 nm period at 786 nm wavelength) [Haum 90]. Light guiding is achieved with a loss on the order of 0.1 dB/cm. Point to point interconnections thus suffer from losses on the order of 3 – 10 dB, depending on the communication distance.

The simple setup of fig. 1a has two basic drawbacks: firstly, the beam is divergent due to diffraction at the aperture of the optical connector, which severely limits the packing density for parallel channels; secondly, the deflection angle is a function of the wavelength, which has as a consequence tight tolerance requirements for the laser wavelengths. More specifically: For an interconnection length $L = x/sin\varphi$ within a light guiding plate with refractive index n and for light with the wavelength λ in vacuo a beam diameter of at least $d_{min} \approx (L\lambda/n)^{1/2}$ is required in order to avoid excessive spreading of the beam by diffraction. Adjacent parallel channels have to be separated from each other by a multiple of this minimum beam diameter d_{min} in order to avoid crosstalk. Thus, for board distances x of up to 1 m a packing density of not too much more than 10 channels/cm^2 can be implemented with reasonable signal to noise ratio. Such a low packing density is competitive with electronics only at extremely high data rates. Specifically it also prohibits the use of monolithically integrated and therefore densely packed laser and detector arrays.

The chromatic aberration (grating dispersion) makes the deflection angles wavelength dependent and causes problems with 'aiming' the beams at the output couplers. Fabrication tolerances, mode hopping and thermal drift may lead to significant differences in the wavelength of the individual semiconductor lasers. For a deflection angle $\varphi \approx 45^o$, which is preferred in order to eliminate the effects of thermal expansion of the light guiding plate, the distance of the optimum position of the coupler and the wavelength have the same relative deviation $\delta x/x \approx 2\delta\lambda/\lambda$. Hence, the cost for selecting and controling the laser wavelength for interconnection distances of up to 1 m is prohibitive.

Both problems, diffractive broadening as well as chromatic aberration, can be overcome by imaging [Stre 93]. A field lens (or a mirror or a diffractive zone plate) between the optical connectors can be used to image the apertures of the transmitters onto those of the receivers. Fig. 1b shows the principle of such a parallel interconnection. The lasers of a laser array are collimated by a

www.manaraa.com

geometrically similar array of microlenses. An additional lens, whose function may be incorporated into the coupling hologram, images all lasers onto one point of the light guiding plate. There the vertex of the mirror is located that performs the one to one imaging of the apertures of the microlenses and therefore acts as the field lens. At the receiver site a completely symmetrical setup is used to focus down onto the detectors. In practice the light guiding plate may be thinner than shown in fig. 1b, if the light path is folded by multiple reflections as in fig. 1a. For long interconnections a chain of several lenses may be used to relais the image.

Imaging guarantees, that light from each transmitter aperture is focused onto the receiver aperture independently from small errors $\delta\varphi$ in its propagation direction – provided that the field lens is sufficiently large to catch the beam. Therefore the chromatic aberration of the grating couplers is completely eliminated by this design. Also, the optical setup is completely symmetrical, which eliminates all odd order monochromatic aberrations: specifically, the image is free from geometrical distortion, which is important for array operation. Moreover, coma is corrected, which leads us to expect good image quality off axis. Hence, the size of the arrays and thus the number of possible parallel channels may be significant. At the optical connectors the system is 'telecentric', which means that the principal rays for all channels are parallel. Thus only the angular alignment of the connectors is critical, the tolerances for displacement are somewhat less severe. Also, the microlenses are used on axis which reduces aberrations of the focal spots on the detectors. A full design of the optical setup is given in [Stre 90] and includes aberration analysis. It is shown theoretically for all interconnection distances up to 1 m and practically in a feasibility experiment for an interconnection of about 20 cm length, that within a field (= cross section of the optical connector) of 1 cm^2 some 1000 optical channels can be transmitted in parallel.

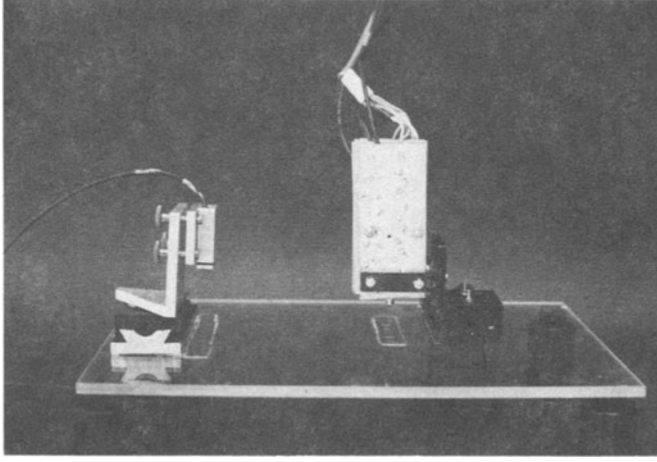


Figure 2: *Experimental setup: Light guiding plate connecting two participants*

www.manaraa.com

4 Optical bus

In a preceding section the difficulties of impedance matching at high data rates were mentioned that are incurred with a bus system having many taps coupling signals in and out a common communications line. Consequently, at high data rates, beyond some 100 MBit/s star couplers, i. e. broadcasting topologies are employed instead of busses which is expensive. Also in much slower systems, such as a multiprocessor, optical implementations of a bus is worthwhile because of synchronicity: optical interconnections easily allow to control propagation delays down to picoseconds and are consequently very well suited for the implementation of global synchronisation modules or clock distribution within a multiprocessor system.

Bus lines are used in electronics to save complexity: instead of wiring N participants with $(N^2 - N)/2$ individual communication lines, they all share one common bus line. The power of each emitter has to be split into N parts for the N listeners, which requires a multiple 1 : N beam splitter. Also each listener must receive power from N different emitters, which might or might not involve a second 1 : N beam splitting component. This beam combination on the receivers involves basic physical questions [Krack 92] regarding the possibility of lossless fan–in in singlemode systems. In a free space optical system an overall theoretical efficiency of 1/(2N-1) compared to the absolute theoretical minimum of 1/N is achieved, if only one single beamsplitter is used for transmitters and receivers simultaneously [Krack 92]. Fig. 3 shows an optical implementation involving a Dammann grating [Damm 71] or a similar phase-only diffraction grating as multiple beam splitter.

Each participant in a bus line has optical transmitters and receivers, and a fiber link transporting the optical signals from the electronic board to an all–

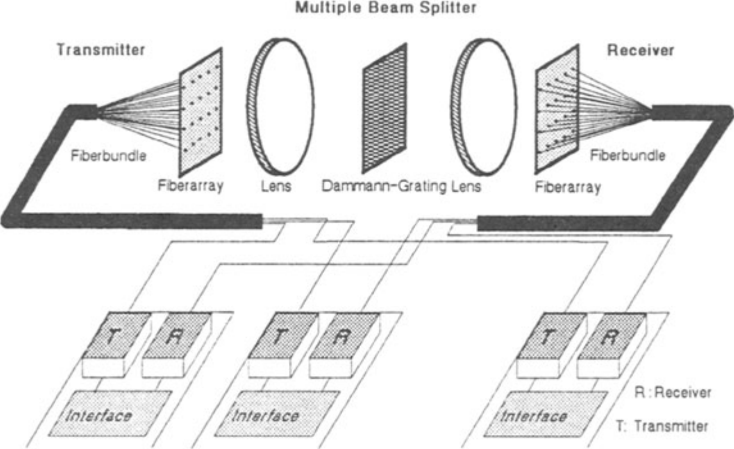


Figure 3: *Principle of an optical bus.*

www.manaraa.com

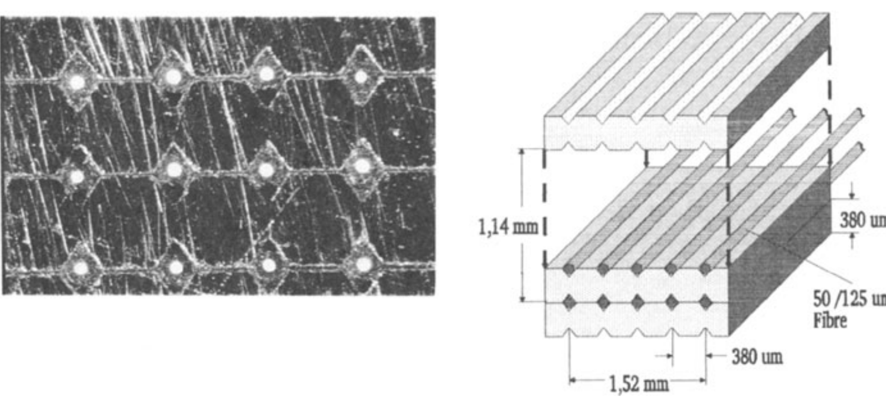


Figure 4: *Fiber end plate for coupling fibers to an optical interconnect stage.*

optical interconnection stage. The fibers end in a regular array in a fiber end plate. Fig. 4 shows a photograph of such a device. It was fabricated by wet chemical etching of V-grooves on both sides of silicon wafers in 100 orientation by using KOH. These grooves can be applied very accurately with microlithographic precision. In a selfaligned assembling procedure alternately a layer of fibers and a grooved silicon plate are stacked onto each other. All components are glued together and the fiber end plate is polished. In this way an accurate array of fibers can be fabricated.

The fibers allow a mechanically flexible coupling of many participants into a highly accurate optical system without putting tight geometrical constraints on the construction of the electronic system. Additionally it concentrates all signals coming from different, possibly widely spaced participants within a tight volume which can be handled with a compact optical system: for example the end plate could have a size of 1 cm^2. Finally the fiber concentrator puts all the signals on a well defined place in the plate with small tolerances. By using a telecentric one to one imaging system the end plates are imaged onto each other. Without the multiple beam splitter this imaging system would implement an ordered point to point connection. However, in the filter plane a multiple beam splitter such as a Dammann grating or a similar device is inserted, whose optical function is shown in fig 5.

Such a grating has rectangular surface corrugations that are fabricated by microlithographic methods. The component shown in fig. 5 was designed by using nonlinear optimization, its structure was plotted by using a laser beam writing system and it was etched into a fused silica substrate by using reactive ion etching [Hasel 92]. By performing a diffraction experiment it can be seen that one incoming beam of light is split into a multitude of beams, that is the grating performs a fan-out. Similarly, since the optical system in fig. 4 is completely symmetric the multiple beam splitter performs at the same time a fan–in function. Consequently, at each outgoing fiber of the second endplate signals of several input channels are superimposed, as well as the signal emerging from

www.manaraa.com

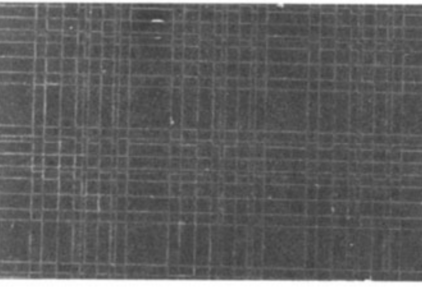
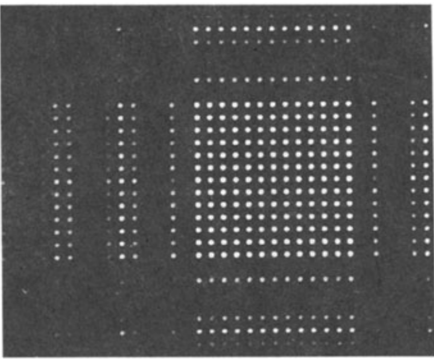



Figure 5: *Surface structure and diffraction pattern from a Dammann grating.*

each incoming fiber is distributed to several listeners. The grating implements a bus topology and serves as fan–out and fan–in element at the same time.

5 Optical switching

In telecommunications as well as in multiprocessors a problem of central importance is setting up reconfigurable interconnections. A switching matrix for N participants implemented as a crossbar requires N^2 switches, as well as the ability of each line to drive N switches ($\approx$ fan–out). In spite of this scaling, which makes crossbars attractive only for a moderate number of participants optical implementations have been proposed [Sto 87]. Massively parallel systems, such as a telephone exchange, require to build up complexity by combining a multitude of (simple) components. Such a switching fabric is schematically shown in fig 6.

Such multistage networks require much less switches than a crossbar, in some cases O(N log N), and have been widely studied [Feng 81]. Typically they consist of several stages with small crossbars (in the minimalistic case of the so called shuffle exchange network 2 x 2 crossbars are used) and permutation elements implementing a random wiring in between them. These global interconnections between stages are fairly long and densely packed lines which are subject to crosstalk and therefore well suited for optical replacements. Holographic optical elements have been widely discussed for this purpose [Schw 92], fig. 7 shows the principle.

In a demonstration setup for an optoelectronic switching network at the University of Erlangen holographic permutation elements were used in conjunction with 'smart detectors'. A minimal network consisting of three subsequent stages with four parallel channels was implemented. From a PC, which served to generate the transmitted data and check the received data for bit errors, an array of lasers was driven to inject optical signals into the system. In the demonstration discrete lasers were used and by a geometrically similar array of microlenses collimated. In the near future it can be expected that monolithically integrated

المنارة للاستشارات
www.manaraa.com

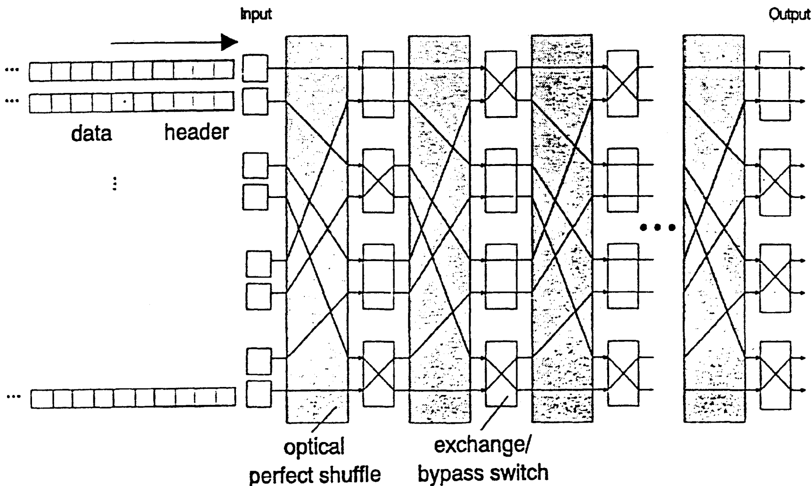


Figure 6: *Multistage switching network (alternating permutation/switch stages).*

arrays of microlasers become available [Cold 92]. The parallely collimated laser beams pass a holographic permutation element realized as volume holograms in dichromated gelatine. This components implements firstly the perfect shuffle, i. e. the global interconnection pattern between stages, and secondly it changes the cross–section by reducing the distances between adjacent channels. Thus the (large) pitch of the discrete laser array is matched to the (420 μm small) pitch of the receiver.

As a receiver an opto-ASIC, i. e. a custom designed optoelectronic CMOS integrated circuit [Zürl 92], was used as a 'smart detector'. It contains photodiodes, analog electronic amplification, digital circuits for implementing 2 x 2 crossbars and line drivers for electronic output. The packet switching network was configured as a self-routing Batcher sorting network. This requires each 2 x 2 crossbar to be able to extract its switch setting from the incoming data. For this purpose each node has a small finite state machine. Thus, the network does not require central control. The photodiodes on the opto-ASIC are an interesting part, because they do not belong to a standard CMOS process. Nevertheless a good responsivity of 0.28 A/W can be achieved. After receiving and switching the signals are transmitted into the next optical permutation element by means of the next laser array. The complete setup is shown in fig. 8

The demonstration was running at 1 MBit/s (limited by the PC-interface for data generation and measurement of bit error rate). It was possible to set arbitrary non-blocking signal paths up and to reconfigure them.

In the long term light emitters (laser diode arrays) and receivers ('smart detectors') should be mounted as near to each other as possible. Obviously, today this can be achieved by building hybrid setups. Suitable projects are in research [Bac 92].

www.manaraa.com

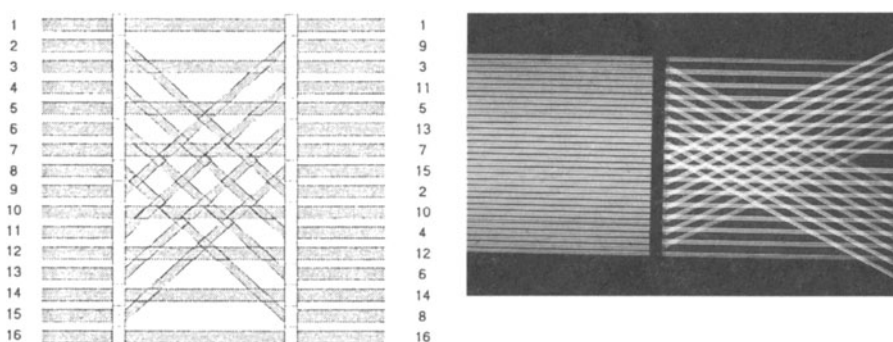


Figure 7: *Holographic permutation elements (a) principle (b) experiment with beams passing a facetted volume hologram visualized by fluorescence.*

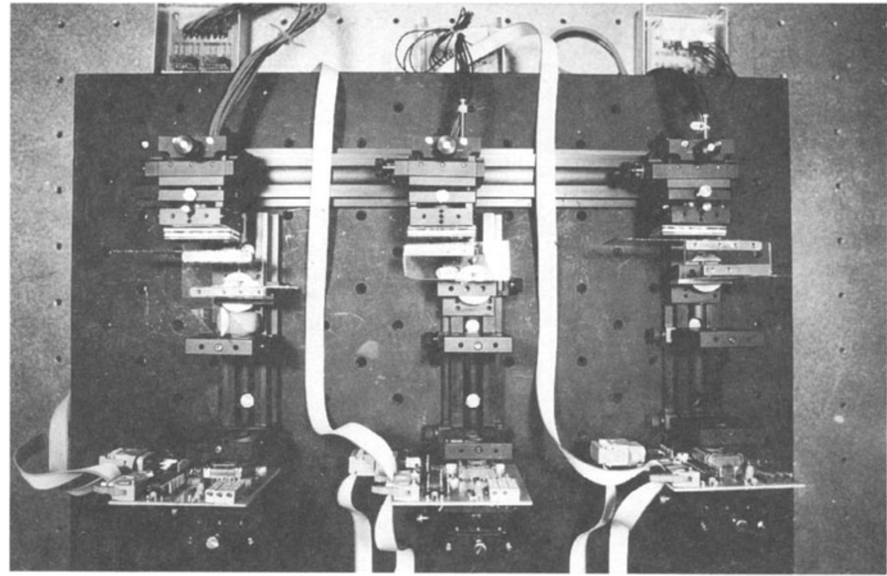


Figure 8: *Experimental setup for a selfrouting multistage shuffle-exchange network with three subsequent stages and four parallel channels.*

In the future it is hoped that light emitters, receivers and digital electronic can be integrated on the same common substrate in a monolithic fashion. Such so-called 'smart pixels' would be a major breakthrough. The philosophy of smart pixels is to have many small electronic processing 'islands' having optical I/Os with optical connections between them. Optoelectronic switches, such as the one shown above, could be implemented in a compact fashion as chip to chip interconnections. By adding functionality to the switching nodes, very

المنارة للاستشارات
www.manaraa.com

soon powerful processors can be envisioned. For example with the architecture similar to a multistage network, simply by giving each node the ability to add incoming numbers and to multiply them by a factor, a special purpose signal processor can be built. It could be used for Fourier transformation and other fast algorithms. In a similar approach (but with a different wiring diagram) other massively parallel computer architectures such as systolic arrays or cellular automata could be implemented optoelectronically by using smart pixels [Fey 92].

6 Conclusion

Optoelectronic interconnections are useful in short range interconnections, for example within multiprocessor systems, for several reasons:

- high bandwidth of each individual channel (in excess of 1 GBit/s)
- high parallelism and packing density (in excess of 1000 channels/cm^2)
- three-dimensional topology and global interconnection patterns
- broadcasting, signal distribution and bus topology at high speed
- synchronisation because all delays are exactly known
- no crosstalk along the line (only at the optoelectronic terminals)
- isolation prevents ground loops
- high impedance devices reduce heat dissipation for communications
- hopefully: integration with electronics towards 'smart pixels'

Several examples for optoelectronic interconnections, namely an optical backplane, an optical bus and reconfigurable optoelectronic switches were presented. This survey was by no means complete but had a more exemplary character.

In the literature there is not much doubt, that optical interconnections will soon be useful within distributed systems, in multiprocessors and in telecommunications. On the other hand, all-optical computers still require some major breakthrough in optical switching devices or logic gates. As Chavel [Chav 91] puts it: *'The only reason we can see at present to talk about an optical computer is not that anyone might need it or that there is a visible advantage to it, but rather that nobody knows how much may still be expected from progress in nonlinear optics and technology; the only reason we can see not to provide optical interconnects to computers, at least at the board to board level is, that electronic has a well established interconnect technology and optics does not, so that meaningful practical issues like ruggedness, alignment tolerances have not yet been adequately worked out.'*

www.manaraa.com

References

[Bac 92] E.-J. Bachus, F. Auracher, O. Hildebrandt, B. Schwaderer: 'An account of the German national joint programme Photonik/Optical Signal Processing', Proc. European Conference on Optical Communication ECOC'92, Berlin 27.09.–01.10.92, VDE–Verlag, ISBN 3 – 8007 – 1897 – 9.

[Berg 87] L. A. Bergmann, W. H. Wu, R. Nixon, S. C. Esener, C. C. Guest, T. J. Drabik, M. Feldman, S. H. Lee: 'Holographic optic interconnects for VLSI', Opt. Eng. 25, 1109 (1986).

[Brenn 88] K.-H. Brenner, F. Sauer: 'Diffractive-reflective optical interconnects', Appl. Opt. 27, 4251 (1988).

[Bro 88] E.R. Brown et. al.: 'High speed resonant tunneling diodes', Proc. SPIE 943, 2 (1988).

[Cha 91] P. Chavel and J. Taboury: 'On alleged and real advantages of optical interconnects: examples', Annales de Physique, Colloque No. 1, supplement No. 1, vol. 16, pp. 153, (1991).

[Cold 92] L.R. Coldren: 'Vertical cavity laser diodes', Proc. European Conference on Optical Communication ECOC'92, Berlin 27.09.–01.10.92, VDE–Verlag, ISBN 3 – 8007 – 1897 – 9.

[Damm 71] H. Dammann, K. Görtler: 'High efficiency in-line multiple imaging by means of multiple phase holograms', Opt. Comm. 2, 312 (1971).

[Die 92] R. Diehl: 'Cooperative effort on optical interconnects in the German national program 'photonics' ', Proc. European Conference on Optical Communication ECOC'92, Berlin 27.09.–01.10.92, VDE–Verlag, ISBN 3 – 8007 – 1897 – 9.

[Feng 81] T. Feng: 'A survey of interconnection networks', IEEE Comp. Dec. 1981, p. 12-27.

[Fey 92] D. Fey: 'Modellierung, Simulation und Bewertung digitaler optischer Systeme', Dissertation, Universität Erlangen–Nürnberg 1992.

[Good 84] J.W. Goodman, F.J. Leonberger, S.Y. Kung, R.A. Athale: 'Optical Interconnections for VLSI systems', Proc. IEEE 72, 850 (1984).

[Hase 84] Hase K. R.: 'Ein Beitrag zur Realisierung rechnerinterner optischer Bussysteme mit planaren Lichtleitern', Dissertation Universität Duisburg 1984.

[Hasel 92] S. Haselbeck et. al.: 'Synthetic phase holograms written by laser lithography', Proc. SPIE 1718 (1992).

www.manaraa.com

[Haum 90] H.J. Haumann et. al.: 'Optoelectronic interconnection based on a light guiding plate with holographic coupling elements', Opt. Eng. 30, 1620.

[Herr 89] J. P. Herriau, A. Deboulbe, P. Maillot, P. Richin, L. d'Auria, J. P. Huignard: 'High speed interconnections - analysis of an optical approach', Int. Symp. Optics in Computing Toulouse, France 1989.

[Jah 90] J. Jahns, A. Huang: 'Planar integration of free space optical components', Appl. Opt. 28 (1990).

[Kos 85] R. K. Kostuk, J. W. Goodman, L. Hesselink: 'Optical imaging applied to microelectronic chip to chip interconnections', Appl. Opt. 24, 2851 (1985).

[Krack 92] U. Krackhardt, F. Sauer, W. Stork, N. Streibl: 'Concept for an optical bus-type interconnection network', Appl. Opt. 31, 1730 (1992).

[Par 92] J. W. Parker: 'Optical interconnections for electronics', Proc. European Conference on Optical Communication ECOC'92, Berlin 27.09.–01.10.92, VDE–Verlag, ISBN 3 – 8007 – 1897 – 9.

[Schw 92] J. Schwider et. al.: 'Possibilities and limitations of spacevariant holographic optical elements for switching networks and general interconnects', Appl. Opt., accepted 1992.

[Sol 83] T. Sollner et. al.: 'Resonant tunneling through quantum wells at 2.5 THz', Appl. Phys. Lett. 43, 588 (1983).

[Sto 87] W. Stork: 'Optical crossbar', Optik 76, 173 (1987).

[Stre 93] N. Streibl, R. Völkel, J. Schwider, P. Habel, N. Lindlein: 'Parallel optoelectronic interconnections with high packing density through a light guiding plate using grating couplers and field lenses', Opt. Commun. submitted 1992.

[Zürl 92] K. Zürl, N. Streibl: 'Optoelectronic array interconnections', Opt. Quant. Electron. 24, 405 (1992).

المنارة للاستشارات
www.manaraa.com

MEMSY
A Modular Expandable Multiprocessor System

F. Hofmann
hofmann@informatik.uni-erlangen.de
M. Dal Cin, A. Grygier, H. Hessenauer, U. Hildebrand,
C.-U. Linster, T. Thiel, S. Turowski

University of Erlangen-Nürnberg
IMMD, Martensstraße 1/3
D-W 8520 Erlangen, Germany

Abstract. In this paper the MEMSY experimental multiprocessor system is described. This system was built to validate the MEMSY architecture - a scalable multiprocessor architecture based on locally shared-memory and other communication media. It also serves as a study of different kinds of application programs which solve a variety of real problems encountered in scientific research.

1 Introduction

Among the different kinds of multiprocessor systems, those with global shared-memory are normally the ones most liked by application programmers because of their simple programming model. Closer examination of typical problems reveals that, for a broad class of these problems, global shared-memory is not what is really needed by the application. Local shared data is sufficient to solve the problems.

Multiprocessors with global shared-memory all suffer from a lack of scalability. By making clever use of fast buses and caching techniques this effect may be postponed, but each system has an upper limit on the number of processing nodes.

Our MEMSY system is now an approach towards a scalable MIMD multiprocessor architecture which utilizes memory shared between a set of adjacent nodes as a communication medium. We refer to this kind of shared-memory as *distributed shared-memory*.

The MEMSY system shall continue the line of systems which have been built at Erlangen using distributed shared-memory. The basic aspects of this architecture are described in [2] and [3].

2 MEMSY Architecture

2.1 Design Goals

The MEMSY architecture was defined with the following design goals in mind:

- economy The system should be based on off-the-shelf components we can buy; only some parts should need to be designed and built by us.

www.manaraa.com

- scalability The architecture should be scalable with no theoretical limit. The communication network should grow with the number of processing elements in order to accommodate the increased communication demands in larger systems.
- flexibility The architecture should be usable for a great variety of user problems.
- efficiency The system should be based on state-of-the-art high performance microprocessors. The computing power of the system should be big enough to handle real problems which occur in scientific research.

2.2 Topology of the MEMSY System

The MEMSY structure consists of two planes. In each plane the processor nodes form a rectangular grid. Each processor node has an associated shared-memory module, which is shared with its four neighbouring processor nodes. The grid is closed to a torus.

One processing element of the upper plane has access to the shared-memory modules of the four processing elements directly below it, thereby forming a small pyramid. There are four times as many processing elements in the lower plane than in the upper.

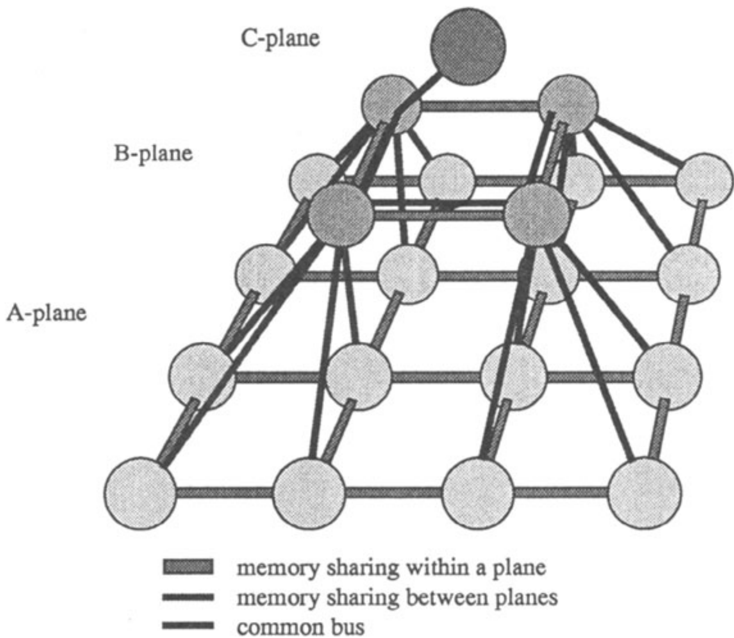


Fig. 2.1 Topology of the MEMSY system (Torus connections are missing)

On top of the whole system there is an optional processor which may serve as a front-end to the system.

www.manaraa.com

The basic idea behind this structure is this: the lower plane does the real work and the upper plane feeds the lower plane with data and offers support functions.

A subset of the processor nodes has access to one or more common bus systems to allow communication over greater distances or broadcasts.

3 Hardware Architecture of MEMSY

The experimental memory-coupled multiprocessor system MEMSY consists of three functional units:

- 4 + 16 processor nodes,
- one shared-memory module at each node, which is called the *communication memory* and
- the interconnection network which provides the communication paths between processor nodes and communication memories.

In addition to these essential units which are described in the following sections, an FDDI net, a special optical bus and a distributed interrupt coupling system are integrated in the system. The FDDI net allows testing and use of another communication media. The optical bus is designed to support various global synchronisation mechanisms. The interrupt coupling system establishes interrupt connections between every two immediate-neighbour nodes which share a communication memory.

3.1 Processor Nodes

Each node of the MEMSY system is identically designed and consists of a Motorola multiprocessor board MVME188 and additional hardware, some of which were designed and implemented as part of the MEMSY project. In figure 3.1, which shows the logical node structure, these parts have a lighter background colour.

The MVME188 board used in the MEMSY system consists of four VME modules. These are; the system controller board, holding e.g. timers and serial interfaces; two memory boards, each holding 16M bytes local memory; and the main logic board, carrying the processor module.

The processor module comprises four MC88100 RISC CPUs, which have multiple internal parallelism, and eight MC88204 cache and memory management units (CMMU), which provide 8*64K bytes cache memory. Special features of the processor module are:

- Cache coherency is supported by the hardware.
- There exists a cache copyback mode which writes data back to memory only if necessary and a write-through mode.
- There exists an atomic memory access which is necessary for efficiently implementing spinlocks and semaphores in a multiprocessor environment.
- The caches provide a burst mode which allows atomic read/write access of four consecutive words[1] while supplying only one address.

www.manaraa.com

The architecture of the processor module, the MVME188 board and the M88000 RISC product[2] are described in greater detail in [9].

All VME modules mentioned above, are interconnected by a high speed local bus. Each node can be extended with additional modules via this local bus or the VME bus. The communication memory interface is attached to the local bus. Its function is to recognize and execute accesses to the communication memories. The interface hardware provides three ports to which the communication memories are connected either directly or via an interconnection network. This interconnection network is described in section 3.2.

To the M88000 the memory interface looks like a simple memory module. The address decoder of the MVME188 is configured in such way, that the highest two address bits determine whether the address space of the local memory boards, of the VME bus or of the memory interface is accessed. In case of the memory interface, the next two address bits determine the port which should be used for each memory access. Four further bits of the address determine the path through the coupling unit and which communication memory is to be accessed.

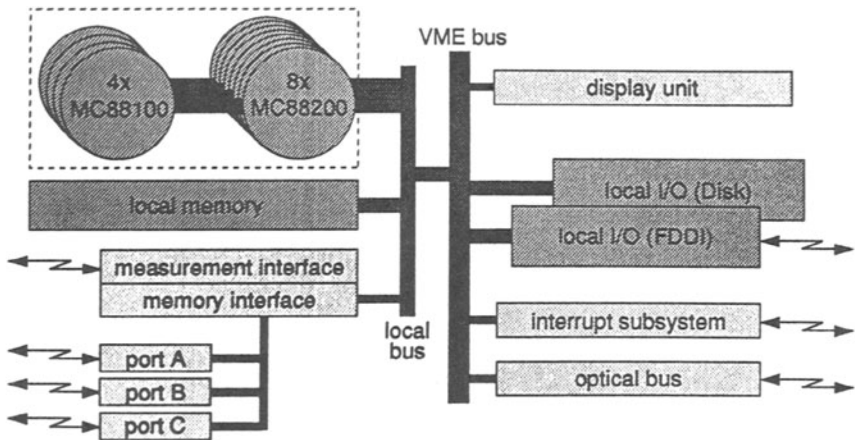


Fig. 3.1 Logical node structure

Addresses and data are transferred in multiplexed mode. The connection is 32 data bits plus four parity bits wide for the address or data transfer. The parity bits are generated by the sender and checked by the receiver. If the communication memory detects a parity error in address or data, it generates an error signal, otherwise a ready signal is generated. If the memory interface receives an error signal or detects a parity error during a read access it transmits an error signal to the M88000, otherwise a ready signal is sent.

1. A word is always 32 bit wide.
2. The combination of MC88100 and MC88204 is referred to as the M88000 RISC product.

www.manaraa.com

The memory interface hardware supports the atomic memory access and the described burst mode. Counters have been included in the interface hardware to count the various types of errors in order to investigate the reliability of the connection. The counters can be read and reset by a processor. In addition there is a status register which contains information about the last error occurred. This can be used in combination with an error address register to investigate the error.

In addition, the memory interface contains a measurement interface to which an external monitor can be connected. A measurement signal is triggered by a write access to a particular register of the memory interface and the 32-bit word written is transferred to the monitor. This enables the use of hybrid monitors for measurements on MEMSY. Refer to [5] for further information on this topic.

3.2 The Interconnection Network

The topology of MEMSY is described in section 2.2. Each node has access to its own communication memory and to the communication memories of the four neighbouring nodes. Additionally, every node of the B-plane has access to the communication memories of its four assigned A-plane-nodes.

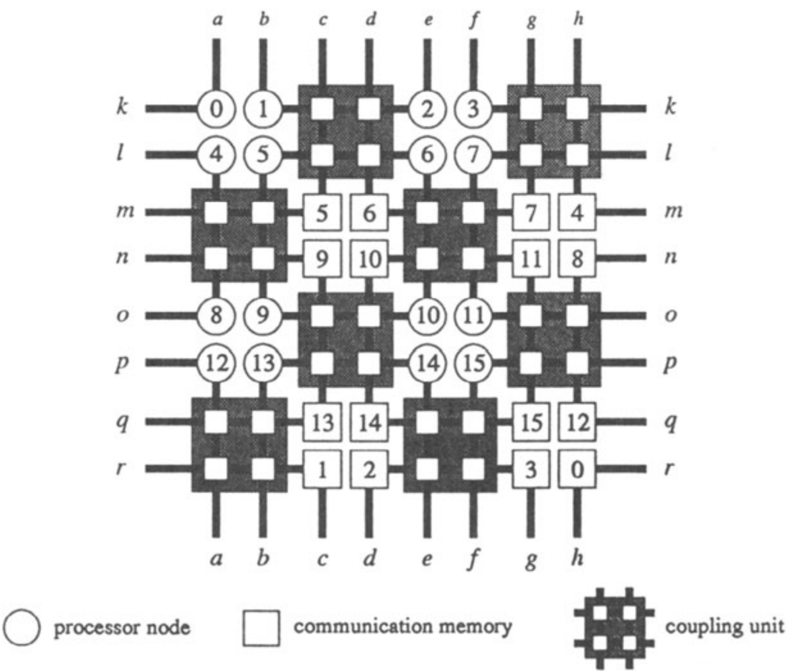


Fig. 3.2 Composition of processor nodes, communication memories and coupling units

المنارة للاستشارات
www.manaraa.com

A static implementation of this topology requires up to 9 ports at each node and up to 6 ports at each communication memory. To reduce this complexity and the number of interconnections, a dynamic network component, called *coupling unit*, has been developed. The use of the coupling unit reduces the number of ports needed at the memory interface and the communication memory to three. Only two of these ports are used for the connections within a plane. The coupling unit supports the virtual implementation of the described MEMSY topology.

The coupling unit is a blocking, multistage, dynamic network with fixed size which provides logically complete interconnection between 4 input ports and 4 output ports. The interconnection structure of MEMSY is a hybrid network with global static and local dynamic network properties.

The torus topology of a single MEMSY plane is implemented by the arrangement of nodes, communication memories, and coupling units as shown in figure 3.2. For reasons of complexity the local dynamic network component is not depicted in this figure. It is described in more detail in the next section.

Each node and each memory module is connected to two coupling units. Thus the nearest-neighbour torus topology can easily be established. A square torus network with $N=n^2$ nodes requires N/2 coupling units. The connections from the nodes of the B-plane to the four corresponding communication memories of the A-plane are also implemented by using coupling units. These are connected to the third ports.

Internal Structure of the Coupling Unit. The hardware component used to implement a multiprocessor system, as described above, is shown in figure 3.3. In our implementation of the interconnection network, accesses to the communication memories via coupling units are executed with a simple memory access protocol. The interconnection network operates in a circuit-switching mode by building up a direct path for each memory access between a node and a communication memory.

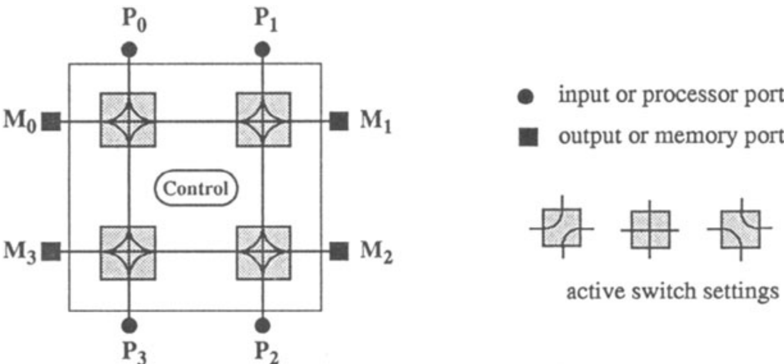


Fig. 3.3 Internal structure of a coupling unit

www.manaraa.com

A coupling unit consists of the following subcomponents:

- 4 p-ports which allow the access to the coupling unit from nodes
- 4 m-ports which provide the connection of communication memories
- 4 internal subpaths which perform data transfer within the coupling unit
- 1 control unit which controls the dynamic interconnection between p-ports and m-ports
- 4 switching elements which provide the dynamic interconnection of p-ports and m-ports

The structure of the p-ports and m-ports is basically identical to a memory interface with a multiplexed 32 bit address / data bus. The direction of the control flow is different for p-ports and m-ports. An activity (a memory access) can be only initiated at a p-port.

The control unit is a central component within the coupling unit. It always has the complete information about the current switch settings of all switching elements. If a new request is recognized by receiving a valid address, the control unit can decide at once whether the requested access can be performed or has to be delayed. For any access pattern the addressed memory port and all necessary internal subpaths are available when all switching elements contained in the communication path to be built-up are either inactive or possess exactly the switch settings required for the establishment of the interconnection.

The necessary switch settings of all required switching elements are fixed a priori for every possible access pattern. The decision about the performability of a requested access is made by comparing the required switch settings with the current ones.

It can be seen from the structure of the coupling unit that two different communication paths containing disjoint sets of internal subpaths can be selected for a memory access. This results from the arrangement of the switching elements in a ring configuration interconnected by the internal subpaths. The decision as to which of the possible communication paths is to be established is made dynamically according to the current switch settings. If possible, the communication path requiring fewer internal subpaths is chosen to minimize the propagation delay caused by the switching elements.

The feature of the coupling unit which allows alternative communication paths is important in the context of fault tolerance. This is discussed in [1].

Performance of the Interconnection Network. An access to shared data in the communication memories requires a significantly higher access time than an access within the node. In addition to the fact that the reduction in access time caused by using a cache is generally no longer possible, the longer transfer paths and the execution of control mechanisms cause further delays. The sequentialization which can be required when conflicts occur either in the communication memories or in the coupling units can cause additional waiting times. The memory access time in our implementation is normally 1 μs and up to 1.3 μs if blocking occurs due to a quasi-simultaneous access.

www.manaraa.com

Since only data which is shared by nodes is held in the communication memories, such as boundary values of subarrays, the increased access time has only a small influence on the overall computing time. Measurements made using the test system INES [4] specially developed to measure the performance of the coupling hardware show that a high efficiency can be achieved under realistic conditions. Thus reducing the complexity of the network by using coupling units causes only a small reduction in performance compared to a static point to point network.

4 Programming Model

The programming model of the MEMSY system was designed to give the application programmer direct access to the power of the system. Unlike in many systems, where the programmer's concept of the system is different from the real structure of the hardware, the application programmer for MEMSY should have a concept of the system which is very close to its real structure. In our opinion this enables the programmer to write highly efficient programs which make the best use of the system.

In addition, the programmer should not be forced to a single way of using the system. Instead, the programming model defines a variety of different mechanisms for communication and coordination[3]. From these mechanisms the application programmer may pick the ones which are best suited for his particular problem.

The programming model is defined as a set of library calls which can be called from C and C++. We choose these languages for the following reasons:

(1) Only languages which have some kind of a 'pointer' make it possible to implement the routines, which access the shared-memory, as library calls. Otherwise costly extensions to the languages would have been needed.
(2) The C compiler is available on every UNIX system. As it is also used to develop the operating system itself, more effort is taken by the manufacturer to make this compiler bug-free, stable and have it generate optimized code.
(3) Compared to programs using index references, programs using pointer references can lead to more efficient machine code.

The MEMSY system allows different applications to run simultaneously. The operating system shields the different applications from one another.

To make use of the parallelism of each processing unit, the programmer must generate multiple processes by the means of the 'fork' system call.

4.1 Mechanisms

The following sections introduce the different mechanisms provided by the programming model.

---

3. We use the term *coordination* instead of *synchronization* to express that not the simultaneous occurring of events (e.g. accesses to common data structures) is meant but their controlled ordering.

www.manaraa.com

Shared-Memory. The use of the shared-memory is based on the concept of 'segments', very much like the original shared-memory mechanism provided by UNIX System V. A process which wants to share data with another process (possibly on another node) first has to create a shared-memory segment of the needed size. To have the operating system select the correct location for this memory segment, the process has to specify with which neighbouring nodes this segment needs to be shared.

After the segment has been created, other processes may map the same segment into their address space by the means of an 'attach' operation. The addresses in these address spaces are totally unrelated, so pointers may not be passed between different processes. The segments may also be unmapped and destroyed dynamically.

There is one disadvantage to the shared-memory implementation on the MEMSY system. To ensure a consistent view over all nodes the caches of the processors must be disabled for accesses to the shared-memory. But the application programmer may enable the caches for a single segment if he is sure that inconsistencies between the caches on different nodes are not possible for a certain time period. The inconsistencies are not possible if only one node is using this segment or if this segment is only being read.

Messages. There are two different message mechanisms which are offered by the programming model: the one described in this section and the one named 'transport', described later.

This message mechanism allows the programmer to send short (2 word) messages to another processor. The messages are buffered at the receiving side and can be received either blocking or non-blocking. They are mainly used for coordination. They are not especially optimized for high-volume data transfer.

Semaphores. To provide a simple method for global coordination, semaphores have been added to the programming model. They reside on the node on which they have been created, but can be accessed uniformly throughout the whole system.

Spinlocks. Spinlocks are coordination variables which reside in shared-memory segments. They can be used to guard short critical sections. In contrast to the other mechanisms this is implemented totally in user-context using the special machine instruction 'XMEM'. The main disadvantage of the spinlocks is the 'busy-wait' performed by the processor. This occurs if the process fails to obtain the lock and must wait for the lock to become free. To minimize the effects of programming errors on other applications, a time-out must be specified, after which the application is terminated (there is a system-imposed maximum for this time-out).

www.manaraa.com

Transport. The transport mechanism was designed to allow for high volume and fast data transfer between any two processors in the system. The operating system is free to choose the method and the path this data is to be transferred on (using shared-memory, FDDI-ring or bus). It can take into account the current load of the processing elements and data paths.

I/O. Traditional UNIX-I/O is supported. Each processing element has a local data storage area. There is one global data storage area which is common to all processing nodes.

Parallelism. To express parallelism the programmer has to create multiple processes on each processing element by a special variant of the system call 'fork'.

Parallelism between nodes is handled by the configuration to be defined: One initial process is started by the application environment on each node that the application should run on. In the current implementation these processes are identical on all nodes.

Information. The processes can obtain various information from the system regarding their positions in the whole system and the state of their own or other nodes.

4.2 Development

The programming model is open for extensions which will be based on experiences we gain from real applications. Specific problems will show whether additional mechanisms for communication and coordination are needed and how they should be defined.

5 Operating System Architecture

Various considerations have been made as to which operating system should be chosen. Basically there are two choices:

- Design and implement a completely new operating system or
- use an existing operating system and adapt it to the new hardware.

By designing a new operating system the whole hardware can be completely integrated and supported. Better fitting concepts than those found in existing implementations can be developed. But it must not be underestimated, that implementing a new operating system requires a lot of time and effort. The second choice offers a nearly ready-to-run operating system, in which only the adaptions to the additional hardware have to be made.

For MEMSY the second choice was taken and Unix was chosen as the basis for MEMSOS, the operating system of MEMSY. Unix supplies a good development environment and many useful tools. The multitasking / multiuser feature of Unix is included with no additional effort.

www.manaraa.com

On each processor node we use the UNIX SYSTEM V/88 Release 3 of MOTOROLA, which is adapted to the multiprocessor architecture of the processor board. The operating system has a peer processor architecture, meaning that there is no special designated processor, e.g. master processor. Every processor is able to execute user code and can handle all I/O requests by itself. The kernel is divided into two areas. One area contains code that can be accessed in parallel by all processors, because there is either no shared data involved or the mutual exclusion is achieved by using fine grain locks. The second area contains all the other code that can not be accessed in parallel. This area is secured with a single semaphore. For example, all device drivers can be found here. In SYSTEM V/88 the usual multiprocessor concepts are implemented, such as message-passing, shared-memory, interprocessor communication and global semaphores. See [8] for more details.

5.1 Extensions

For the implementations of the above mentioned multiprocessor concepts the assumption has been made that all processors share a global main memory. But the operating system is not able to deal with distributed memory such as our communication memory. Therefore certain extensions and additions have been made to the operating system. Only little changes have been made to the kernel itself. Standard Unix applications are runnable on MEMSY because the system-call interface stayed intact.

Integration of the additional hardware, particularly the communication memories and the distributed interrupt-system, was one of the first steps. One of the next steps made was the implementation of basic mechanisms for all sorts of communication and coordination, which depend on the shared-memory. On top of these mechanisms most of our other extensions are built. The Unix system-call interface was extended by additional system calls, as described in section 4.1.

In the following sections only some of the extensions are described. Our concept for support of user programs and a hierarchy of communication mechanisms using the distributed shared-memory is introduced.

Support of Distributed User Programs. Various demands on high-performance multiprocessor systems are made by the users. A system should be highly available and easy to use. There should be as little interference with other user programs as possible and the computing power should always be the maximum available. For their programs users demand the usual, or even an enlarged functionality, short start-up times and the possibility of interactive testing.

The Application Concept. In MEMSOS most of the users' needs are supported by the realization of our *application concept*. We define the set of all processes belonging to one single user program as an *application*. An application can be identified by a unique *application number*. Different applications running on MEMSY are distinguishable by that number. Single application processes, called *tasks*, inherit the

www.manaraa.com

application number and are assigned a *task number*, which is a serial number unique for this application and processor node. So an application task can be identified by its application number, task number and node number.

At system initialization time one outstanding application is created. This initial application, the *master application*, is made up of application daemons running on each node and a single master application daemon, which may be distributed. All daemons communicate with each other. It is the purpose of these daemons to create the user applications on demand and to keep track of them. For each new application the master daemon allocates a free, unique application number. As the first task of each application the application *leader task* is started on each node by the other daemons. The leader task creates the application environment and all subsequent tasks. It supervises the execution of the application tasks on that node it is running on and communicates with all leader tasks of the same application. The leader tasks act as agents to the master application.

This simple application concept makes it possible to easily control and monitor distributed user programs. Because single applications can be distinguished from one another, more than one application can be allowed to run in parallel on MEMSY. By changing the processor binding and process/task scheduling more efficiency can be achieved.

Processor Binding and Process Scheduling. In the SYSTEM V/88 operating system each processor can have its own assigned process run queue which it prefers to use. Each Unix process is bound to a certain run queue. During a process switch the binding can be changed. The run queue, bound to a processor, can also be changed depending on system work load. In the original implementation (R32V3.0) there was only one run queue for all processors assigned, although more run queues could have been possible. See [7] for more details.

To support applications more efficiently the processor binding and the number of run queues was altered. In our implementation there exists one *system run queue* for all system processes, which is bound to one of four processors. For each application running on a node an *application run queue*, local to that node, will be created. These application run queues are handled by the remaining processors. The binding is not static and can be changed dynamically.

Additionally a new concept called *gang scheduling* has been realized. Each application is assigned an application priority. The tasks of the application with the highest priority will be scheduled first. The application priority can change dynamically, depending on the system work load. The system workload is supervised by a distributed scheduler process, which will change application priorities and processor binding accordingly.

Interrupt Mechanism. As shown in section 3.2 the access time to the communication memory is higher than, for example, to the local memory. To use polling mechanisms on the communication memory is therefore very inefficient and must be avoided, at least in the kernel. Because of this an interrupt connection between

www.manaraa.com

nodes is necessary. This connection is made only between those immediate nodes which share a communication memory. We use a special hardware, supported by software, to generate these inter-node interrupts.

With every interrupt triggered a word is provided at a defined memory location in the communication memory owned[4] by the triggering node. The interrupt hardware recognizes the port on which the interrupt occurred and the software can locate the corresponding node number and communication memory, from which the supplied word can be read.

The interrupt word consists of two parts. The first part is 8 bit wide and represents the *interrupt type*, the second part, the *data-part*, is 24 bit wide and is free for other use. The interpretation of the type depends on the data-part.

An interface is provided by the interrupt module, so that it can be used by other kernel modules[5]. A module has to reserve interrupt types as needed and register a callback function for every reserved type. Some types are already reserved by the interrupt module itself. They are used e. g. for establishing an initial connection or for state information important for other nodes. For a kernel module the reserved types must be system-wide identical. A single *interrupt-send* routine is provided to initiate an interrupt. Parameters of this function are the destination node number, the interrupt type, the data-part and a time-out value. With this time-out value one can switch between time-out mode, blocking and non-blocking mode.

In case of an interrupt the interrupt mechanism reads the supplied interrupt word, extracts the type and then calls the registered callback function with the senders node number, the interrupt type and the data-part as parameters.

Certain enhancements have been implemented to increase the robustness of the interrupt mechanism. The interrupt mechanism automatically tries to establish connections to all immediate neighbours which are accessible. It monitors these connections and reports changes in status to the kernel modules by using the callback functions. By using a FIFO queue, the interrupt mechanism is able to smooth short peaks in the interrupt frequency.

Message-Passing Mechanism. The message-passing mechanism was implemented as one of those kernel modules using the interrupt mechanism.

A message consists of the message header and the message body, which can hold six words (24 bytes). For the messages a static buffer pool is allocated in the communication memory modules which the node owns. A special buffer management was implemented. It has the responsibility of keeping track of each buffer sent. This is very important for maintaining consistency of the buffer pool.

The interface of the message-passing module is constructed in the same way as the interface of the interrupt module. One has to reserve message types and register a callback function for each type reserved. To actually send a message a single *message-send* function is provided.

---

4. To provide a uniform structure of the communication memory, the physical memory may be divided into logical parts, which may be assigned to different nodes.
5. We call each of our implemented kernel extensions a module.

www.manaraa.com

If the *message-send* routine is called, the message-passing mechanism allocates a buffer for the message, fills in the message header and copies the message body. Some processor nodes may have more than one logical communication memory, so the buffer is allocated in that memory module which the receiver or the routing node has access to. The message-passing mechanism then calls the *interrupt-send* function with parameters destination node, type and index of the allocated message buffer.

Message buffers sent to an immediate neighbour are not sent back immediately, but are gathered by the receiver. This is done to reduce the interrupt rate. There are three events in which accumulated buffers are sent back:

- A certain amount is exceeded. The limit is a tunable parameter.
- A message is sent in the opposite direction. The accumulated buffers belonging to the receiver are simply added to the message.
- A neighbour requests the return of the used buffers.

A simple protocol guarantees that a message is received by the destination node. Additional protocols assure that a received message is accepted by the destination kernel module.

Shared-Memory Mechanism. Another communication mechanism, beside the message-passing mechanism, is the shared-memory mechanism. In the following section we introduce our implementation of this mechanism.

The shared-memory mechanism consists of two parts. These are the communication memory manager which provides the linkage to the physical shared-memory and, as main part, the shared-memory manager. The shared-memory manager implements the necessary protocols to maintain the consistency of allocated shared-memory segments. It also supplies a simple interface useable by other kernel modules.

Communication Memory Manager. To allocate or free pages[6] from the communication memory, the shared-memory mechanism uses calls to the communication memory manager. The pages available for shared-memory are numbered consecutively and linked by using a table containing one entry for each page. The entries determine the number of the following page, which need not be the one physically following. What distinguishes this memory manager from others is the lack of automatic memory mapping or unmapping. Therefore a call to the *allocate* function does not return the start address of the allocated memory, but a pointer to a table containing the page numbers. The information about the pages is essential, because the address space mapping may not be the same on all nodes. All tables are situated in the communication memory itself so that they are accessible by immediate-neighbour nodes. Additional calls exist for calculating addresses out of page numbers and for mapping and unmapping the allocated memory into and out of the kernel address space.

6. The hardware only supports pages of 4K byte granularity.

المنارة للاستشارات
www.manaraa.com

Shared-Memory Manager. The shared-memory manager provides the functionality used for communicating with immediate neighbours and keeps track of allocated pages to allow their re-integration in case of faults on neighbouring nodes. On allocation of a shared-memory segment, the memory manager validates provided parameters and chooses that communication memory which the destination node has access to. If an immediate neighbour wants to share an allocated memory segment, the memory manager provides upon request the offset to the corresponding page table. For the inter-node communication the message-passing mechanism is used. On the destination node the shared-memory manager is able to locate the page table and to map the shared segment into the kernel address space.

On top of the shared-memory manager a system-call interface (as described in [6]) is established. This interface allows an efficient use of the shared-memory mechanism by the application programmer.

In MEMSOS we want to examine certain aspects of different communication mechanisms. In section "Message-Passing Mechanism" the basic message-passing mechanism was introduced. This mechanism was built on top of the interrupt mechanism described above. In this section we have described the shared-memory mechanism. In this shared-memory mechanism we use the message-passing mechanism as the basis for communication. This was done because the amount of communication needed and the time used for it is fairly small in comparison with the data transferred and the time needed for reading and writing the data.

6 Conclusions

In this paper the MEMSY project was introduced. MEMSY belongs to the class of the shared-memory multiprocessors. Viewed from the hardware level massive parallel systems with global shared-memory have not been realizable up to now. A compromise has to be made between an efficient system on one side and a general purpose system on the other side.

MEMSY offers tightly coupled processor nodes arranged in a hierarchy of grids. The sharing of memory is only between nearest-neighbour nodes. It was shown that with the additional hardware, called coupling units, it is possible to reduce the amount of necessary connections without too much loss in efficiency. Because of the constant complexity of inter-connections and the modular concept the system is easily scalable.

An easy to use programming model which offers even a great variety of paradigms used in the programming models of other high-performance multiprocessor systems was introduced. Because of this programming model many existing user programs are easily portable to our system. Currently some programs are ported to the MEMSY system. By examining these programs we hope to gain more information about the system performance and be able to take valid measurements.

Our application concept was introduced. It offers a way to supervise distributed user programs. It is the basis for further work to be done in the area of user support and load balancing.

www.manaraa.com

Because MEMSY is an experimental multiprocessor system it was tried to implement as many communication mechanisms as possible. An important aspect in doing this was to be able to compare their usefulness and performance and therefore be able to validate our multiprocessor concept.

References

1. M. Dal Cin et al., "Fault Tolerance in Memory Coupled Multiprocessors"; in this volume
2. G. Fritsch et al., "Distributed Shared-Memory Architecture MEMSY for High Performance Parallel Computations"; *Computer Architecture News*, Vol. 17, No. 6, Dec. 1989, pp. 22 - 35
3. W. Händler, F. Hofmann, H.-J. Schneider, "A General Purpose Array with a Broad Spectrum of Applications"; *Computer Architecture*, Informatik Fachberichte, Springer Verlag, No. 4, 1976, pp. 311-335
4. U. Hildebrand, *Konzeption, Bewertung und Realisierung einer dynamischen Netzwerkkomponente für speichergekoppelte Multiprozessoren*, Dissertation, Arbeitsberichte des IMMD, Univ. Erlangen-Nürnberg, Band 25, No. 5, 1992
5. R. Hofmann, "The Distributed Hardware Monitor ZM4 and its Interface to MEMSY"; in this volume
6. Kardel, W. Stukenbrock, T. Thiel, S. Turowski, *Anleitung zur Benutzung des MEMSY-Programmiermodells für Anwender*; interner Bericht, IMMD 4, Univ. Erlangen-Nürnberg, Oktober 1991
7. Karl J. Rusnock, *Multiprocessor SYSTEM V/88 Release 3 Design Specification*; Motorola, Confidential Proprietary, May 1989
8. K. Rusnock, P. Raynoha, "Adapting the Unix operating system to run on a tightly coupled multiprocessor system"; *VMEbus Systems*, Oct. 1990, Vol. 6, No. 5, pp. 8-28
9. K. Rusnock, *The Multiprocessor M88000 RISC Product*; Motorola Microcomputer Division, Tempe, AZ 85282, 1991

www.manaraa.com

Fault Tolerance in Distributed Shared Memory Multiprocessors

M. Dal Cin, A. Grygier, H. Hessenauer, U. Hildebrand,
J. Hönig, W. Hohl, E. Michel, A. Pataricza[1]

Informatik III
Universität Erlangen-Nürnberg

Abstract

Massively parallel systems represent a new challenge for fault tolerance. The designers of such systems cannot expect that no parts of the system will fail. With the significant increase in the complexity and number of components the chance of a single or multiple failure is no longer negligible. It is clear that the redundancy, reconfigurability and diagnosis techniques must be incorporated at the design stage itself and not as a subsequent add-on. In this paper we discuss the fault tolerance techniques developed for MEMSY, a massively parallel architecture. These techniques can, in principle, be easily transferred to other distributed shared memory multiprocessors.

1 Introduction: Fault Tolerance and Parallel Computers

Fault tolerance is the ability of a system to tolerate the presence of a bounded number of faults (and the resulting errors) and to continue in operation until scheduled maintenance can take place. It is achieved either by fault masking or by fault handling in such a way that the application service is not threatened and the faults do not become visible to the user. Fault tolerance is a critical design parameter especially for massively parallel computers. There are at least three reasons:

First, if the system contains several thousands of processing nodes, the probability of node failures cannot be neglected, regardless of which technology is used.

Second, applications that run on massively parallel computers require long execution times. For example, to attack the "Grand Challenges" [5] weeks or even months are needed. Therefore, a massively parallel computer must provide long uninterrupted computation times and reliable access to a large amount of data.

Finally, massively parallel computers should be scalable. A scalable computer is one whose power in every performance dimension grows transparently as processing nodes are added. Adding more and more components beyond a certain number, however, can dramatically decrease reliability and hence, system performance. To overcome this scalability limit, fault tolerance measures are necessary.

Fault tolerance requires redundancy in hardware, software or time. It is obvious that for massively parallel computers hardware redundancy has to be employed as sparingly as possible. Fault handling is, therefore, more adequate and more cost-effective than providing fault-masking hardware redundancy. This holds true particularly for very small failure rates. Fault handling includes error detection, error location, and fault con-

1. Guest researcher from TU Budapest, Dept. Measurement and Instrumentation Engineering

www.manaraa.com

finement. It also includes damage assessment as well as error recovery and fault treatment in conjunction with continued service to the user. These techniques must be implemented in a manner that does not severely affect performance and scalability. Yet, high error coverage and low error latency have to be achieved.

These requirements pose strong demands on the implemented fault detection mechanisms. They imply that each processing node of a massively parallel computer has the capability to check itself concurrently with program execution. Testing and test management are too time consuming. Moreover, the majority of faults in a computer is transient [18] and cannot be detected with high enough probability by (off-line) testing. Although software techniques can be used for error detection, they are, however, likely to have high error latency. Therefore, hardware solutions are preferred.

It would be ideal if fault tolerance for massively parallel computers could be based on fail-silent processing nodes. A fail-silent component must be self-checking and fail-safe. It either operates correctly, or does not produce any output. Also it allows to determine whether or not it is fault-free without having to rely on time-outs. Such a component is said to suffer crash failure semantics [3]. To construct a fault-tolerant system out of fail-silent components is relatively easy.

It is obvious that an ideal fail-silent processing node cannot be built economically. The implementation must, therefore, be based on a realistic fault model and an analysis of its error coverage. Near fail-silence can be achieved, for example, by self-checking logic, by a master-checker node configuration or by watchdog coprocessors.

As soon as an error is detected, it must be handled. If roll-back is part of the error handling mechanism, each processing node must periodically checkpoint its state in a memory with stable storage property [11]. The stable storage property is necessary to prevent the faulty node from corrupting its state information, and to protect this information against latent memory errors. (Such errors can be detected by the stable storage by memory scrubbing.) With checkpointing, a processing node is enabled to resume a failed computation by reading the relevant state information in the case of a temporary fault. Checkpointing also enables a functioning node to resume computation of a failed node in case of a permanent fault (reconfiguration). Yet, in massively parallel systems there is no global memory. Hence, checkpoints are to be stored distributed in such a way that a consistent global checkpoint can be maintained, even if some nodes or communication links become faulty.

In a massively parallel system the communication subsystem can also fail. Its fault tolerance requires redundant communication links and a robust communication protocol which should always allow correctly functioning processing nodes to communicate correctly in bounded time. To ensure uncorrupted data exchange, standard techniques employing checksums and retransmissions can be used.

In summary, fault tolerance for massively parallel systems should ideally be based on:

- Fail-silent processing nodes employing concurrent error detection with high error coverage and low error latency
- Fault-tolerant interconnection networks with provisions for detecting and tol-

www.manaraa.com

erating link and switch errors
- Stable checkpoint memories allowing memory scrubbing to detect latent errors
- Efficient error handling mechanisms (rollback and reconfiguration)

In the following an approach to implement these features into the experimental fault-tolerant version of the multiprocessor system MEMSY [7] is described.

2 Hardware Error Detection

The primary goal in designing error detection mechanisms is high fault coverage, while keeping redundancy on a moderate level. This can be achieved by combining built-in standard mechanisms on chip level with efficient hardware fault detection mechanisms on system level.

For economic reasons, the large hardware overhead resulting from multiple modular redundancy for the whole system is not feasible. Accordingly, modular redundancy has to be restricted to the most crucial hardware resources, such as the CPU. Duplication within the computing nodes can be based on master-checker mode or on the use of a watchdog processor to monitor concurrently the program execution of the main processor.

2.1 Master-Checker Mode

The master-checker (MC) mode is based on the duplication of the processors. Both processors run the same program clock-synchronously and process the same data stream. In the Motorola M88k microprocessor family, MC-mode can be set up practically without external components, as the necessary logic is integrated onto the chip itself. An internal comparator is assigned to each bus pin, which compares internal and external values at the output pins. Only one processor (the master) outputs data. The output bus drivers of the other processor (the checker) are disabled. Accordingly, during data transfer the comparators on the checker pins compare the internal signals and those driven by the master. In the case of mismatch an error is signalled.

The M88k family allows setting up of the MC-mode either at the chip level (i.e. by duplicating a CPU chip and setting up the MC-mode separately on the instruction and data buses), or at the processor level (i.e. by setting up the MC-mode on the system bus).

The latter mode is implemented in an experimental version of the MEMSY node by use of the Motorola MVME 188 board system comprising two identical MC88100 processors. Both processors can either be used independently or jointly with reduced performance as MC-system. In MC-mode the corresponding pins of both integrated circuits (CPU and/or MMU chips) are interconnected by means of an add-on board.

Error coverage and latency

Duplication ensures full coverage of transient and permanent faults in a single unit of a MC-pair, but it does not detect errors affecting both units, e.g an error in the instruction or data stream. Moreover, it does not detect errors in the memory or peripherals.

www.manaraa.com

Therefore, the memory modules of MEMSY and the bus have parity protection. Hence, error detection of the computing core covers most single faults.

In addition to fault coverage, short error latency is a primary objective as well. A long error latency may result not only in a significant loss of computing time, but can also cause serious difficulties in fault diagnosis, since long fault propagation chains weaken the correlation between a fault and the resulting operational errors.

In MC-mode, only the data transfer on the bus can be observed and, accordingly, data in the CPU and in the cache remain unchecked until they are written back to the main memory. The Motorola MC88200 MMU chip supports three different memory update policies:

- *Cache inhibit*: Only direct data transfer with the main memory.
- *Write through*: New values of data are immediately written back into both the main memory and cache. Read operations access the data in the cache
- *Copy back*: Both read and write operations access the data in the cache. Modified data is written back only at the end of the computation or at the occurrence of a cache miss.

For the most frequently used (and probably most important) data copyback to the main memory can be delayed until the very end of the computation. Therefore, with copy back the error latency can, in the worst case, approach the full computing time! In Table 1 the fault coverage of the MC-based checks of different bus transactions are summarized for different memory update policies.

<table>
<tr><th rowspan="2">Cache</th><th rowspan="2">Cache policy</th><th colspan="2">Checks of read operations</th><th colspan="2">Checks of write operations</th></tr>
<tr><th>Address</th><th>Data</th><th>Address</th><th>Data</th></tr>
<tr><td rowspan="3">Data</td><td>Inhibit</td><td>Yes</td><td rowspan="5">No</td><td rowspan="2">Yes</td><td rowspan="2">Yes</td></tr>
<tr><td>Write through</td><td rowspan="2">No</td></tr>
<tr><td>Copy back</td><td>No</td><td>No</td></tr>
<tr><td rowspan="2">Code</td><td>Inhibit</td><td>Yes</td><td rowspan="2" colspan="2">Not applicable</td></tr>
<tr><td>Enabled</td><td>No</td></tr>
</table>

Table 1 Fault coverage of MC-based checks

For the instruction fetch only read operations are performed. Hence, there are only two alternative cache policies: cache enabled and cache inhibited. Only if the cache is inhibited, the MC-setup performs checks on the instruction flow with low latency. However, inhibiting the cache may reduce performance drastically.

www.manaraa.com

The most important limitation of the error detecting capabilities of the MC-mode results from the only partial duplication, i.e. the system remains still unprotected against faults in common resources located in the "outside world" of the processing core. Hence, in addition to the MC-mode of the CPU an error detection mechanism is required, which checks the instruction flow and guarantees at least a moderate fault coverage for the whole system.

2.2 Watchdog Processors for Control Flow Checking

A watchdog processor (WDP) is a relatively simple coprocessor which concurrently monitors the correctness of program execution. A WDP compares reference signatures with run-time signatures. They encode the specified and the real program flow, respectively. At program start the reference signatures are transferred to the WDP. The run-time signatures are generated by the WDP while the program is running on the main processor (MP).

Control flow checking can be divided into *derived* and *assigned signature* based methods, depending on the run-time signature generation method used [14]. First consider derived signature based control flow checking. The WDP compresses the instruction code stream executed by the MP into run-time signatures. Then it compares these signatures with the reference signatures. Unfortunately, this method requires full observability of the instruction stream. In the case of on-chip caches or on-chip instruction prefetch queues (IPQ) similar problems arise as discussed in Sec. 2.1. To solve these problems the WDP has to be embedded in the processor core [12]. In case of a IPQ, the WPD has to emulate the processor pipeline.

In an assigned signature based method, the MP explicitly sends compilation time generated signatures to the WDP uniquely identifying its current state in the program execution. This method is, therefore, independent of the processor architecture.

Extended signature integrity check

The assigned signature based method developed for MEMSY is called extended signature integrity check (ESIC). It is an extension of the signature integrity checking (SIC) method described in [13]. As in the original SIC method a preprocessor extracts the so-called control flow graph from the high level programming language source code [8], [15]. In this graph nodes represent branch-free program blocks (instruction sequences) and arcs correspond to control transfers (e.g. branch-type instructions as if-then-else or case statements). To each program node an unambiguous label (signature) is assigned. The graph is translated into a WDP program, reflecting the control structure of the main program. At all branches of the main program signature transfer statements are inserted into the source code in order to transfer node labels to the WDP during main program execution. These signatures identify the current main program location. Similarly, in the WDP program receive_signature statements are inserted at the corresponding positions. Both programs are compiled into machine codes of the target and watchdog processor, respectively.

The MP and the WDP are started simultaneously and then they execute the main and

www.manaraa.com

the watchdog programs, respectively. The WDP monitors concurrently the correct execution order of the main program blocks by checking the received actual signature sequence. A sequence of signatures is accepted as correct, if it corresponds to an existing path in the program graph. In case of a conditional branch instruction, it is only checked, whether the target instruction belongs to the set of admissible target instructions.

In the original SIC method the WDP program is generated by eliminating all non-control statements from the main program; e.g. if the main program contains a subroutine call, then a similar call will be executed by the WDP, but no WDP code corresponds to arithmetic statements.

We implemented the WDP as a finite deterministic push-down automaton. For this automaton a preprocessor extracts the program graph in a tabular form which defines for each current signature (state) the set of the allowed subsequent signatures.

In order to overcome problems related to subroutines, the preprocessor generates a separate control flow graph for each subroutine and, accordingly, a separate automaton table. The start and the end node of every subroutine are labelled by special signatures, viz. "start of subroutine" (SOP) and "end of subroutine" (EOP), respectively. This signatures contain a field uniquely identifying the subroutine.

If a SOP-signature belonging to an existing subroutine entry point is received, the current state is pushed onto the WDP private stack and the new state of the WDP is the initial state (start node) of the subroutine, i.e. the WDP switches over to the automaton table of the subroutine. If the WDP receives an EOP-signature it checks, whether the signature belongs to the current subroutine, and if so its next state is popped from the private stack, i.e. the WDP resumes checking the calling program.

Contrary to the SIC method the WDP works as an interpreter of signature streams. Therefore, it is necessary to detect the absence of run-time signatures e.g. when the MP stopped due to a fatal system crash. The absence of a signature is detected by time-out. If subroutines are called which do not send signatures (e.g. library calls of the programming environment), the MP can switch off this time-out mechanism by sending a special signature to the WDP (and can switch it on again the same way).

The most important classes of errors which remain undetected by the WDP are:

- Errors which are not manifested as control flow distortions, e.g. errors changing the value of variables
- Branch selection errors in the case of control transfer instructions, if the faulty successor node is syntactically allowed by the program graph
- Short time transients, which occur between two consecutive signatures, but leave the signature stream intact
- Wrong subroutine calls

Errors of the first three types are resistant to any high level control flow check based error detection mechanism. The fourth group results from the new subroutine handling method in ESIC, as a SOP-signature is an allowed successor for each state of the program graph. However, by putting restrictions on the set of the allowed successors this limitation can be reduced considerably.

www.manaraa.com

WDP hardware and measurement results

To demonstrate the usefulness of ESIC for MEMSY a WDP prototype was built based on a T800-25 MHz transputer. The WDP is connected to the system bus by a FIFO memory. It is used for downloading WDP programs and for the signature transfer. The main advantage of the FIFO is, that the signature transfer to the WDP is reduced to a single, constant addressed memory access. To avoid signature stream overflow, the MP can be stopped temporarily by an interrupt generated by a FIFO-full condition. The links of the transputer may form a communication network for the exchange of diagnostic information between the WDPs of different MEMSY processing nodes.

We measured the time overhead for sending and interpreting a signature by simulation on the multiprocessor system Sequent Symmetry with 16 processors and directly on a memory coupled multi-transputer system [15]. The computation times of iterative multigrid solvers of the 2-D Poisson equation and of the 2-D Navier-Stokes equation on the multi-transputer system are shown in Table 3. Also shown is the number of signatures generated and the number of conflicts. A conflict occurs if the buffer is full and the application has to wait until the WDP processor has read a signature out of the FIFO memory.

The time required for sending one signature varies from 1.7 to 4.0 microseconds. This results from the large number of conflicts particularly in the case of the Poisson equation. The time needed by the WDP to check one signature is 10 microseconds. In the case of the Navier-Stokes equation the average time between two send_signature statements is longer than the time to check a signature. Hence, the time overhead for the control flow check is very small (4%).

	Signatures	T_{cNC} [sec]	T_{cC} [sec]	T_s [μsec]	Conflicts	Overhead
Poisson	5 504 000	54.5	76.8	4.0	235 564	40%
Navier-St.	3 092 000	138.4	143.7	1.7	28	4%

T_{cNC} = computation time without check
T_{cC} = computation time with check
T_s = average time for sending a signature

Table 2 Measurement results

2.3 Simultaneous use of master-checker mode and watchdog processors

As described before, the MC-mode assures high fault coverage for the computing core, but leaves the system unprotected against faults in other parts. A WDP provides a moderate protection against faults in the overall system resulting in control flow errors, but can not detect pure data errors and data related control flow errors such as the selection of a wrong branch in an if-then-else statement.

If both methods are applied simultaneously, the signatures are sent by the main processors to the WDP and are checked by the checker. Therefore, even data related control

www.manaraa.com

flow errors can be detected, if they originate from faults in the computing core. The main results of the simultaneous use of the MC-mode and the watchdog processor are summarized in Table 3. Shaded parts denote the method dominating the fault coverage.

<table>
<tr><th rowspan="2" colspan="2">Fault location</th><th colspan="3">Fault coverage in %</th></tr>
<tr><th>Master-checker only</th><th>Watchdog only</th><th>Simultaneous use</th></tr>
<tr><td rowspan="2">CPU-internal</td><td>control</td><td rowspan="5">~ 100</td><td>~ 80</td><td rowspan="5">~ 100</td></tr>
<tr><td>data</td><td>0</td></tr>
<tr><td colspan="2">Code cache</td><td rowspan="2">50-80</td></tr>
<tr><td rowspan="2">Data cache</td><td>control</td></tr>
<tr><td>data</td><td>0</td></tr>
<tr><td rowspan="2">External components</td><td>control</td><td rowspan="2">0</td><td>50-80</td><td>50-80</td></tr>
<tr><td>data</td><td>0</td><td>0</td></tr>
</table>

Table 3 Fault coverage

As it is shown in this table, the only important uncovered errors are external data errors. However, such errors can easily be detected if the main memory is protected by a proper error-detecting code.

3 Fault Tolerant Interconnection Network and Stable Storage

3.1 Fault Tolerance Aspects of the Network Unit

The network (or coupling) unit used as a building block of the MEMSY interconnection network (cf. [6], [7]) provides mechanisms at hardware level which support the efficient use of alternative communication paths in case of faults. The basis for this fault tolerance feature is the ring structure of the internal subpaths. Alternative communication paths consist of disjoint sets of internal subpaths. Hence, the permanent failure of one internal subpath within a network unit can be tolerated. The result will be a reduced bandwidth but full interconnection is guaranteed (graceful degradation).

The failure of a single internal subpath is always tolerated. A simple example demonstrates the reaction to a faulty memory access. Consider the following scenario (cf. Fig. 1):
Processing node P_i is faultfree and performs an access to the communication memory CM_j. P_i observes the memory access as faulty. (For error detection, parity checking is used.) To mask a temporary fault the memory access is repeated. If the error still occurs the fault may reside either within the communication memory or within the interconnection network.

www.manaraa.com

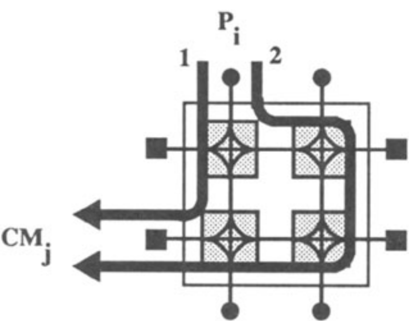


Fig. 1 Alternative communication paths from P_i to CM_j within the network unit

At this point it is not possible to decide which one has lead to the faulty memory access. To locate the fault the communication memory is accessed via different communication paths and the results are compared, Fig. 1.

At hardware level this possibility to localize faults is supported as follows. Each communication memory which is directly accessible by a processor is mapped into the processor's address space (Fig. 2) such that accesses to communication memory CM_j can be performed within the address subspaces AS_{jd}, AS_{j1} or AS_{j2}. The same offset address within any of these address subspaces will select the same memory cell but a different communication path within the network unit:

- AS_{jd}: dynamic selection of the communication path according to the current switch settings
- AS_{j1}: communication path 1
- AS_{j2}: communication path 2.

As long as no fault has been detected, dynamic selection of the communication path is used. This provides higher efficiency by avoiding conflicts within the network unit.

address subspace	
	CM_{j+1} (dynamic)
AS_{j2}	CM_j (fixed 2)
AS_{j1}	CM_j (fixed 1)
AS_{jd}	CM_j (dynamic)
	CM_{j-1} (fixed 2)

Fig. 2 Multiple mapping of communication memories

www.manaraa.com

In case of a fault, all accesses to communication memories must be carried out either within AS_{j1} or AS_{j2}. Which of these two address subspaces is selected is determined individually for every processing node.

Let us now consider stuck-at faults. A stuck-at fault of one switching element within the network unit may result in one of three situations depending on the static switch setting (Fig. 3):

- The switch is inactive: The p-port and the m-port are isolated (Fig. 3a)
- The switch connects the p-port with the m-port (Fig. 3b)
- The switch connects the p-port and the m-port with different internal subpaths (Fig. 3c, Fig. 3d)

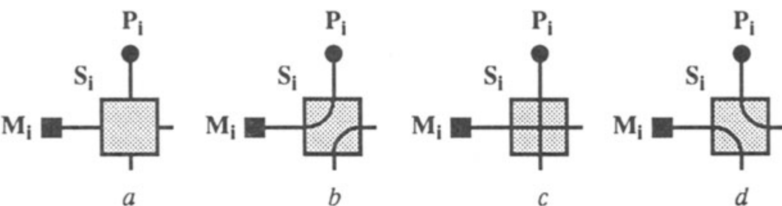


Fig. 3 Stuck-at faults of a single switching element of the network unit

Thus, a switch is either intact, inactive or stuck-at. The inactive state is equivalent to link failures. If switch S_i is inactive then processor P_i can still access its communication memory M_i by using its second port and a different network unit (cf. [7]). Many single stuck-at faults can be tolerated within the network unit itself, Fig. 4 demonstrates this. Fig. 4a is the desired access pattern; Fig. 4b and Fig. 4c show how the stuck-at faults of switch S_0 are masked by the network unit. An analysis of the fault tolerance capacity of the interconnection network of MEMSY is given in [6].

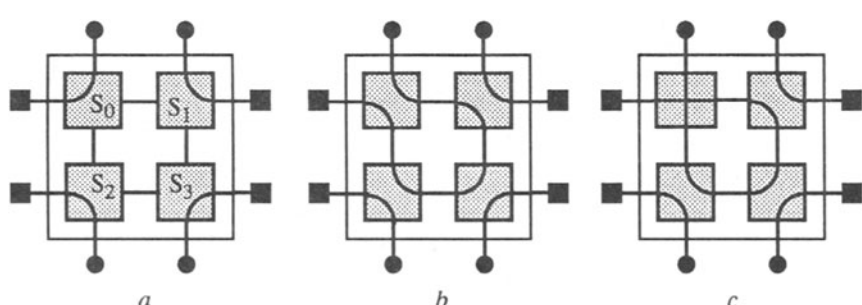


Fig. 4 Masked stuck-at faults

المنارة للاستشارات
www.manaraa.com

3.2 Stable Storage for System Data and Checkpoints

For storing checkpoints a very reliable memory is necessary: viz. a 'stable storage' [1], [11]. It must have some special *properties*:

- Persistency:
  Information stored in the stable storage must not be changed by faults. If a fault occurs it must be corrected within the stable storage.
- Autonomy:
  Processors that are connected to the stable storage have no direct access to stored data. They act as clients. Only the control of the stable storage can manipulate the stored data. Thus, it is guaranteed that a faulty processor cannot destroy information in the stable storage.
- Information hiding:
  The data units in the stable storage are "stable objects". The clients only know name (a reference) and size of the objects.
- Atomic update for stable objects:
  Any update operation must be done for complete stable objects. If it fails, the stable object must be set back to its previous state.
- Fault tolerant control for the stable storage:
  The control of the stable storage must be able to recognize its own defects at once and to react in the correct manner, such that no data can be destroyed in the stable storage. Furthermore all stable objects must still be protected and accessible (readable) in spite of such a defect.

Establishing a checkpoint implies a certain expense: memory space and time. (The required memory space depends on the user's program. In general some megabytes per processing node are needed. The time for establishing a checkpoint is in the range of seconds per processing node.) With global checkpointing all processors will establish their checkpoints at about the same time. Hence, for a large multiprocessor system only distributed checkpointing is feasible, so that all processors can establish their checkpoints concurrently. Therefore, it would be best to attach a stable storage to each processing node.

If a processing node becomes faulty its data in the stable storage must be accessible to other processing nodes. MEMSY offers a nearest neighborhood connection between processing nodes. So the stable storage can be placed where the communication memory is.

However, the properties required for the stable storage lead to an expense in hardware that is much higher than that of a communication memory. In MEMSY not all processing nodes will be equipped with a stable storage. Each elementary pyramid (4+1 processing nodes) will contain one or two stable storage units that are installed in parallel to a communication memory module (Fig. 5). Within an elementary pyramid there is a full connection between the nodes. Therefore, all processing nodes have direct access to one stable storage at least.

The *stable storage unit* consists of a control unit and two large memory modules for stable objects. From outside the stable storage appears like a coprocessor which can be

المنارة للاستشارات

www.manaraa.com

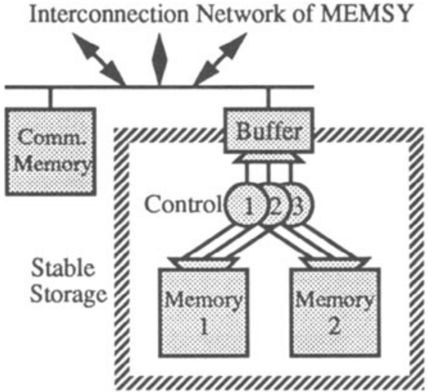


Fig. 5 Structure of the stable storage unit

addressed via a memory interface. The processing nodes communicate with the stable storage via command and parameter registers. The status of their command is returned in a result register.

The commands are:

- install or delete a stable object
- execute a read or a write operation on a stable object
- check the stable objects and correct faults if necessary
- return information about the state of the stable storage.

There is a buffer for exchanging data between the processing nodes and the stable storage. Processors put data blocks into it when they command a write operation. With a read operation the stable storage puts a copy of the stable object into the buffer and informs the processor via the result register.

Protection of stable objects against illegal access is very important. Therefore, the processors must run a protection protocol. Only if it passed correctly, the stable storage executes the command. Otherwise the command will be rejected. In this way the stable storage checks the correct operation of a processor. As part of the protocol, the processor must attach its password to each command. The processor must ask for the password before it uses the stable storage the first time.

For error detection in the data transfer between a processing node and the stable storage parity bits (4 bits for a 32-bit word) are used. Furthermore, for each data block a checksum is generated, transferred and checked. The stable storage will execute the command only if no error is detected.

A critical situation arises when the stable storage executes an update of a stable object. Therefore, all stable objects are stored twice (in two different memory modules). This makes it possible to correct errors during updates of an object by copying a consistent state from the other not affected module.

www.manaraa.com

The control of the stable storage has to detect its own faults immediately and to operate in spite of faults. Therefore, the control unit will be implemented as TMR unit (Triple Modular Redundancy). Thus an error of a single module is detected at once and masked. Connected processing nodes are informed, so that they can react appropriately. For example, they can transfer their stable objects to another stable storage.

The stable storage needs information (control blocks) for administration purposes. Within the stable storage all stored information (checkpoints and administration data) is protected by ECC (Error Correcting Code). For protection the control blocks are treated as stable objects, too.

4 Fault Treatment by Rollback Recovery

The rollback-recovery scheme for memory-coupled multiprocessors as proposed for MEMSY is based on the notion of distributed snapshots [2], [10]. The state of an application program is defined by the state of every participating process plus the state of shared memory segments. Other approaches, such as conversations [16], [17] or message logging schemes [9] are not considered suitable, the latter because logging is impossible, as shared-memory communication does not necessarily involve a message passing programming model.

Furthermore, we consider numerical applications as the primary use for memory-coupled multiprocessors [7]. These applications are characterized by high communication loads. Most numerical applications are basically iteration loops, offering suitable spots for global checkpointing. Thus, to optimize checkpointing, we assume the programmer specifies the type of data to be stored and the exact places within the execution of a program, when checkpoints shall be taken. A fully transparent scheme would cost much more coordination overhead.

Our primary goal of an implementation of the rollback-recovery mechanism is achieving as high an efficiency as possible, that is, we want the mean runtime of a job on a fault-ridden, fault-tolerant system (T_{FF}) to be about as long as the runtime of the same job on a fault-free system with no fault tolerance mechanisms (T_0). The efficiency can be defined as:

$$E = \frac{T_0}{T_{FF}}$$

Our second goal is a high trustworthiness of the results of a long-running computation. The means to reach this goal are provided by the error detection mechanisms described earlier in this paper. We expect 99% of all faults to be detected.

To reach the goal of high efficiency two mechanisms are employed:

- two-level checkpointing scheme with 'hard' and 'soft' checkpoints
- hardware-assisted, non-blocking coordination of global checkpointing

The two-level checkpointing scheme is based on the assumption, that there are different media for storing checkpoints available: fast, unreliable storage and comparatively slow, stable storage. Fast storage can be the local memory (main and background

www.manaraa.com

storage) of a processing node, while stable storage can be replicated filesystems on a remote server or stable RAM (see section 3.2). Error recovery from the local, unreliable checkpoint storage, would not always succeed, but may fail with a certain probability, whereas recovery from stable storage is supposed to succeed always.

Another aspect of numerical applications is the large size of data structures. We therefore assume, that checkpoints occupy a considerable amount of space in the storage medium, imposing an upper limit on the number of checkpoints kept in storage. Thus, global coordination is necessary, to determine at which times checkpoints are obsoleted and may be discarded.

Global checkpointing requires all processes of an application to record their state atomically, even if errors during checkpointing occur. A decentralized two-phase commit protocol is employed to ensure a new checkpoint has been taken before outdated checkpoints are discarded. As decentralized two-phase commit protocols require efficient broadcast algorithms, special support hardware reducing the number of broadcasts is presently under development.

4.1 Fault Classes

We divide faults into two classes: 'soft' faults and 'hard' faults. Soft faults allow global recovery from state information kept locally, while hard faults require rollback to checkpoints from stable storage. Hard faults would include hardware failures and node crashes, as node crashes cause the loss of local memory data. The two-level checkpointing scheme is based on the assumption that most errors fall into the soft category. This assumption is based on several observations.

First, once detected, hardware errors cause exceptions, caught by the operating system of a node. The operating system then determines the cause of the exception. If the exception occurred in user mode, the application program is signalled, otherwise, in most cases, the node will crash. Yet it is safe to assume that the amount of resources, such as cpu time, spent on the execution of the operating system in kernel mode is far less than the amount spent on the execution of the application. Most transient hardware errors, therefore, will fall into the soft category.

Permanent faults, such as node failures, fall into the category of hard errors. Reconfiguration and subsequent recovery is not possible from the local checkpoints, as the state information in local storage of the defective node cannot be accessed from other nodes. Studies in [18] show the frequency of permanent hardware errors about ten times less than the frequency of transient errors. Even though operating system software errors and their manifestation are an unknown quantity, but it is safe to expect that at least 75% of all errors are soft errors.

4.2 Two-level Rollback Scheme

There are several advantages to a two-level rollback algorithm. First, soft checkpoints are created comparatively quickly, because of the use of fast storage media. Besides soft checkpoints can be written in a 'careless' fashion, as there is no need to guarantee sure recovery. Fast checkpointing reduces the length of the optimal check-

www.manaraa.com

pointing interval, thus reducing the loss of job progress in case of errors. Shorter intervals also reduce the probability of the decay of checkpoint data, as checkpoints are kept for a shorter time. As recovery with a soft checkpoint may fail, efficiency is reduced resulting in increased job runtime.

As a remedy, hard checkpoints are created. In case of successful 'soft' rollback the soft checkpoint has been proven to be intact, thus, whenever rollback to a soft checkpoint succeeds, this checkpoint is made a hard checkpoint, by moving it to stable storage. As 'hard' rollback is considered to be always successful, even for very long job runtimes the efficiency drops asymptotically to a comparatively high value. On the other hand, hard checkpoints are saved seldom, one hard checkpoint per mean-time-to-failure (MTTF) on average. Thus, the impact of slow access to stable storage on overall runtime is kept low.

Fig. 6 shows the efficiency of different checkpointing schemes, versus the runtime of a job, expressed in multiples of a system's MTTF. The efficiency is defined as above. Value '1' denotes the efficiency of a fault-free no fault-tolerance system. The first curve ('no fault-tolerance') should better be labelled as 'almost no fault tolerance', as some means of fault detection are employed: The job is rerun, until the same result has been computed at least twice. The second example, denoted as 'restart' examines the case in which errors are detected (at no time cost) and where, in case of an error, the job is restarted from the beginning. The third example ('soft checkpointing only') shows the efficiency of a checkpointing algorithm, where soft checkpoints are taken and rollback to a soft checkpoint succeeds with a probability P_s=7/8. Finally, the last example ('hard & soft checkpoints') shows the efficiency of the proposed checkpointing scheme.

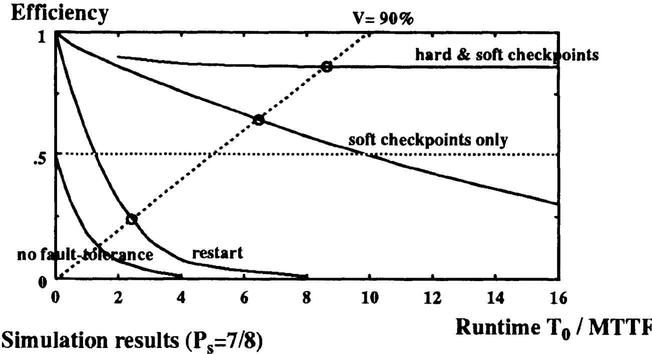


Fig. 6 Efficiency of several checkpointing schemes

We define the trustworthiness of a computation result as the probability that no error during the job runtime has gone undetected. The dotted line shows the boundary, after which this probability falls below the 90% mark, assuming a coverage of 99%. The trustworthiness of a job with a runtime equal to 8 times the MTTF is higher than 90% if hard and soft checkpointing is employed, and less than 90% if only soft checkpointing

www.manaraa.com

is used. Interesting consequences are that any long-running computation requires a coverage of error detection mechanisms significantly higher than 99%, and that efficient recovery mechanisms allow longer job runtimes. In our example the line of 90% trustworthiness is intersected at a higher value of job runtime for mechanisms with higher efficiency.

4.3 Parallelized Checkpoint Coordination

As progress of the application program does not depend upon the result of checkpointing, the two-phase commit protocol can run in parallel with the application, avoiding busy wait for coordination messages. Furthermore, global synchronisation at the end of each checkpointing interval is avoided. Otherwise, every process would be forced to wait for the slowest participant, thus penalizing overall job runtime, if the participants have different runtimes in each checkpoint interval.

With parallel checkpoint coordination the progress of processes is allowed to drift apart, limited by the amount of storage assigned for checkpoints. In the minimal configuration at least two checkpoint buffers have to be provided. One buffer is needed for the 'active' checkpoint. In case of faults the application rolls back to this checkpoint. The coordination protocol guarantees, that the active checkpoint has been established globally. A second buffer is needed for the 'tentative' checkpoint, a checkpoint which exists locally, but about which it is not known, whether every other process reaches the same point of progress. The tentative checkpoint might be committed eventually and become the new active checkpoint, or it might be aborted and removed. Even with only these two buffers different processes can be allowed to drift one checkpoint interval apart, as Fig. 7 shows.

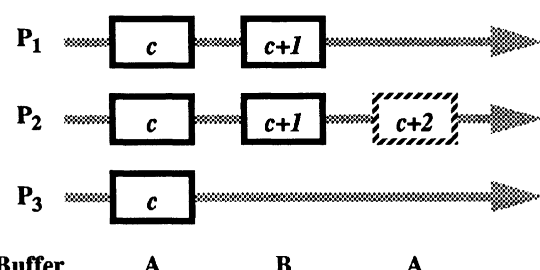


Fig. 7 Usage of two checkpoint buffers

Process P_2 is the farthest ahead. P_2 tries to establish checkpoint number c+2, but is not allowed to, as buffer A is still occupied by the active checkpoint with number c. It has to wait for P_3 to establish checkpoint c+1 as well. After c+1 is committed and thus made the active checkpoint, buffer A is freed and P_2 is allowed to proceed.

www.manaraa.com

4.4 Hardware-assisted Broadcast

Two-phase commit with a decentralized communication structure requires all participants to broadcast their local state. Afterwards, every participant decides independently what to do with the tentative checkpoint, according to the information received locally. In order to make sure that every participant reaches the same decision, the broadcast algorithm has to be reliable and has to guarantee the same ordering of messages. For MEMSY yet another path has been chosen: a special broadcast hardware, based on an optical bus, is provided to allow simultaneous broadcasts: while every participant is sending its local state, it receives the global state at the same time. In a way rather similar to a wired-OR network, any single participant can force the whole application program to abort a tentative checkpoint.

5 Conclusion

The fault-tolerant version of MEMSY is presently under construction. The system is dedicated to the computation of long-running numerical applications. Thus, the primary goals of designing the fault tolerance mechanisms have been to achieve an error coverage as high as possible and a high overall efficiency of these mechanisms.

6 References

[1] Banâtre, M.; Muller, G.; Rochat, B.; Sanchez, P.: Design Decisions for the FTM: A General Purpose Fault Tolerant Machine, Proc. 21th FTCS, pp. 71-78, 1991

[2] Chandy, K. M.; Lamport, L.: Distributed Snapshots: Determining Global States of Distributed Systems, ACM T.o.C.S., vol. 3, no. 1, pp. 63-75, 1985

[3] Cristian, F.: Understanding Fault Tolerant Distributed Systems, Com. ACM vol. 34 (1991), pp. 56-78

[4] Dal Cin, M.: New Trends in Parallel and Reliable Computing: Massive Parallelism and Fault Tolerance. Invited paper, Proc. µP'92, 7th Symposium on Microcomputer and Microprocessor Appl., Budapest, April 1992, pp. 1-10

[5] Grand Challenges: High Performance Computing and Communications. The Fiscal Year 1992 U.S. Research and Development Program. Report by the Committee on Physical, Mathematical, and Engineering Sciences, NSF Washington 1992

[6] Hildebrand, U.: A Fault Tolerant Interconnection Network for Memory-Coupled Multiprocessor Systems, In: Dal Cin, M.; Hohl, W.(eds.): Proc. 5th Int. Conf. Fault Tolerant Computing Systems, Informatik-Fachberichte 283, pp. 360-371, Springer 1991

[7] Hofmann, F. et al.: MEMSY - A Modular Expandable Multiprocessor System, in this volume

[8] Hohl, W.; Michel, E.; Pataricza, A.: Hardware Support for Error Detection in Multiprocessor Systems - A Case Study, Proc. µP'92, 7th Symposium on Microcomputer and Microprocessor Appl., Budapest, April 1992, pp. 81-90

[9] Kai Li; Naughton, J. F.; Plank, J. S.: Checkpointing Multicomputer Applications, Proc. 10th Symposium on Reliable Distributed Systems, pp. 2-12, 1991

www.manaraa.com

[10] Koo, R.; Toueg, S.: Checkpointing and Rollback-Recovery for Distributed Systems, IEEE T.o.S.E., pp. 23-31, Jan. 1987

[11] Lampson, B. W.: The Stable System, in Lampson, B. W.; Paul, M.; Siegert H. J. (ed): Distributed Systems: Architecture and Implementation, LNCS 105, pp. 254-256, 1988

[12] Leveugle, R.; Michel, T.; Saucier, G.: Design of Microprocessors with Built-In On-Line Test, Proc. 20th FTCS, pp. 450-456, 1990

[13] Lu, D. J.: Watchdog Processors and Structural Integrity Checking, IEEE T.o.C., Vol. 31. No.7, 681-685, 1982

[14] Mahmood, A; McCluskey, E. J.: Concurrent Error Detection Using Watchdog Processors - A Survey, IEEE, T.o.C., Vol. 37. No. 2, pp. 160-174, 1988

[15] Michel, E.; Hohl, W.: Concurrent Error Detection Using Watchdog Processors in the Multiprocessor System MEMSY, Proc. 5th Int. Conf. Fault-Tolerant Computing Systems, Nürnberg, Informatik Fachberichte 283, pp. 54-64, Springer, September 1991

[16] Russell, D. L.; Tiedeman, M. J.: Multiprocess Recovery Using Conversations, Proc. 9th FTCS, pp. 106-109, 1979

[17] Shrivastava, S.; Mancini, L.; Randell, B.: On The Duality Of Fault Tolerant System Structures. In: J. Nehmer (ed.), Experiences With Distributed Systems, Proc. Int. WS. Kaiserslautern 1987, pp. 10-37, Springer LNCS 309, 1988

[18] Siewiorek, D. P.: Faults And Their Manifestation, Springer LNCS 448, pp. 244-261, 1987

www.manaraa.com

Optimal Multiprogramming Control for Parallel Computations

Eike Jessen, Wolfgang Ertel, Christian B. Suttner

Institut für Informatik, TU München
Arcisstr. 21, 8000 München 2
email: jessen@informatik.tu-muenchen.de

Abstract. Traditionally, jobs on parallel computers are run one at a time, and control of parallelism so far was mainly guided by the desire to determine the optimal number of processors for the algorithm under consideration. Here, we want to depart from this course and consider the goal of optimizing the performance of the overall parallel system, assuming more than one job is available for execution. Thus the central issue of this paper is the question how the available processors of a parallel machine should be distributed among a number of jobs. In order to obtain guidelines for such multiprogramming control, we use the speedup-behaviour and the accumulated processor time of a job as its characterization.

1 Introduction

Traditionally, parallel computing systems are run in serial mode, i.e. there is at most one job being processed at a time, and the next job is not started before the prior one is completed. However, similar to conventional computing systems, there are some reasons for multiprogramming in parallel computing systems. Economy of usage, for example, is obviously an important motivation. For this, there are two viewpoints to consider: the system aspect based on computation cost, and the user aspect based on computation benefit (which depends on the relevance and the timeliness of the result). Generally, if the bottleneck resource is not utilized completely by a single job, multiprogramming will render better throughput at the price of higher response times. In a parallel computer, one has to find an optimal compromise between (intra-job) parallelism and (inter-job) degree of multiprogramming: it may be suboptimal to run the system with high parallelism, if the job cannot use the processors efficiently, instead of running several jobs at reduced parallelism. Therefore we aim towards guidelines for the selection of optimal parallelism and degree of multiprogramming.

We begin with an investigation of typical speedup types and how they can arise (Section 2). Then, we switch to defining profit of computations and reconsider how the optimal degree of multiprogramming is obtained for the serial computation case (Sections 3 and 4). We then derive our guidelines for the parallel case (Section 5) and give a summary of them in Section 6. Finally, we compare our results to related work (Section 7).

www.manaraa.com

2 Speedup–Characteristics

In section 5 we will see that the speedup–characteristic of the jobs has great influence on the optimal multiprogramming strategy. Therefore we first present and motivate an appropriate classification of speedup functions and show their relevance by some examples obtained with the parallel theorem prover RCTHEO.

2.1 Classification of Speedup Functions

Let the service time of a job executed on a single processor be $b(1)$ and, executed on p processors in parallel, $b(p)$. Then the ratio

$$s(p) = \frac{b(1)}{b(p)} \tag{1}$$

is called the **speedup** function of the job. There are the following basic types of speedup functions $s(p)$:

lin	linear:	$s(p) = p$
sub	sublinear:	$s(p) < p$
sup	superlinear:	$s(p) > p$
sat	saturated:	$s(p) < p; \lim_{p\to\infty} s(p) = s_{\max}$
ret	retrograde:	$s(p)$ drops after it has reached its maximum

Fig. 1 shows typical instances of the five basic shapes. The first three types ***sub***, ***lin***, and ***sup*** cannot occur in real parallel systems since they are unbounded. Every realistic speedup function has an upper bound (and thus is of type ***sat*** or ***ret***) since the parallel service time has a lower bound due to the discrete character of all computation proccesses.

Although the types ***sub***, ***lin***, and ***sup*** do not occur in practice, they can be used to describe the behaviour of a parallel system for a number of processors, up to which the saturation effects can be neglected. Clearly, the classification scheme is applied to the speedup function in the region $1 \leq p \leq k$ where k is the maximum number k of processors available on the actual hardware architecture.

2.2 Different Problems and their Speedup Functions

Most computational tasks can be classified into AND–problems and OR–problems[1]. An AND–problem is one which consists of a set of solvable subproblems which must all be solved, whereas for an OR–problem it is sufficient to solve only one of a set of subproblems. If an AND–problem with serial service time (= total amount of work) $b(1)$ is solved in parallel with p processors, the shortest possible service time $b(p)$ is equal to

$$b(p) = \frac{b(1)}{p}, \quad \text{and} \quad s(p) = p,$$

[1] For reasons of simplicity we do not consider AND/OR-problems as they are hierarchical combinations of AND- and OR-problems.

www.manaraa.com

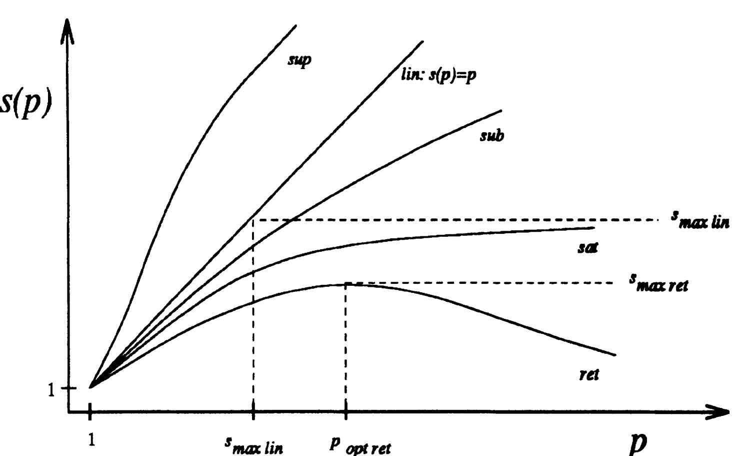


Fig. 1. Speedup-functions: *lin:* linear, *sub:* sublinear, *sup:* superlinear, *sat:* saturated, *ret:* retrograde.

if work can be partitioned into p independent equal parts and no overhead for parallel execution is performed. Hence, the best possible speedup function obtainable with AND–parallelism is of type *lin*.[2] Superlinear speedups of class *sup* are impossible in case of AND–parallelism, since the total amount of work is fixed[3] and equals $b(1)$.

In case of OR–parallelism, however, superlinear speedups are possible as we will show now. Assume an OR–problem Q consisting of n subproblems $Q_1, \ldots, Q_n$ of equal size u, i.e. after u computation steps the solver either returns a solution or a failure message for each subproblem. Suppose exactly one of the n subproblems (say the i-th subproblem) is solvable. If the sequential solver examines the subproblems in the order of increasing indices it will take $i \cdot u$ steps to solve the problem. For solving the problem in parallel, we assume that each of the p processors gets an equal number m of subproblems, i.e. $\exists m \in \mathbb{N} : \; n = m \cdot p$. We distribute the n subproblems onto the p processors, such that the j-th processor gets $Q_{(j-1)m+1}, \ldots, Q_{jm}$.

Now, let $p = 2$, $n = 6$, $i = 4$, i.e. we have the situation illustrated by

$Q_1 \; Q_2 \; Q_3$	$Q_4 \; Q_5 \; Q_6$
processor 1	processor 2

with two processors solving the six subproblems $Q_1, \ldots, Q_6$ and a solution

[2] Here we neglected that every speedup function has an upper bound.

[3] Due to e.g. memory contention effects in the sequential system, $b(1)$ can be more than the total amount of useful work to be done, which can lead to superlinear speedup. For a detailed discussion of such effects see [1].

www.manaraa.com

yielded by Q_4. Then we get $b(1) = 4u$, $b(2) = 1u$ and

$$s(2) = 4.$$

It can be shown[4] that $s(p) < p$ for the above problem, if we use average service times to compute the speedup as

$$s(p) := \frac{\overline{b(1)}}{\overline{b(p)}}$$

with

$$\overline{b(1)} = \frac{1}{n!} \sum_{\substack{\text{all permutations of} \\ \{Q_1,\ldots,Q_n\}}} b(1) \qquad \text{and} \qquad \overline{b(p)} = \frac{1}{n!} \sum_{\substack{\text{all permutations of} \\ \{Q_1,\ldots,Q_n\}}} b(p).$$

This result only holds for subproblems of equal size. In the next example we will see that for subproblems of different size superlinear speedups are still possible. Let Q_1, Q_2, Q_3, Q_4, Q_5 be of size $100u$ and Q_6 of size u. Then the average sequential service time is $\overline{b(1)} = 251u$. With 6 processors we get $\overline{b(6)} = 1u$ which results in s(6) = 251. This shows that strongly superlinear speedups appear quite naturally in OR-parallel problem solving.

These superlinear speedups are caused by the fact that the amount of work w to be done for solving the problem is, in case of OR-parallelism, not constant w.r.t. the number of processors. In case of time-dependent degree $p(t)$ of parallelism we define w as

$$w = \int_0^{b(p)} p(t)\, dt.$$

For constant $p(t) = p$ during service time the integral simplifies to $w = b(p) \cdot p$.

As a matter of fact all five different types of speedup-function occur in OR-parallel search. In figure 2 we show a typical speedup-curve obtained from average run-times of the parallel automated theorem prover RCTHEO.[5] This plot shows for small, intermediate and large p the different types *sub*, *sup* and *sat* in one speedup-figure.

Figure 3 shows which speedup-types can occur for different classes of problems and algorithms characterized by their *CPU-utilization* and their work w as defined above. The upper part of the circle represents full CPU-utilization, whereas the lower part stands for partial CPU-utilization (loss of efficiency) through communication, load-imbalance, etc. The left part covers systems which show a monotone increase of w with increasing p. This is typically caused by CPU-time (not waiting-time!) spent for communication, coordination-overhead,

[4] In case of one solution (as above) the proof is very easy: $\overline{b(1)} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}$, $\overline{b(p)} = \frac{p}{n}\sum_{i=1}^{n/p} i = \frac{n+p}{2p}$ and we get $s(p) = \frac{(n+1)p}{n+p} < p$.

[5] The OR-parallel theorem prover RCTHEO [2, 3] is based on SETHEO, a Prolog technology based automated theorem prover for first order predicate logic [4]. SETHEO uses backtracking to search for a proof in an OR-search-tree. The parallel search algorithm used by RCTHEO is called random competition.

المنارة للاستشارات
www.manaraa.com

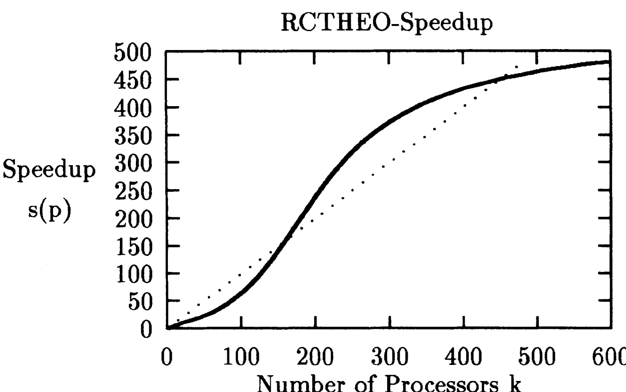


Fig. 2. Speedup-function obtained with the OR-parallel theorem prover RCTHEO for the proof of a group-theory problem.

etc. In the right part we have decreasing w with increasing p, which can only be caused by algorithms whose efficiency increases with increasing number of processors.

As already mentioned above in case of AND-parallelism, if we have full CPU-utilization and constant w, the resulting speedup shows linear behaviour. If the work w decreases with increasing p and CPUs are fully utilized ($p(t) = p = constant$) we get

$$s(p) = \frac{b(1)}{b(p)} = p \cdot \frac{b(1)}{w(p)} > p,$$

i.e. in the upper right part of figure 3 we observe superlinear speedup, which occurs in OR-parallelism (see example above). If however communication or other overhead causes increasing w, only sublinear, saturated, or retrograde speedup behaviour is possible (upper left part). Although the same types of speedup characteristics occur, speedup becomes even worse, if the CPUs are not fully utilized.

Even if CPUs are only partly utilized, the speedup may show superlinear behaviour, depending on the trade-off between CPU-utilization and decrease of w, as shown in the lower right quarter of the circle. In this region, however, all other speedup-types can occur as well.

3 Benefit, Cost, and Profit

For the assessment of scheduling strategies, Greenberger ([5]) introduced priority functions for individually weighted jobs with different urgencies. We adopt his idea by using benefit functions which give the benefit bf provided by the result of the computation as a function of the response time y, (see fig. 4).

www.manaraa.com

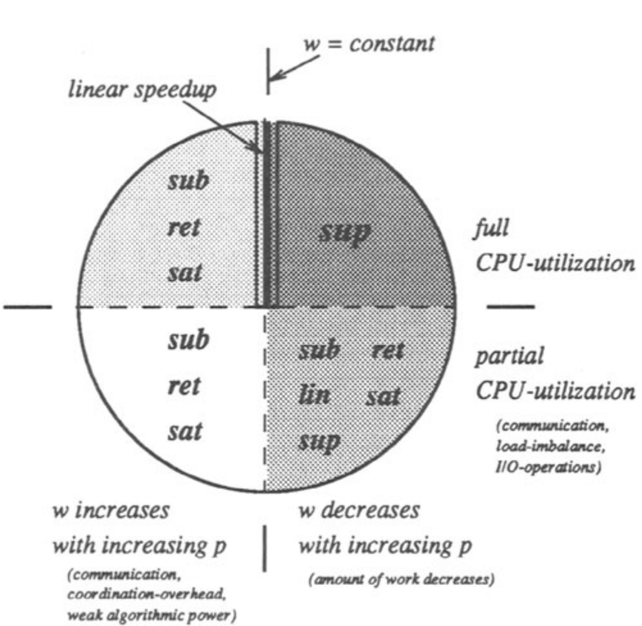


Fig. 3. Speedup-types in the space of work w and CPU-utilization.

Generally, the larger the response time, the smaller the benefit. In many cases, the benefit will drop sharply for a given response time (deadline, realtime constraint).

We assume the benefit to be measured in money. This makes the benefit commensurable with the cost of the result. The difference of benefit and cost shall be called profit. For a computing facility, an optimal operational strategy would be maximizing the profit rate. By this we define the sum of the profit gained in our benefit/cost model, divided by the time interval in which we sum up profit. Correspondingly, we define a benefit rate and a cost rate of the system operation and the system. The cost rate of a system is constant, independent of its utilization, as long as we do not consider changes in configuration. This simplifies our analysis.

4 Profit of Serial Computations

To understand the consequences of our terms of benefit, cost and profit rate and their dependence on the degree of multiprogramming, we first consider a monoprocessor computing system which executes a stream of serial (i.e., not internally parallel) computations, f jobs (degree of multiprogramming) at a time. It is well known ([6]) that, due to Little's Formula and the existence of the mean

www.manaraa.com

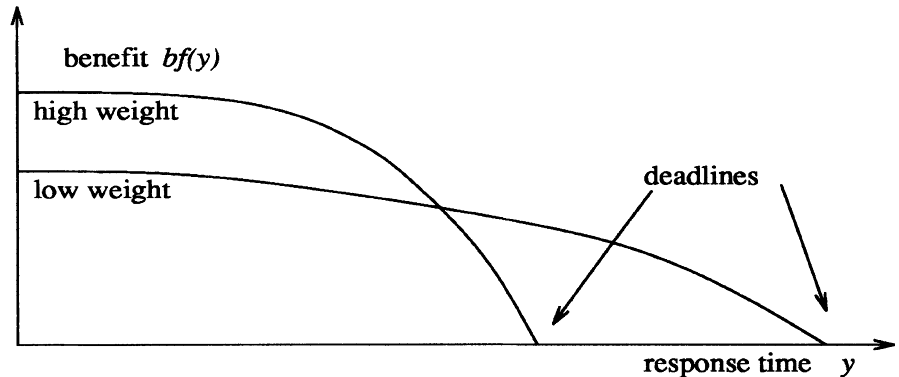


Fig. 4. Benefit functions $bf(y)$; y response time.

service time $\overline{b}$ as a minimal mean response time $\overline{y}$ and the bandwidth c as the maximal throughput of the system, there are basic relations between mean response time $\overline{y}$, throughput d, and mean degree of multiprogramming $\overline{f}$ (see fig. 5). Of particular importance is the saturation degree of multiprogramming, $f^* = \overline{b} * c$, as it separates the range of low load from that of high load and as the value of f^* gives the throughput gain that can be achieved by multiprogramming.

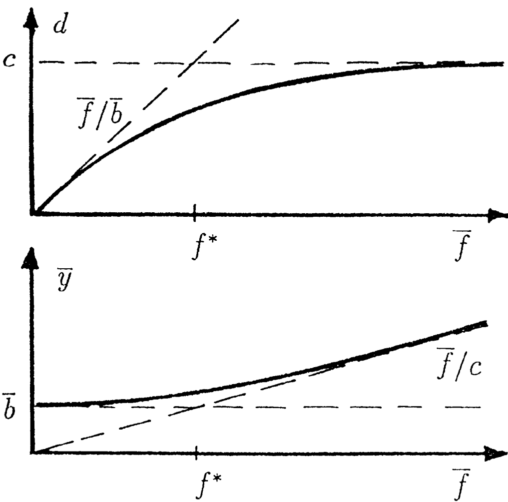


Fig. 5. Throughput d and mean response time $\overline{y}$ as a function of the mean multiprogramming degree $\overline{f}$ of the system. c : system bandwidth (maximal throughput), $\overline{b}$: mean service time, f^* : saturation degree of multiprogramming.

www.manaraa.com

To see the influence of the mean multiprogramming degree $\overline{f}$ on the profit rate, see fig. 6. By mapping the benefit $bf(\overline{y})$ via $\overline{y}(\overline{f})$ on $\overline{f}$, we get $bf(\overline{f})$, i.e. benefit as a function of the degree of multiprogramming; $bf(\overline{y})$ depends on the distribution of y because of the nonlinear $bf(y)$. Notice that $bf(\overline{f})$ will be roughly constant up to f^* and will then follow the shape of $bf(\overline{y})$, though somehow stretched. This form of our benefit function opens the way to the benefit rate, which is

$$bfr(\overline{f}) = d(\overline{f}) * bf(\overline{f}) \tag{2}$$

as any completed computation (average d per time unit) will render a mean profit bf and thus establish the profit rate bfr. Graphically, this multiplies the plot $bf(\overline{f})$ with that of $d(\overline{f})$ which was already considered in fig. 5. A major insight is that the benefit rate is maximized in the neighbourhood of f^* as below there is the (nearly) linear rise of the throughput and above there is the generally declining characteristic of the benefit function. The profit rate is the difference between benefit rate and cost rate. The latter is constant for a given configuration, as we stated above.

Clearly, at a low degree of multiprogramming, the system is not profitable because the throughput is too low. At a high degree it is not profitable because it cannot deliver results in time. Our result is roughly independent of the shape of the benefit function $b(\overline{y})$ if this is declining at all.

5 Profit of Parallel Computations

For the analysis of the optimal operation of a parallel processing system, we use the same technique as in the serial case. The parallel case, however, is more difficult, because there are two degrees of freedom in parallelism: intra-job parallelism p and inter-job degree of multiprogramming f. Their product may equal or exceed the system capacity k (number of processors); in the latter case we have a multiplexed mode on processors. Obviously, assuming there are enough tasks available, it should be avoided not to use all processors (i.e. avoid $p*f < k$).

We will assume that the computational resource (given by the set of processors) is the bottleneck of the system, i.e. its utilization is higher than that of any other resource. So, the system bandwidth c is limited by the bandwidth of the set of processors, and

$$c = \frac{k}{\overline{w}} \tag{3}$$

where k is the number of processors and $\overline{w}$ is the mean work per task.
If the mean processor utilization under uniprogramming is ρ, a throughput gain of up to $1/\rho$ is achievable by multiprogramming.

Multiprogramming, however, is also advisable, if the tasks are able to use the processors continuously ($\rho = 1$), but show a steep increase of mean work $\overline{w}$ under rising parallelism, such that the service time remains constant or increases (*saturated* or *retrograde* speedup case). Then, multiprogramming by (maybe dynamical) partitioning of the processor system is advisable; each task

www.manaraa.com

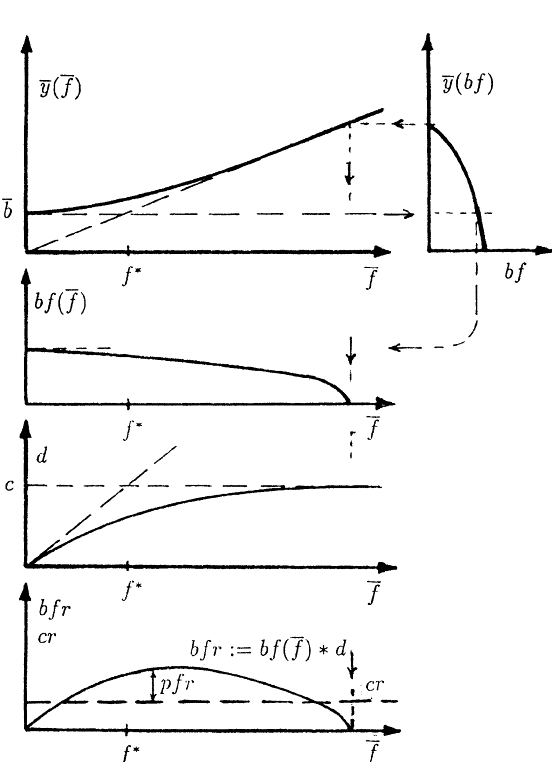


Fig. 6. Mean response time $\overline{y}$ as a function of mean multiprogramming degree $\overline{f}$ and of benefit bf; benefit bf, throughput d and benefit rate bfr as functions of $\overline{f}$. $bf(\overline{f})$ and $bfr(\overline{f})$ are constructed by functional composition. In the bottom diagram, the cost rate characteristic cr of the system is included, leaving the profit rate pfr as the difference between benefit rate bfr and cr.

gets a partition with a subcapacity at which there is no saturation or retrograde speedup.

Generally, we follow the analysis pattern which we used for the serial system. Mean service time and system bandwidth are now, however, not constant, but functions of the (intra-) task parallelism p. For the service time, this function is usually expressed indirectly by the speedup. For our analysis, the direct representation, as in fig. 7, is more favourable.

Similarly as service time, the mean work $\overline{w}(p)$ may be constant, rising, or declining. Speedup characteristic and $\overline{w}(p)$ characteristic are not independent (see fig. 3).

www.manaraa.com

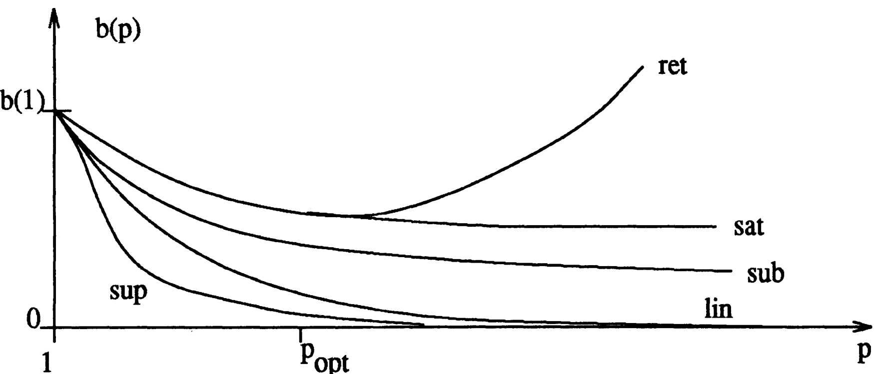


Fig. 7. Service time $b(p)$ under parallelism p for different speedup characteristics.

So we have a system bandwidth

$$c(p) = \frac{k}{\overline{w}(p)} \tag{4}$$

which is also a function of p.

Let us first assume $c(p) = constant$, i.e. $\overline{w}(p) = b(1)$; thus parallelism does not change mean work. Our analysis leads to fig. 8. Reducing the service time is the only effect of rising parallelism; however, under saturated or retrograde speedup, there is a lower bound for service time, beyond which service time is not reduced any more.

The resulting benefit rate function $bfr(p)$ shows for linear or sublinear speedup that maximizing p will maximize the benefit rate and that this will imply a small degree of multiprogramming, near $f^*(p)$. More precisely,

$$f^*(p) = b(p) * c(p) \tag{5}$$

$$= \frac{b(1)}{s(p)} * \frac{k}{b(1)} = \frac{k}{s(p)} \tag{6}$$

In the linear case, we have

$$s(p) = p$$

and

$$f^*(p) = \frac{k}{p} \tag{7}$$

as $p \leq k$, $f^*(p) \geq 1$. $bfr(p)$ is maximized by $p = k$ which gives $f^*(k) = 1$. So we have uniprogramming as the operation mode with the highest benefit rate. For a sublinear case, we may assume e.g.

$$s(p) = \alpha * p \qquad (0 < \alpha < 1)$$

www.manaraa.com

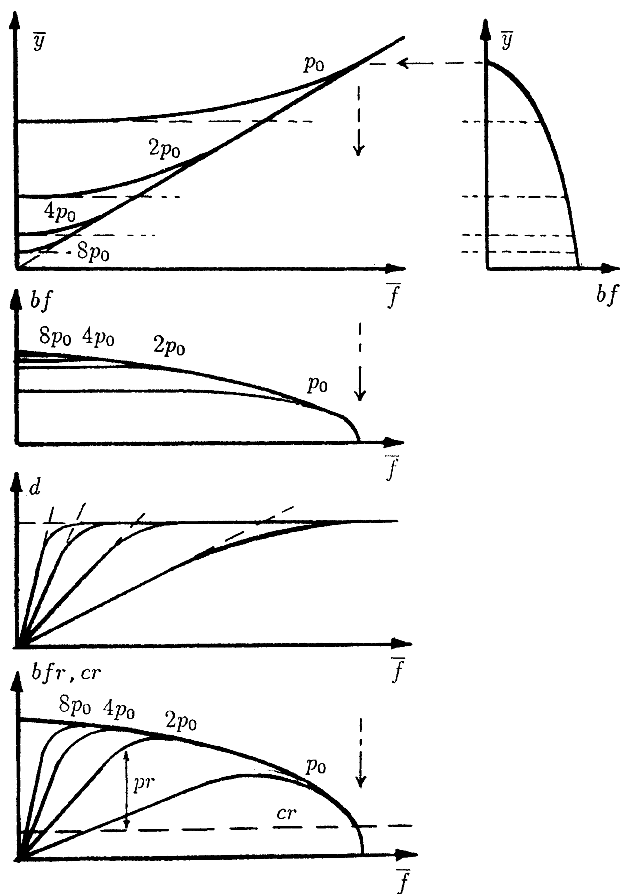


Fig. 8. Benefit rate bfr and profit rate pfr under various degrees of parallelism p_0, $2p_0$, $4p_0$, $8p_0$, constructed as in fig. 4, for constant system bandwidth c.

above a certain $p > 1$ (in compliance with the requirement $s(1) = 1$), and get

$$f^*(p) = \frac{k}{\alpha * p} \tag{8}$$

and at least ($p = k$)

$$f^*(k) = \frac{1}{\alpha} > 1$$

As $\overline{w}(p) = constant$, α reflects a mean underutilization of the processors by $(1 - \alpha) * 100\%$. It is optimal to compensate this by a multiprogramming degree in the range of $1/\alpha$.

www.manaraa.com

In the saturated or retrograde case, there is no shorter service time beyond $p = p_{opt}$.
So, we have

$$f^*(p) = \frac{k}{p_{opt}} > 1 \quad \text{for} \quad p_{opt} < k$$

in the best case, and multiprogramming in the range of $\frac{k}{p_{opt}}$ should be applied. We now omit our prior assumption $\overline{w}(p) = constant$, i.e. parallelism changes mean work. The results are shown in fig. 9. Rising p will (except for the saturated and retrograde cases) decrease service time $b(p)$, but it may increase or decrease mean work $\overline{w}(p)$.

In the case of rising $\overline{w}(p)$ we have two adverse effects on the benefit rate: decreasing service time improves the behaviour at low values of $\overline{f}$, while the decreasing bandwidth deteriorates the behaviour at high values of $\overline{f}$. Whether this leads to optimal points of operation at high p and low $\overline{f}$, as seen before, depends on the relative effect of p on service time b and bandwidth c.

In the case of falling $\overline{w}(p)$ the arguments are much simpler. This case, which is implied by superlinear speedup, of course recommends to choose high parallelism p.

In both cases, underutilization of processors makes multiprogramming optimal in the range of $f^*(p)$ according to formula (6). In the case of falling $w(p)$, it can be preferable not to introduce further jobs to use the idle processor time, but to choose $p > k$, as this will not only lead to full processor utilization but also deliver a further reduction of $w(p)$!

6 Consequences for Parallelism and Multiprogramming

Our analysis has been restricted to tasks of equal benefit functions. The important case of different benefit functions is much more complex; it also opens the scene for benefit–dependent service strategies.

Furthermore, we assumed that the processors are exchangeable in their role and neglected the multiprogramming overhead, such as for control and context switching. Multiprogramming needs additional memory; if our parallel computing system needs replicated code, this can be an important cost factor. However, our model could respect this by a cost rate rising with $\overline{f}$.

Principally, of course, one can simulate the favourable behaviour of a parallel computing system under jobs with declining $w(p)$ by a time slice simulation on a serial computing system and win an equal advantage. Technically, one has to choose the size of the time slices so that one gets an optimal balance between time slice overhead (which is in favour of a long slice) and preferential treatment of the short subproblems (which excludes time slice longer than the execution time of the short subproblems). As the execution times are not known, however, one is in danger to miss either efficiency or selectivity of the regime.

Our arguments for multiprogramming were mainly economical. Considering a parallel computation server, there are also reasons for an adequate response ratio (i.e. ratio of service time to response time) in favour of multiprogramming.

www.manaraa.com

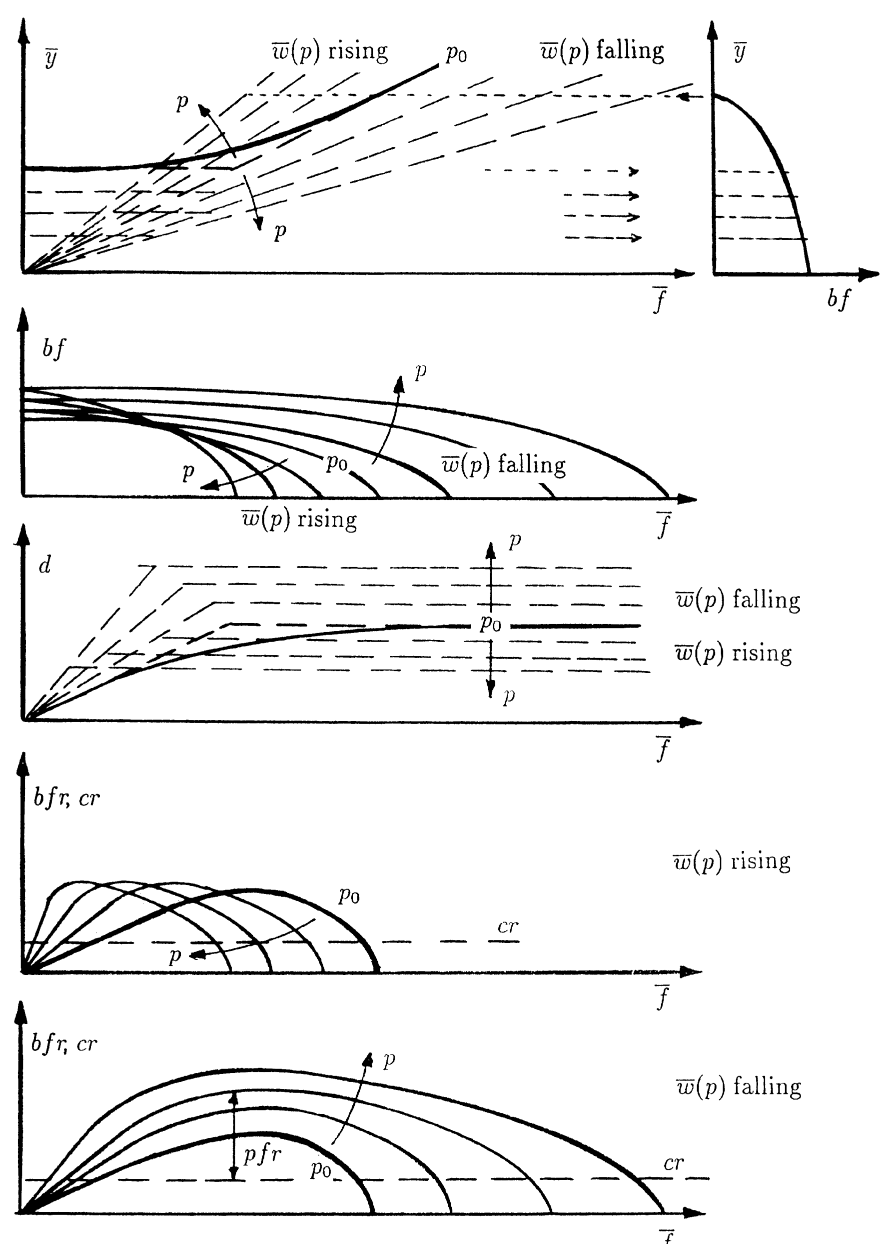


Fig. 9. Benefit rate bfr and profit rate pfr under various degrees of parallelism p, constructed as in fig. 4/6, for the rising/falling $\overline{w}(p)$–cases. $\overline{y}$ mean response time, p parallelism, $\overline{w}$ mean work, $\overline{f}$ mean multiprogramming degree, bf mean benefit, d throughput, bfr benefit rate, cr cost rate, pfr profit rate.

www.manaraa.com

So far, our recommendations for maximizing the profit, which need more detailed analysis in some cases, are:
(1) Choose parallelism p such that the benefit rate is maximal. This implies

- for the saturated and retrograde case $p \leq p_{opt}$. If $p_{opt} < k$ and the tasks have high processor utilization, the system should be partitioned under multiprogramming if this reduces overhead.
- else $p = k$ for the case of constant or decreasing mean work $\overline{w}(p)$; otherwise the optimal p will depend on the relative effect of p on service time $\overline{b}$ and on work $\overline{w}$.

(2) Choose multiprogramming with a degree in the range of $f^*(p)$ (or slightly above) whenever $f^*(p) = b(p) * c(p) \geq 1$, where p is selected as stated above.

We may sum up our guidelines in the work/utilization diagram fig.10 (compare also fig.3). The diagram establishes the relation between types of speedup functions, degree of parallelism, and degree of multiprogramming.

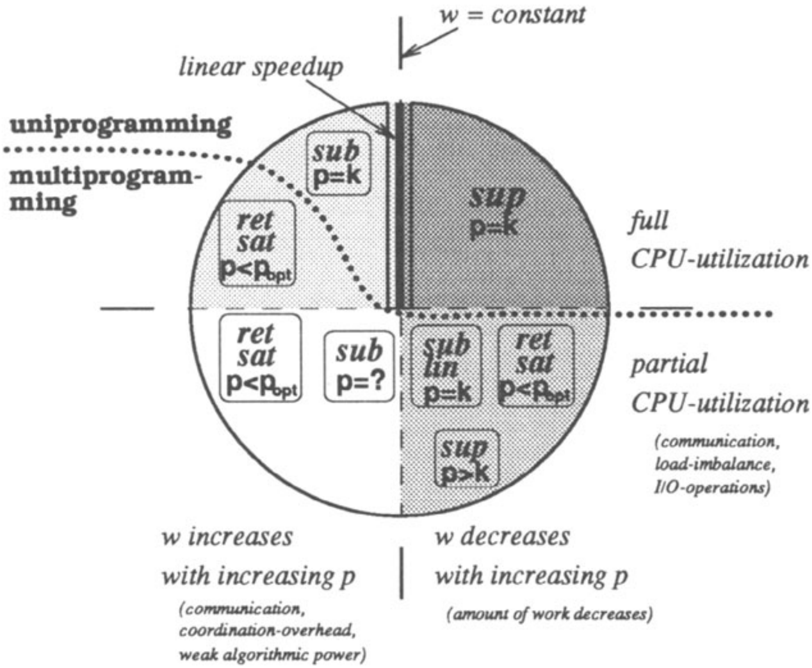


Fig. 10. Guidelines for parallelism p and uni/multiprogramming in a parallel system with k processors, depending on mean work per task and processor utilization.

www.manaraa.com

7 Comparison with Related Work

In this section, we want to set our approach and it's results into perspective to related research and common practice.

It is well known that the number of processors which maximizes the speedup for a parallel algorithm may not lead to an efficient use of processors (e.g. [7], [8], [9]). This is easily understood considering the common speedup characteristic *sat* (see figure 1): after a certain point, a large number of additional processors is required just to obtain tiny speedup improvements. Thus, the ideal working point for a particular task should optimize the speedup over cost ratio, rather than speedup alone. Various researchers suggested therefore to minimize $p \times (b(p))^r$ ([7], [9]) or, equivalently for $r = 2$, to maximize $E \times S$ ($= \frac{S^2}{p} = \frac{b(1)^2}{p \times b(p)^2}$; efficiency $E = \frac{S(p)}{p}$) instead ([8]). The most general results in that regard (including a good sketch of related approaches) are found in [9], where $p \times (b(p))^r$ is minimized depending on the overhead characteristics (overhead as a function of p) of the task. Raising $b(p)$ to it's rth power allows the system designer to give a preference to speedup or utilization, whichever is the operational goal for the system (see also [10]). For our considerations, the effect of the parameter r is hidden in the benefit function. That function defines the penalty for trading speedup for efficiency. Thus, in effect, it defines the operational goals for the affiliated program. Since we assumed the same benefit function for all tasks, this amounts to a coverage of the operational goals for the parallel system.

In [11], the power of a parallel system is defined as processor utilization divided by the mean response time. Besides an analysis deriving the number of processors which maximizes the power for a single task for different workload models, the authors also evaluate the case where tasks arrive at a Poisson rate (but tasks can be processed only one at a time, i.e., no multiprogramming). They found that the number of processors which maximizes the power is independent of the task arrival rate for their model. However, as pointed out, this result does not apply for multiprogramming.

In [12], Eager et.al. propose the maximum of the ratio of efficiency to execution time as a desirable point of operation ("knee in the execution time-efficiency profile"). They argue that taking efficiency and execution time as indicators for payoff and cost, respectively, or doing so vice versa, leads to two different operational objectives, both of which are optimized at the knee (these objectives are achieving efficient processor usage taking execution time as a cost measure versus achieving low execution times taking efficiency as a cost measure). They remark that for multiprogramming, low efficiency should be avoided by allocating processors to tasks according to the number of processors proposed by the knee in the program's execution time-efficiency profile.

In summary, related research so far mainly concentrated on the optimization of the speedup or execution time over cost ratio for individual tasks. For measuring the cost of a parallel computation, either efficiency or execution time have been used. In both cases, often the traditional definition of execution time as the number of processors times parallel execution time ($p \times b(p)$) has been used (with

www.manaraa.com

the exception e.g. of [11]). However, defining the expenditure of resources for the parallel computation as $p \times b(p)$ is only accurate assuming the exclusive usage of the parallel machine. While this has been the standard for a long time, nowadays typical parallel machines can be used in multiprogramming environments. The ability of a parallel machine to handle requests for and freeing of nodes during the computation of an application requires to take into account dynamically changing numbers of processors for an accurate definition of computational costs. In order to do so, we proposed to take the amount of work actually spent for a parallel computation ($W(p)$) as a measure for its computational costs (note that $W(p) \leq p \times b(p)$ for any algorithm, i.e. the new definition results always in less or equal costs than the traditional one). It should be added that for realistic taxation strategies the total system costs should be distributed among the users (viz. programs), which means that idle times are also proportionally charged for. Then, a fair assessment of computational costs could be based on $\frac{W(p)}{\epsilon}$, where ϵ is the system utilization of the maximally requested system fraction during the computation.

In summary, the major differences between our approach and other research are the goal of optimization and the assessment of computational costs. As described above, researchers so far have been interested in the number of processors that optimize important criteria of the performance of a single program, typically based on speedup and efficiency characteristics of the program. For multiprogramming systems, those results lead to a straightforward and simple operating strategy, where each task obtains the number of processors which optimize its performance (local optimization). In this paper, however, we approached optimization from the viewpoint of overall system optimization, accepting that this might not be optimal for a particular task (global optimization).

Acknowledgements.
Many thanks to Heidi Brückner for her help in preparing this manuscript.

References

1. D.P. Helmbold and C.E. McDowell. Modeling Speedup(n) greater than n. In *Proceedings of the International Conference on Parallel Processing*, pages III–219–225, 1989.
2. W. Ertel. OR-Parallel Theorem Proving with Random Competition. In *Proceedings of LPAR'92*, pages 226–237, St. Petersburg, Russia, 1992. Springer LNAI 624.
3. W. Ertel. *Parallele Suche mit randomisiertem Wettbewerb in Inferenzsystemen*, volume 25 of *DISKI*. Infix-Verlag, 1993.
4. R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: A High-Performance Theorem Prover. *Journal of Automated Reasoning*, 8(2):183–212, 1992.
5. M. Greenberger. The Priority Problem. Technical Report MIT-MAC 22, 1965.
6. R.R. Muntz and J. Wong. Asymptotic Properties of Closed Queueing Network Models. *Proceedings of the Eight Annual Princeton Conference on Information Sciences and Systems*, 1974.

www.manaraa.com

7. M. Barton and G. Withers. Computing Performance as a Function of the Speed, Quantity, and Cost of the Processors. In *Proceedings of the 1989 International Conference on Supercomputing*, pages 759–764, 1989.
8. H.P. Flatt and K. Kennedy. Performance of Parallel Processors. *Parallel Computing 12*, pages 1 – 20, 1989.
9. A. Gupta and V. Kumar. Analyzing Performance of Large Scale Parallel Systems. Technical report, University of Minnesota, 1992.
10. L. Kleinrock. Power and Deterministic Rules of Thumb for Probabilistic Problems in Computer Communications. In *Proceedings of the International Conference of Communications*, pages 43.1.1–43.1.10, 1979.
11. L. Kleinrock and J.-H. Huang. On Parallel Processing Systems: Amdahl's Law Generalized and Some Results on Optimal Design. *IEEE Transactions on Software Engineering*, 18(5):434–447, 1992.
12. D.L. Eager, J. Zahorjan, and E.D. Lazowska. Speedup Versus Efficiency in Parallel Systems. *IEEE Transactions on Computers*, 38(3), 1989.

www.manaraa.com

The Distributed Hardware Monitor ZM4 and its Interface to MEMSY

Richard Hofmann
email: rhofmann@immd7.informatik.uni-erlangen.de

Universität Erlangen, IMMD VII

Abstract

Computers, especially parallel and distributed systems, are highly complex. Even more complex a task is programming such systems in order to get them run correctly and efficiently. This is due to the manyfold interactions between the processes running at the same time while carrying out a common task. Monitoring is a valuable technique to analyze and understand the dynamic behavior of parallel and distributed software. We prefer hybrid monitoring, a technique which combines advantages of both software monitoring and hardware monitoring.

The paper contains a description of a hardware monitor ZM4 which makes our concepts available to programmers, assisting them in debugging and tuning of their code. This is followed by a short survey of the interfaces between ZM4 and a variety of object systems, which make the ZM4 a universal monitor system. Finally, the interfacing to MEMSY is described as the result of a fruitful cooperation between the hardware, software and performance evaluation group in the project.

Keywords: *hardware monitoring, hybrid monitoring, event-driven monitoring, instrumentation, performance evaluation, debugging, parallel and distributed systems, MEMSY.*

1 Introduction

Users and operators of parallel and distributed systems often find it very difficult to exploit the immense computing power at their disposal. Writing and debugging parallel programs which use the underlying hardware in an efficient way proves to be a difficult task even for specialists. There is typically not enough insight into the internals of the hardware, the system software and their corresponding effect with the user program. Bugs are hard to locate, and tuning, which depends on a detailed knowledge of such factors as idle times, race conditions or access conflicts, is often not done systematically but by using ad-hoc methods. To analyze the functional behavior and the performance of a parallel program it is not enough to employ standard methods such as profiling and accounting. Sophisticated methods and tools are needed to handle these issues.

Event-driven monitoring is a technique well-suited for analyzing programs running on a parallel or distributed system. It can be done by hardware, software or hybrid monitoring, where the last combines advantages of both hardware monitoring and software monitoring. In order to be able to cope with both sorts of hardware oriented monitoring, we built a universal distributed hardware monitor system (ZM4), which can be adapted to arbitrary object systems (the system on which the program under study is

www.manaraa.com

running). Additionally, ZM4 is scalable, and it has a high-precision global clock which allows to monitor several nodes of the object system simultaneously, providing globally valid time stamps.

The paper is organized as follows: after briefly stating our notion of hybrid monitoring, the basics of the hardware monitor ZM4 are described. We then present an overview of the currently available interfaces between the ZM4's infra structure and different object systems, emphasizing the fact that building a new interface to a universal monitor is the same as building a complete monitor for that object system. Finally the interface to MEMSY will be shown from concept to implementation both in the hardware part and the driver software.

2 Hybrid Monitoring

Event-driven monitoring represents the dynamic behavior of a program by a sequence of events. It is the only monitoring method (in contrast to time-driven monitoring) suitable for efficient program analysis [6], as the aim of monitoring is gaining insight into the dynamic behavior of a parallel program. Time-driven monitoring (sampling) only provides statistical information about program execution and is therefore insufficient for behavior analysis. An event is an atomic instantaneous action. The definition of events depends on the monitoring technique used. There are three monitoring techniques: hardware, software and hybrid monitoring.

Using hardware monitoring, the event definition and recognition can be difficult and complex. An event is defined as a bit pattern on a processor bus or in a register. It is detected by the probes and detection circuitry of a hardware monitor. In this case it is difficult to relate the recorded signals to the monitored program, i.e. to find a problem-oriented reference. Using software or hybrid monitoring, the events are defined by inserting monitoring instructions at certain points in the program under investigation (program instrumentation). These instructions write an event description into a reserved memory area of the monitored system (software monitoring), or to a hardware device which is accessible for a hardware monitor (hybrid monitoring).

In defining events by program instrumentation, each monitored event can uniquely be associated with a point in the program; this provides a source-related reference. Thus, the evaluation of the event trace can be done on the program level which is familiar to the program designer. As hybrid monitoring combines source-related event specification with a small interference on the object system's behavior, it is our favorite monitoring technique.

Whenever the monitor device recognizes an event, it stores a data record (a so-called event record). An event record contains the information what happened when and where and consists of at least the event description and a time stamp. The time stamp is generated by the monitor and represents the acquisition time of the event record. Beside these fields, an event record can contain optional fields describing additional aspects of the event occurred. The sequence of events is stored as an event trace.

It is strongly recommended to wisely restrict instrumentation to essentials. One reason is that CPU time overhead increases with the number of events issued. The other reason is that a problem should be analyzed on an adequate level of abstraction.

www.manaraa.com

Therefore instrumentation should be limited to those events whose tracing is considered essential for an understanding of the problems to be solved. Bearing this common prerequisite in mind, monitoring is a great help when analyzing programs running in modern parallel or distributed systems.

3 ZM4 — a Universal Distributed Monitor System

3.1 Demands and Conceptual Issues

A monitor system, universally adaptable to arbitrary parallel and distributed computer systems, must fulfil several architectural demands. It must be able to

(a) deal with a large number of processors (nodes in the object system),
(b) cope with spatial distribution of the object nodes,
(c) be adaptable to different node architectures,
(d) supply a global view on all interesting events in the object system for showing causal relationships between events in different nodes,
(e) provide a problem-oriented (source-related) view.

We have designed and implemented a universal distributed monitor system, called ZM4 (abbreviation for German "Zählmonitor 4"), which fulfils the demands (a) - (d). Its concepts for meeting these challenges are:

(a) In order to deal with a large number of object nodes the monitor ZM4 has a distributed architecture, scalable by allowing an arbitrary number of monitor agents.
(b) ZM4 interconnects the monitor agents by a local area network. Therefore, monitor agents need not be spatially concentrated and can also monitor spatially distributed object systems.
(c) ZM4 is not dedicated to just one object system but can record events from arbitrary object systems with arbitrary physical event representation.
(d) ZM4 has a global clock with an accuracy of 100 ns. This provides sufficient precision for establishing a global view in any of today's parallel and distributed systems.
(e) A problem-oriented view can be achieved by representing measured events and activities by the identifiers familiar to the programmer.

Issues (a) – (d) are dealt with in this section, containing the description of the architecture of ZM4 and its major component DPU. Issue (e) is rather a problem of object instrumentation than of monitoring hardware. However, the ZM4 hardware monitor supports (e) by accepting a wide variety of physical event formats.

The following considerations are important for the notion of global view in distributed systems (see also [4]): monitoring distributed systems or multiprocessors provides an event stream for each processor. When processors are working on a common task, they have to exchange information, resulting in an interdependence of their event streams. One concept to globally reveal all causal relationships is to order events. It suffices to locally order the events of each processor and to globally order events concerning interprocessor communication. Since local ordering is automatically achieved if the events are recorded in the order of their occurrence, we can restrict the following arguments to global ordering.

www.manaraa.com

In systems communicating via message passing a global ordering of the communication events can be achieved by the inherent causality of Send- and Receive-operations [8]. If monitoring shall provide not only a functional sequence of correctly ordered events but also performance, it is necessary to introduce time. Duda et al. describe a mechanism to estimate a global time from local observations in systems communicating via message passing [1].

Systems communicating via shared variables lack this easy mechanism to globally order events and to derive a global time. As the read-access to a shared variable can immediately follow the (state-changing) write-access, two consecutive accesses to a shared variable must be ordered correctly. Thus, only a monitor clock with a global resolution less than the access time to the shared memory allows to globally order communication events in systems with shared variables. As these demands of time resolution exceed those for ordering Send/Receive-events by orders of magnitude, a monitor using a clock with an accuracy less than half of the access time to shared memory can be used universally.

3.2 Architecture of the Hardware Monitor System ZM4

The ZM4 monitor system is structured as a master/slave system with a control and evaluation computer (CEC) as the master and an arbitrary number of monitor agents (MA) as slaves (see fig. 1). The distance between these MAs can be up to 1,000 meters. Conceptually, the CEC is the host of the whole monitor system. It controls the measurement activities of the MAs, stores the measured data and provides the user with a powerful and universal toolset for evaluation of the measured data [9].

The MAs are standard PC/AT-compatible machines equipped with up to 4 dedicated probe units (DPUs). We use their expandability for configuring ZM4 appropriately to the various object systems. Each MA provides processing power, memory resources, a hard disk and a network interface for access to the data channel. The MAs control the DPUs and buffer the measured event traces on their local disks. The DPUs are printed circuit boards which link the MA and the nodes of the object system. The DPUs are responsible for event recognition, time stamping, event recording and for high-speed buffering of event traces.

A local clock with a resolution of 100 ns and a time stamping mechanism are integrated into each DPU. The clock of a DPU gets all information for preparing precise and globally valid time stamps via the tick channel from the measure tick generator (MTG). Time stamps in a physically distributed configuration may be adjusted after the measurement according to the known wire length. While the tick channel together with the synchronization mechanism is our own development, we used commercially available parts for the data channel, i.e. ETHERNET with TCP/IP. The data channel forms the communication subsystem of ZM4, and it is used to distribute control information and measured data.

The ZM4's architectural flexibility has been achieved by two properties: its interfacing concept and a scalable architecture. The DPU can easily be adapted to different object systems, which will be discussed in sec. 4. ZM4 is fully scalable in terms of MAs and DPUs. The smallest configuration consists of one MA with one DPU (see fig. 1, left), and can monitor up to four object nodes. Larger object systems are matched by

www.manaraa.com

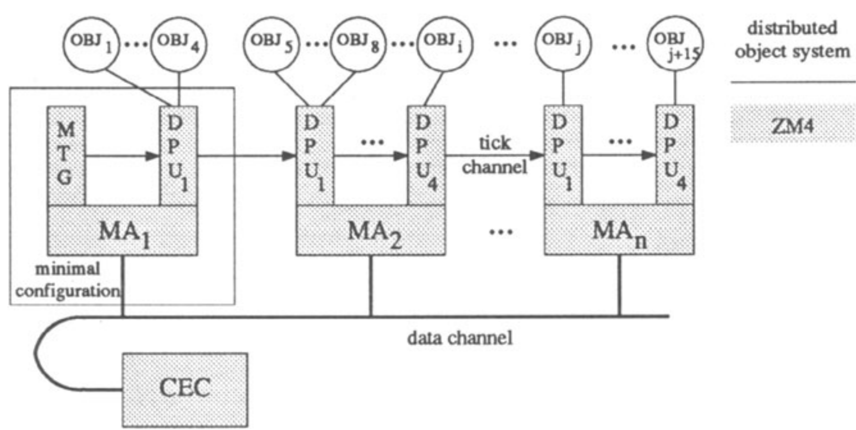


Fig. 1: Distributed Architecture of ZM4

more DPUs and MAs, respectively. In the following, the DPU architecture, the event recorder and the globally synchronized clock are discussed in a top-down fashion.

3.3 DPU Architecture

The DPUs implement a functional separation into the three tasks of event processing: interfacing, event detection and event recording (see general DPU in fig. 2, left).

Dedicated DPU-Parts: The interface has a tight connection to the object system, so it cannot be universal but must be dedicated to the object system. The event detector investigates the rapidly changing information supplied by the interface in order to recognize the events of interest, and to supply the event recorder with appropriate information about each event. The complexity of the event detector largely depends on the type of measurement: to recognize predefined statements in a program running on a processor without instruction cache and memory management unit, a set of comparators or a memory-mapped comparison scheme suffices. If the object system uses a processor with a hardware cache, or if predefined sequences of statements are intended to trigger an event, much more complex recognition circuits will be necessary [7]. In some cases of hybrid monitoring, the object system itself presents the event description in form suitable for the external monitor. In this case no event detector is needed, and the interface only has to adapt the object system to the event recorder electrically and mechanically. So the event recorder directly captures the event description, which is prepared by the object system (see simple DPU in fig. 2, right).

Universal DPU-Part: The universal part of a DPU is the event recorder. It is completely independent of the object system. It receives a bit pattern from the event detector or the hybrid interface, triggered by a signal for its occurrence. Its functionality includes event capturing, time stamping, event record definition and event record buffering.

www.manaraa.com

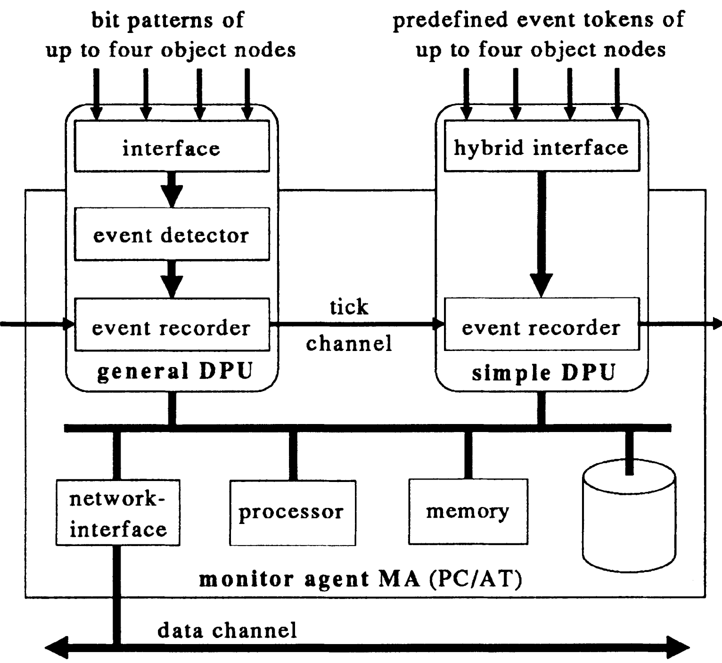


Fig. 2: Monitor Agent equipped with DPUs

3.4 Universal Event Recorder

The event recorder has to fulfil two tasks: assigning globally valid time stamps to the incoming event descriptions, thereby building event records, and supplying a first level of high-speed buffering.

The interface between event detector or hybrid interface and event recorder is a data path transferring the event description itself, and a control path signalling the occurrence of events. The control path mainly consists of four request lines (Req_i) and four grant lines (Gnt_i), each pair Req_i/Gnt_i servicing an asynchronous and independent event stream. That means, up to four object nodes can be monitored with only one event recorder.

Each of the four event streams can be furnished with an arbitrary fraction of the data field, which in total supplies 48 bits. If at least one of the request lines signals an event, the DPU's capture logic latches the data field into a 48 bit data buffer in order to establish a stable signal condition for further processing. The output of this data buffer together with the flag register (8 bit) and the clock's display register (40 bit) define a 96 bit physical event record. This is written into the FIFO memory within one 100 ns cycle of the globally synchronized clock.

Each event stream is associated with a bit (E_1 to E_4) in the flag register which indicates that its event stream contributed to a valid event. This mechanism allows to recognize the relevant part(s) of the data field and ignore the rest of it. Coincidence of

www.manaraa.com

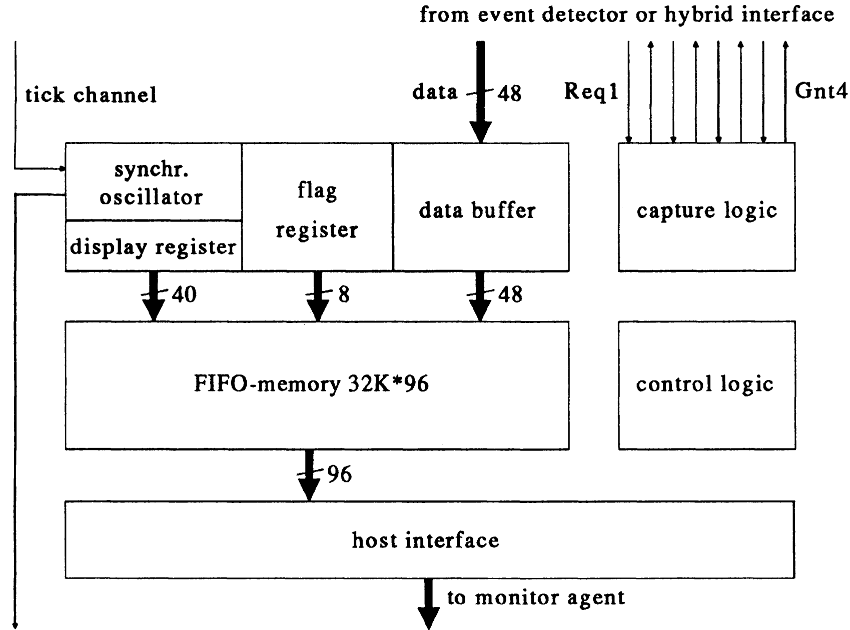


Fig. 3: Universal Event Recorder

events in different streams is possible. Then more than one bit in the flag register is set, meaning that their corresponding parts in the data field are valid event descriptions. There is an additional bit E_s which indicates that a fifth event stream — internal to the monitor system — has generated a synchronization event from decoding the information transmitted via the tick channel. The transmitted synchronization information supports a sophisticated fault-tolerant protocol which allows to prove the correctness of all time stamps at the synchronization events and confirms a clock skew of less than 5 ns (cf. [4]).

Providing a bandwidth of 120 Megabytes/s at the input of the FIFO memory, the event recorder has a peak performance of 10 million events/s. The high-speed buffering having a depth of 32 K event records helps to partly overcome the restricted bandwidth (10,000 events/s) of the monitor agent's local disc: for a short time bursts of events can be recorded and buffered in the FIFO even if the mean event rate of the disc will be exceeded by orders of magnitude. In case of a buffer overflow, a flag is set in the following event record. A second advantage of the FIFO buffer architecture is the ability to read the FIFO buffer while monitoring is going on. This enables continuous monitoring, i.e. there is no restricted maximal length of a trace. So, a high input event rate and arbitrary trace length add to the universality of this event recorder.

www.manaraa.com

4 Interfaces

Within the ZM4 concept, an interface has the task to electrically and mechanically adapt the object system in order to record the events occurring in the object system. As the object systems differ in their behavior and the questions to be answered by monitoring vary, a wide variety of interfaces has evolved. These interfaces can be devided into three classes, starting from basic interfaces for already existing parallel output ports of computers to interfaces for direct adaptation of microprocessor buses, and ending with intricate special purpose interfaces. All of them can arbitrarily be combined when carrying out a measurement. As the following subsections show, building an interface between the ZM4's infra structure (i.e. all parts of the ZM4 except the interface) has the same result as building a powerful new monitor for this system from scratch at only a small fraction of the effort and cost.

4.1 Interfaces for parallel computer ports

In the case the object system itself already has a parallel output port, e.g. for a parallel printer, this port can be used by the instrumentation (i.e. the pieces of software responsible for recognizing events and informing the monitor about it) for putting out the event description when a specified event occurs. The task for the interface designer merely is to correctly connect the output port's signals (normally these are 8 data bits and one strobe bit) to the input connectors of the event recorder.

As such interfaces typically only utilize a small fraction of the event recorder's 48 bit wide data field, obviously up to four object computers can be connected to one event recorder. This is possible due to its ability to deal with four event streams at the same time without any interference between them. Interfaces like these can easily be prototyped and quickly used for real measurements. Currently we have at our disposal the following interfaces:

Interface	streams	width/stream [bit]
DIRMU interface	4	8
Transputer link interface	4	8
Centronics interface (printer)	4	8
MEMSY interface	1	32

This table with simple interfaces already reveals that the ZM4 can easily be adapted to numerous and quite different object systems. Especially the Centronics interface can be used for nearly all PCs and many workstations.

4.2 Interfaces for Microprocessor Buses

Not always 8 bits are sufficient for coding all information associated with an event, and not always a parallel port is available and can be used for monitoring purposes. Typically such parallel ports are missing in multiprocessor systems, e.g. transputer systems, whose I/O-activities normally are carried out by a dedicated host computer. An interface directly connected to the processor pins (or the backplane bus) allows to

www.manaraa.com

adapt the ZM4 to this particular system with moderate effort, too. Such an interface has to be organized in a way, that the object system regards it as a peripheral device, and that it agrees with the simple protocol of the ZM4's event recorder.

Technically spoken, such a microprocessor bus interface is a parallel port to be installed in the object system's hardware, which has to fulfil the same task as an interface already installed also would have to do. The data path width of the parallel interface typically will be fixed at a value which allows to utilize the whole data path width of the event recorder. This allows for allocating 16 bits for the coding of the event itself (i.e. 65536 different events can be distinguished) and 32 bits for event attributes; this meets the requirements of the today's very popular off the shelf microprocessors. If this wide data path is not necessary for a particular monitoring application, it is of course possible to connect more than one interface to an event recorder by simply using an event stream with its associated Req-signal and allocating the necessary number of bits in the data path. A combination of whatever interfaces are needed can easily be done by forking the flat band cables between the event recorder and the interfaces, and combining corresponding parts of the cable to the connectors of the interfaces.

Since the ZM4 project was launched, the following interfaces have been developed and successfully used:

Interface for SMP-Bus: This interface is only 8 bits wide, the natural width of the SMP-bus, because every access of the processor to memory/io transfers one byte of data. It was used in combination with the single signal interface, described in the next subsection.

Interface for SUN4/390: This interface adapts the ZM4 to the backplane bus of the SUN server; it is 16 bits wide and can be accessed by arbitrary processes running on the server in system and user mode in order to put out event data to be recorded by ZM4 or any other monitor device connected to it. In order to allow a user process accessing a hardware resource like this, a dedicated driver has been incorporated into the UNIX operating system.

Interface to Transputer: The Transputer bus interface adapts the ZM4 (more precise: the event recorder) to the 32-bit family of Transputers, i.e. the T414, T425, T800, by directly accessing the signals at the socket of the respective microprocessor. For this purpose a dedicated fixed/flexible printed circuit board was developed, which grabs all relevant signals of the Transputer via an intermediate socket, and connects them to the interface board on the other side. An event is signalled the external monitor via this interface by an assignment to a variable, which is located in a memory area not occupied by existing RAM. The hardware signals associated with this assignment are then recognized by the interface and transformed into an event description and the request signal. This mechanism allows to transfer all 48 bits within a single instruction: 16 bits are transferred within the address and 32 bits as the value assigned, yielding in a very low overhead for signalling an event to the ZM4.
A special feature of this interface is the hardware event filter integrated in it, which allows the inclusion or exclusion of each possible event separately: the 16 bit portion of the event data from the object system is compared with the set of all events to be included, and only if the event matches, a request to the event recorder will be issued. The information which events are relevant or irrelevant is specified by the user of the

www.manaraa.com

monitor; specifying these events is possible by defining ranges of events and in terms of binary patterns. This user supplied information is parsed, transformed and transferred to the interface via the event recorder using a serial protocol.

5 The Interface to MEMSY

The Interface to the multiprocessor system MEMSY, which is described in detail elsewhere in this volume, is the result of a fruitful cooperation of the hardware and software developers on the one side and the monitoring and performance evaluation group on the other side. As was stated earlier in sec. 4.1, the interface to MEMSY forms a 32 bit wide information path dedicated for putting out event descriptions for the ZM4 from the user or system level in MEMSOS, which is an extended and adapted version of the UNIX operating system. This information path consists of a hardware part and a software part, which are described in the remainder of this section.

5.1 Hardware

As MEMSY is a modular system not only at the top level where the computers are interconnected to a pyramid topology, but also on the module level, where (off the shelf) processor boards and (proprietary) interconnection boards are combined to form

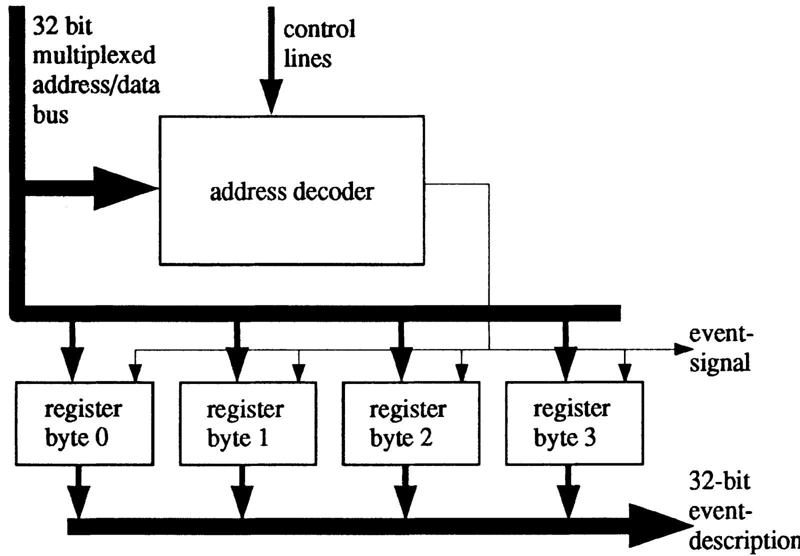


Fig. 4: Hardware Structure of MEMSY interface

a MEMSY module, an elegant solution could be found to integrate the monitor interface into MEMSY. As can be seen from Hofmann et al. [2] in Fig. 3.1 elsewhere in this

www.manaraa.com

volume, the measurement interface is connected to the local bus of the MEMSY nodes, which allows an access time to it as small as the access time to the local high speed memory, yielding a minimal hardware overhead.

In fig. 4 the hardware structure of this interface is shown, with the signals of the local bus entering in the upper left corner. The 32 bit wide multiplexed address/data bus is connected to the address decoder and four 8 bit registers, which store the event description for the external monitor [3]. When an event occurs, an integer value can be written into this register array, which usually is a unique identification of the event occurred. The output of this register, together with the clock signal derived from the address decoding, is fed to a connector on the front panel of the MEMSY-module. By this way, the external part of the interface between this dedicated parallel monitor port of MEMSY and the event recorder of the ZM4 could be reduced to a simple flat band cable with the appropriate connectors on each side.

A timing diagram of this interface is shown in fig. 5. On the upper line the sequence of address data pairs on the multiplexed address/data bus is drawn. Let us assume, that the first A-D-cycle is caused by a monitored program, putting out an event description caused by an event occurring. When this happens, the address decoder recognizes the

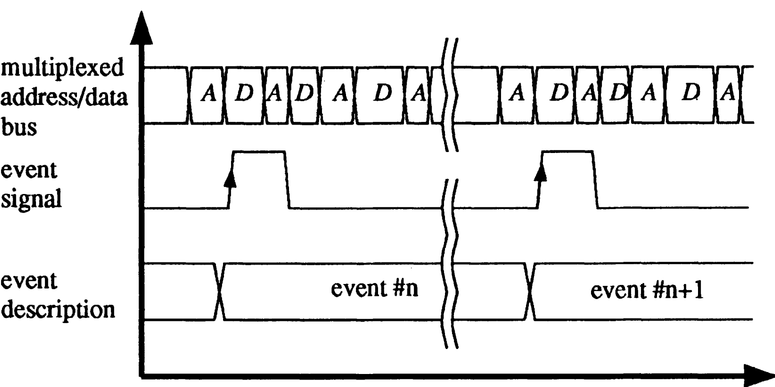


Fig. 5: Hardware Structure of MEMSY interface

corresponding address together with a certain combination of the control lines on the local bus. Immediately after the contents of the multiplexed address/data bus change from the address- to the data part, the data part is copied into the registers and at the same time the occurrence of the event is signalled to the external monitor by means of the positive going edge of the event signal. This marks the time of the event.

5.2 Software

Astonishingly, the greatest challenge in making a computer system accessible to hybrid monitoring is not building the interface hardware, but finding a fast way for putting out the event description from the user level (which normally is of interest) to the hardware

www.manaraa.com

port dedicated to monitoring. The reason for this difficulty lies in the virtuality of modern processor architectures, and the operating system making use of this and the corresponding memory and io protection mechanisms. So, it is normally impossible for a process running on the user level of a UNIX operating system to directly access any hardware resource, except the memory mapped to this process by the hardware built into the processor. While the mapping of virtual memory addresses to physical memory addresses is carried out by the processor hardware very efficiently, this is not true for accesses to any other hardware resources. In order to do that, the operating system has to be requested which actually carries out the requested access for the user process.

This imposes a considerable overhead in the very simple transactions necessary for putting out the small amount of data for an event description: for handling the event description itself approximately 1 to 5 assembler instructions are needed, but using the standard way for accessing hardware resources in UNIX and many other operating systems usually accounts to typically about 200 to 500 assembler instructions, as we investigated while developing the interface for SUN4/390 [12], the SUPRENUM interface [10] and the Centronics interface applied to software running under the SCO-Xenix operating system [5].

The standard way for accessing hardware resources as discussed before is to write a device driver and link it to the operating system kernel. This allows for a standardized interface between software- and hardware-resources. The effect of such an interface is 1) a change from user level to the operating system level, resulting in saving all registers of the user process and restoring them from the system values 2) a table lookup in order to find the driver code, 3) a call to the driver routine, and 4) a change back to the user level. The most time consuming part of this procedure is the twofold change in the protection level which is carried out by the operating system itself, and serves to completely isolate the system state from the user state, and at the same time to isolate all user processes from each other.

Although this mechanism provides nearly perfect protection of the operating system and user processes from erroneous behavior of a particular process, a lot of work is done not necessary for simply putting out one or two machine words to a parallel port. At this point the operating system crew within the MEMSY project incorporated an optimized system call for monitoring into MEMSOS [11], which only needs about 50 assembler instructions, and the time needed for putting out a 32 bit wide event descriptions amounts to about 4 micro seconds. This reduction in comparison with the standard way is due to restricting the storage and restorage of registers to the absolute minimum necessary for carrying out that particular task.

To the programmer or user, who wants to make use of this new capability, the system call can be accessed by a set of procedure calls in the standard library just in the same way as any other procedures can be dealt with. Before event descriptions can be put out, the interface software has to be initialized with the parameterless procedure `mmessopen()`, which checks if the interface is already in use by an other application. After successfully returning from `mmessopen`, event descriptions can be put out with `mmessout` *(event description)* where *event description* can be an arbitrary number with type `long int`. Every call to `mmessout` triggers the software and hardware activities as described before. After finishing the measurement activities,

www.manaraa.com

the measurement interface should be freed for other processes with the procedure `mmessclose()`. This releases the interface, and all subsequent calls to `mmessout` will be ignored.

6 Summary

In this paper the concept of event-driven monitoring was introduced as an aid for abstracting the complex dynamic behaviour of a parallel or distributed computer system to a wisely selected number of events of interest. This kind of abstraction has two main advantages: 1) the amount of monitored data is dramatically reduced, and the data monitored is interesting in terms of the level of abstraction defined. 2) The same concept of events is used in the field of specification and modeling of parallel and distributed systems. So event-driven monitoring can be used to augment the pure functional models resulting from algebraic specification with realistic performance parameters.

Additionally the monitored event traces can be analyzed in terms of predecessor/successor relations, allowing to derive causal relationships from these relations, if the definition of events was done properly. And knowing the causal relationships between the events occurred is the basis for improving the system's behaviour in terms of correctness and performance.

In order to augment event-driven monitoring in arbitrary parallel and distributed systems the universal distributed hardware monitor ZM4 was developed and built. Its main features are 1) the distributed master/slave-archtecture, 2) its precise global clock, which allows to order all relevant events in the correct temporal order, and 3) its universal adaptability, which was shown in the description of the interfaces currently available.

Finally the interface to MEMSY was described as the outcome of a fruitful cooperation between different groups within the MEMSY project. The very low overhead of the software driver for hybrid monitoring and the small effort necessary for the hardware part at the same time would not have been possible for us, if we did not have had access to both the computer hardware and the source code of the software.

References

[1] A. Duda, G. Harrus, Y. Haddad, and G. Bernard. Estimating Global Time in Distributed Systems. In *Distributed Systems, Proceedings of 7th Int. Conf.*, Berlin, September 1987.

[2] F. Hofmann et al. MEMSY, A Modular Expandable Multiprocessor System. in this volume.

[3] A. Grygier. Personal communication. March 1992.

[4] R. Hofmann. *Gesicherte Zeitbezüge für die Leistungsanalyse in parallelen und verteilten Systemen.* PhD thesis, Universität Erlangen-Nürnberg, 1993.

[5] R. Hofmann, R. Langer, and R. Speyerer. Ereignisgesteuertes Hybridmonitoring in einer UNIX-Umgebung. Technical Report 9/89, Universität Erlangen-Nürnberg, IMMD VII, Dezember 1989.

[6] R. Klar. Das aktuelle Schlagwort — Hardware/Software–Monitoring. *Informatik–Spektrum*, 1985(8):37–40, 1985.

www.manaraa.com

[7] R. Klar and N. Luttenberger. VLSI–based Monitoring of the Inter–Process–Communication of Multi–Microcomputer Systems with Shared Memory. In *Proceedings EUROMICRO '86, Microprocessing and Microprogramming, vol. 18, no. 1-5*, pages 195–204, Venice, Italy, December 1986.

[8] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[9] B. Mohr. *Ereignisbasierte Rechneranalysesysteme zur Bewertung paralleler und verteilter Systeme*. PhD thesis, Universität Erlangen–Nürnberg, 1992. VDI Verlag, Fortschritt-Berichte, Reihe 10, Nr. 221.

[10] M. Siegle, R. Hofmann, and K.-H. Werner. Messungen an SUPRENUM. *Informatik–Spektrum*, 14(6):356, Dezember 1991.

[11] W. Stukenbrock. Personal communication. April 1992.

[12] N. Wang. An Experimental Environment for a Performance Study of X Window Systems. Technical Report 1/92, Universität Erlangen–Nürnberg, IMMD VII, Januar 1992.

www.manaraa.com

Graph Models for Performance Evaluation of Parallel Programs

Franz Hartleb

University of Erlangen–Nürnberg
Martensstraße 3, D–8520 Erlangen, Germany

Abstract. For parallelizing an algorithm and for mapping a given program onto a parallel or distributed system there are generally many possibilities. Performance models can help to predict which implementation and which mapping is the best for a given algorithm and for a given computer configuration. Stochastic graph modeling is an appropriate method, since the execution order of tasks, their runtime distribution, and branching probabilities are represented.

In this paper a survey of the modeling possibilities and the analysis techniques implemented in our tool PEPP is presented. The analysis techniques include a new approximation method and powerful bounding methods for the mean runtime.

1 Introduction

In order to minimize the time for solving given problems the execution time of different algorithms and implementation strategies have to be evaluated. As implementing these different versions is too costly or even impossible if the desired hardware is not available, predicting the runtime with a model is an appropriate method.

As the runtime of a user program normally depends on the input values and on the mostly unknown duration of library and system functions it is not easy to predict it, even if a program is running on a monoprocessor system. In parallel systems where processes communicate with each other and share resources there are additional factors that have an influence on the runtime, e.g. race conditions, mutual waiting of processes, or access conflicts on interconnection networks. Therefore, modeling the behavior of parallel programs in order to predict their runtime can be very complicated.

Graph models have been used extensively in order to model and to analyze the behavior of parallel programs [12, 11]. Experience showed that realistic models often consist of hundreds of nodes. In this case the exact evaluation methods fail because of too high computation times. Two ways of overcoming this are approximate methods or methods to obtain lower and upper bounds of the mean runtime which need much less computation time.

If one knows the runtimes of the essential parts of the parallel program it is possible to use this knowledge for predicting the performance of not yet implemented

www.manaraa.com

alternatives. This is done for analyzing open problems, such as different synchronization methods or interconnection networks, based on numerical distribution functions. Integrating such measurement results and models for performance prediction allows us to increase the accuracy of the predicted runtime.

Additionally, the analysis based on numerical functions instead of parametric functions has two essential advantages. First it is much more efficient and second modeling the tasks' runtime is not restricted to a particular type of distribution functions like exponential or phase type distribution functions.

This paper is part of an effort to develop a set of tools PEPP (Performance Evaluation of Parallel Programs) for evaluating the performance of parallel programs. All tools are based on graph models. PEPP is used for automatic instrumentation of C-programs [4] and to predict the runtime distribution, the mean runtime, or bounds for the mean runtime of parallel programs. In section 2 we define the stochastic graph model, the node types, and the runtime distributions used in PEPP. A survey of the analysis techniques is given in section 3.

2 Stochastic Graph Models

A parallel program is modeled by an acyclic, directed, stochastic graph $G = (V, E, \underline{T})$ which consists of a set of nodes V, a set of directed edges $E \subset V \times V$ (arcs), and a vector of random variables $\underline{T}$. Each task or module t_i of the parallel program is modeled by one node v_i and the corresponding random variable T_i which describes the runtime behavior of task t_i. The tasks' runtimes are assumed to be independent random variables. The dependency between tasks is modeled by arcs. An arc from node v_i to node v_j means that task t_j can start execution only if the execution of task t_i is finished. In the following section, we also use the set of predecessor nodes $p(v_i)$ and the set of successor nodes $s(v_i)$ of node v_i. If $p(v_i) = \emptyset$, then v_i is called a start node. v_i is called an end node, if $s(v_i) = \emptyset$. The level $l(v_i)$ of node v_i is defined as the longest path from a start node to node v_i:

$$l(v_i) = \begin{cases} 0 & \text{if } p(v_i) = \emptyset \\ \max\limits_{v_j \in p(v_i)} l(v_j) + 1 & \text{else} \end{cases} \tag{1}$$

2.1 Node Types in PEPP

In order to make modeling and model evaluation more efficient in PEPP, we define four node types:

- An *elementary node* represents one task t_i of the parallel program with service time distribution $F_i(t)$.
- The execution of n identical tasks on n processors in parallel is modeled by a *parallel node*. All the n tasks are assumed to have the same runtime distribution $F_i(t)$.
- Iterations and loops are modeled by *cyclic nodes* or *hierarchical loop nodes*. A cyclic node is characterized by the runtime distribution of the body of the

www.manaraa.com

iteration or loop and the probabilities p_k which are the probabilities that the body is executed again after the k-th iteration. The body of a hierarchical loop node may be described by any PEPP graph. This allows us to model cycles and the graph remains acyclic.

- A *hierarchical node* contains an arbitrary graph model consisting of elementary, parallel, cyclic, and hierarchical nodes. Any graph model can be included in a higher order graph by this node type.

2.2 Runtime Distributions

For the tasks' runtime distribution $F(t)$ many parametric distributions and numerical distributions can be used in PEPP. The runtime distribution may be

- deterministic $F(t) = \begin{cases} 0 & \text{if } t < d \\ 1 & \text{else} \end{cases}$
- exponential $F(t) = 1 - e^{-\lambda t}$
- Branching Erlang $F(t) = 1 - \sum_{n=1}^{N} p_n e^{-\lambda_n t} \sum_{j=0}^{k_n - 1} \frac{(\lambda_n t)^j}{j!}$
- one deterministic and one exponential phase $F(t) = 1 - e^{-\lambda(t-d)}$
- numerically given

The deterministic, the exponential, and the Branching Erlang distribution are well known and often used in performance models. In many cases the approximation of the real runtime behavior with deterministic or exponential distributions is not appropriate and the approximation with Branching Erlang distributions often leads to a very high number of phases which makes the model intractable.

In order to avoid the problems of inappropriateness and intractability we developed two approaches for describing runtime distributions.

Approximating the runtime with one deterministically and one exponentially distributed phase allows us to adapt the first and the second moment of the runtime and we get only two phases for one node.

In the second approach we describe any parametric or measured distribution function by its numerical representation (figure 1). A numerical density function of an arbitrary density function $f(t)$ is defined as a tupel $(D, N, f_0, f_1, \ldots, f_I)$ [5] with displacement D, order N, and

$$f_0 = \int_0^{D+2^{N-1}} f(t)\,dt; \qquad f_i = \int_{D+(2i-1)2^{N-1}}^{D+(2i+1)2^{N-1}} f(t)\,dt; \qquad f_I = \int_{D+(2I-1)2^{N-1}}^{\infty} f(t)\,dt \tag{2}$$

The width of the discrete steps is defined by the unit b and order N. To make the order of different densities adaptable, we always choose $D \bmod 2^N = 0$. This is important for model analysis. The numerical density is called ϵ–exact, if

$$\int_0^{D-2^{N-1}} f(t)\,dt \leq \epsilon \qquad \wedge \qquad \int_{D+(2I+1)2^{N-1}}^{\infty} f(t)\,dt \leq \epsilon \tag{3}$$

This definition keeps the number of intervals small. In our implementation ϵ is 10^{-6}.

المنارة للاستشارات
www.manaraa.com

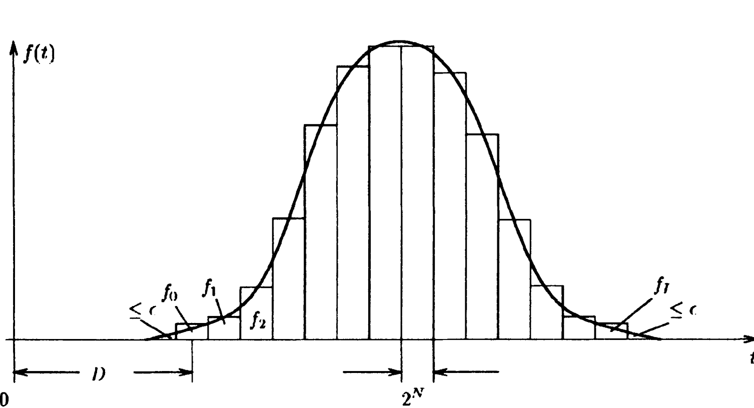


Fig. 1. Continuous density function and the corresponding numerical function

3 Model Analysis

The analysis method depends heavily on the structure of a given graph. In order to classify the models in *series-parallel* and *non-series-parallel* graphs we define two reduction operators:

- *serial reduction*: This operator can be applied on two nodes v_i and v_j, if $s(v_i) = \{v_j\}$ and $p(v_j) = \{v_i\}$. By the serial reduction operator the two nodes v_i and v_j are replaced in the graph model by one node v_{sr} with $p(v_{sr}) = p(v_i)$, $s(v_{sr}) = s(v_j)$, and

$$F_{sr}(t) = \int_0^t F_i(\tau) f_j(t-\tau) d\tau \tag{4}$$

- *parallel reduction*: Two nodes v_i and v_j can be reduced to one node v_{pr}, if $p(v_i) = p(v_j)$ and $s(v_i) = s(v_j)$. The two nodes are replaced by one node v_{pr} with $p(v_{pr}) = p(v_i)$, $s(v_{pr}) = s(v_i)$, and $F_{pr}(t) = F_i(t)F_j(t)$.

A graph G is called a series-parallel graph if G can be reduced to one node by the operators serial reduction and parallel reduction. All other graphs are called non-series-parallel graphs.

Obviously, if we have a series-parallel graph, we are able to obtain the runtime distribution of the modeled parallel program by reducing the graph to one node v_{sp}. The random variable T_{sp} corresponding to v_{sp} describes the runtime behavior, and the mean runtime as well as higher moments can be obtained. This technique was used in [7, 9, 8] and [2] to analyze series-parallel graphs where the tasks' runtime distributions are modeled by exponential polynomials. However, applying this method the number of parameters grows so fast that it can be used only for small graphs. We cope with this problem by transforming the parametric functions to numerical functions and applying the reduction operators to the numerical functions [5]. This allows us to evaluate series-parallel graphs of nearly arbitrary size.

www.manaraa.com

The exact analysis of non-series-parallel graphs is very costly, even for the small graphs. In [6] a method is introduced to compute the runtime distribution of non-series-parallel structured graphs in which the tasks' runtimes are described by polynomial functions. If the tasks' runtime distributions are given by General Erlang functions, we can use the well known transient state space method. As the number of states grows exponentially with the number of Erlang phases, this method is not applicable in general. We therefore propose to reduce the number of states by approximating the runtime distribution of each task with one deterministically and one exponentially distributed phase. The mean runtime of a graph consisting of deterministically and exponentially distributed phases can be obtained with the approximate state space method. This analysis is done in three steps:

- The runtime distribution of each node is approximated, depending on the variance of the node, by one deterministically and one exponentially distributed phase or by a two-phase hyperexponential distribution.
- Serial reduction by adding the expected runtime and the runtime variance of serial connected nodes.
- The remaining graph, consisting of deterministically and/or exponentially distributed phases is analyzed by an approximate state space method described in [10].

Applying this method, the computation costs for analyzing graph models can be reduced by a factor of 10^2 and more if the tasks' runtime variance is large. Nevertheless, when modeling programs with a degree of parallelism larger than 10, this method may also fail due to the number of states in the state space.

An approximate method for analyzing regularly structured non-series-parallel graphs is described in [2].

If all these methods fail we apply methods for bounding the expected runtime. Using only the first moment and the variance of the tasks' runtime, bounds for the mean runtime of the whole program are obtained in [1]. Lower and upper bounds are also obtained in [13] where the nodes' runtimes are assumed to be independent and identically distributed of type NBUE. The mean service time of the critical path is used as a lower bound. This does not yield good results for high parallelism because only the first moments of the runtime distributions and no parallelism are considered. The upper bound is obtained by replacing the tasks' runtime distribution functions by exponential distribution functions with the same first moment and building the product of the distributions of all paths from a start node to an end node. Again only the first moments are considered and therefore the bounds are poor in general.

We use the whole information contained in the tasks' runtime distribution to obtain lower and upper bounds for the mean runtime of parallel programs. The three methods implemented in PEPP are based on the same principle. To get an upper bound of the mean runtime we add nodes or arcs to a given non-series-parallel graph in order to make it series-parallel reducible. The mean runtime of the remaining graph is an upper bound. Removing nodes or arcs from the original graph leads to a lower bound of the mean runtime. A detailed description of the different methods and a comparison is given in [3].

www.manaraa.com

Both, graph construction and graph analysis can be done hierarchically. Choosing a hierarchical evaluation method means that the subgraphs of level $i-1$ are analyzed and the distribution functions or the mean runtimes are inserted into the graph of level i. Applying the approximate state space method each subgraph is replaced by one deterministically distributed phase and we obtain a lower bound for the mean runtime. With the series-parallel-reduction method each subgraph is replaced by a lower or an upper bound inserting the best bounds which are obtained by the different methods. Therefore, we use the advantages of the different methods to get the best bounds for the mean runtime. If a given graph already has a series-parallel structure, the bounding methods always lead to exact results.

4 Conclusions and Prospects

This paper gives a survey of modeling possibilities and analysis techniques implemented in the analysis tool PEPP. The approximate state space analysis allows us to obtain very accurate results for the mean runtime of graphs with a non-series-parallel structure. The series-parallel reduction based on numerical distribution functions leads to the runtime distribution of the modeled program in a very efficient way. Applying this method to non-series-parallel graphs, we get lower and upper bounds for the mean runtime and the runtime variance.

For further research it would be of interest to combine the advantages of the different bounding strategies in one graph level, insertion of arcs and insertion of nodes, in order to get better bounds in less computation time.

Acknowledgements

The author would like to thank Dr. R. Klar for reading an earlier draft of this paper and for many valuable discussions.

References

1. L.P. Devroye. Inequalities for the Completion Times of Stochastic PERT Networks. *Math. Oper. Res.*, 4:441–447, 1980.
2. G. Fleischmann. *Performance Evaluation of Parallel Programs for MIMD-Architectures: Modeling and Analyses (in German)*. Dissertation, Universität Erlangen–Nürnberg, 1990.
3. F. Hartleb and V. Mertsiotakis. Bounds for the Mean Runtime of Parallel Programs. In R. Pooley and J. Hillston, editors, *Sixth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 197–210, Edinburgh, 1992.
4. R. Klar, A. Quick, and F. Sötz. Tools for a Model–driven Instrumentation for Monitoring. In G. Balbo, editor, *Proceedings of the 5th International Conference*

www.manaraa.com

on Modelling Techniques and Tools for Computer Performance Evaluation, pages 165–180. Elsevier Science Publisher B.V., 1992.

5. W. Kleinöder. *Stochastic Analysis of Parallel Programs for Hierarchical Multiprocessor Systems (in German)*. Dissertation, Universität Erlangen–Nürnberg, 1982.
6. J.J. Martin. Distribution of the Time through a Directed, Acyclic Network. *Operations Research*, 13(1):46–66, 1965.
7. R. Sahner. *A Hybrid, Combinatorial Method of Solving Performance and Reliability Models*. PhD thesis, Dep. Comput. Sci., Duke Univ., 1986.
8. R. Sahner and K. Trivedi. Performance Analysis and Reliability Analysis Using Directed Acyclic Graphs. *IEEE Transactions on Software Engineering*, SE-13(10), October 1987.
9. R. Sahner and K.S. Trivedi. SPADE: A Tool for Performance and Reliability Evaluation. In N. Abu El Ata, editor, *Modelling Techniques and Tools for Performance Analysis '85*, pages 147–163. Elsevier Science Publishers B.V. (North Holland), 1986.
10. F. Sötz. A Method for Performance Prediction of Parallel Programs. In H. Burkhart, editor, *CONPAR 90–VAPP IV, Joint International Conference on Vector and Parallel Processing. Proceedings*, pages 98–107, Zürich, Switzerland, September 1990. Springer–Verlag, Berlin, LNCS 457.
11. F. Sötz and G. Werner. Load Modeling with Stochastic Graphs for Improving Parallel Programs on Multiprocessors (in German). In *11. ITG/GI–Fachtagung Architektur von Rechensystemen*, München, 1990.
12. A. Thomasian and P.F. Bay. Analytic Queueing Network Models for Parallel Processing of Task Systems. *IEEE Transactions on Computers*, C-35(12):1045–1054, December 1986.
13. N. Yazici-Pekergin and J.-M. Vincent. Stochastic Bounds on Execution Times of Parallel Programs. *IEEE Transactions on Software Engineering*, 17(10):1005–1012, October 1991.

www.manaraa.com

Load Management on Multiprocessor Systems

Thomas Ludwig

Technische Universität München
Institut für Informatik
Arcisstr. 21, Postfach 202420
D–W8000 München 2
phone: +49–89–2105–2042 or –2382
email: ludwig@informatik.tu-muenchen.de

Abstract. This paper concentrates on load management methods for multiprocessor systems with distributed memory. Most of the programs running on these machines do not create new processes during runtime and there is usually no sharing of nodes between multiple users. This causes load movement by process migration to be indispensable. We implemented a testbed to find out which load parameters are significant for deciding on a system rebalance. The testbed also allows the investigation of other important aspects of load management schemes.

1 Introduction

During the last years much research has been devoted to the development of multiprocessor systems. There are not only considerable advances in hardware architecture but also in the design and implementation of software tools. Nevertheless, it is still easier to build a parallel machine than to use it efficiently. Two major types of parallel computers are available now: multiprocessor systems with shared memory and systems with distributed memory. Most of the problems concerning programming and running programs efficiently do not exist for shared memory systems due to the fact that these machines resemble monoprocessor systems with multitasking facilities. This paper completely concentrates on multiprocessors with distributed memory.

Using systems with distributed memory usually makes program development and performance tuning of the programs very difficult. One severe problem is the load imbalance between processing nodes that may appear during runtime. Two approaches can be used to cope with this particular type of problem which can sometimes reduce the performance of a parallel system to that of a single monoprocessor:

- A static approach manages the distribution of processes to the processors before runtime (usually called mapping). In many cases this approach requires a priori knowledge about the behaviour of the program, i.e. execution times of the processes or precedence relations between them.
- If this a priori knowledge is not available, a dynamic approach is applied, where the distribution of processes happens during runtime of the program. This approach is referred to as load sharing or load balancing.

www.manaraa.com

As load sharing and load balancing are used for different types of approaches, we will use the general term load management. From the operating system's point of view, load management is a global resource scheduling problem, in which the usage of the CPU, the communication network and the memory on each node must be taken into consideration.

Many investigations have been undertaken in this field but the main problems are not yet solved. However, there are many interesting simulation results showing concepts for possible implementations.

2 Load Management Issues

For every load management system it is important to determine which performance values are to be optimized. Depending on the specific way of using the machine, several aspects are important: In environments with more than one user or several programs sharing the same set of processing nodes, load management usually improves the mean response time of programs as well as the mean variance of response times. Permanent creation of new processes makes load management schemes easy to implement as its mechanism must only have the capability to move new processes to nodes with low load. Single user/single program environments which are typical for multiprocessor systems add some restrictions to this concept. Load management must be able to reduce the response time of a single program even if this program does not create new processes during runtime. This requires mechanisms for moving running processes between nodes.

Most of the research is oriented towards this first aspect of load management and concentrates mainly on efficiency improvements in distributed systems. Nevertheless, many concepts are also applicable to multiprocessor systems with single user/single program environments. We will therefore first give a short overview of existing approaches and classification criteria for load management schemes (see also [9, 12]).

At first, we should distinguish between load sharing and load balancing. Although there is no common agreement to separate these two terms, we can define load sharing as trying to avoid idle–times on any processor in the system while processes on other nodes are waiting to be served. Load balancing, however, tends to balance the load in the system according to a given criterion (see [7], [10]).

Also, the notions of static load management and dynamic load management are often confused. Following [3] and [5] we call the management static if it redistributes load during runtime of the program without any knowledge of the current system state. Consequently, dynamic management means taking into consideration load parameters of the system and of nodes. In the following we will only investigate dynamic load management schemes.

Management decisions have to be taken by a decision unit which represents the intelligence of the system. Two important parameters have to be specified and tuned:

www.manaraa.com

1. How many decision units are present in the multiprocessor system and how are they mapped onto the processing nodes?
2. How much knowledge about the system state must each unit have to make clever load management decisions?

Many combinations of possible concepts have been studied using numerical analysis and simulations but there seems to be no final answer to these questions.

There are three possible solutions to the placement problem of the decision units: local, clustered and centralized location. Centralized components have the advantage of having knowledge about the whole system. This makes management decisions more efficient and allows quick reactions to changing load characteristics in different parts of the system. On the other hand, having central components is a contradiction to the architectural principles of systems with distributed resources because they usually cause loss of efficiency. With local components the scalability of the computer system would not be influenced by the load management units. However, in this case, the redistribution of load is performed locally so that a global balance can only be reached slowly. Clustered methods try to find a tradeoff between these two principles. Most of the investigations deal with local methods, for example [6], [11], [13], and [15]. There are also some results which show the possible benefits of both approaches under similar conditions (see [16] and [17]).

The problem of the amount of information which is necessary for decision making has two aspects. First, how many load parameters should be measured on each node? Simple methods work only with idle–time or number of processes, whereas more sophisticated methods also use communication measures and parameters of individual processes. Secondly, what overview of the system should the decision unit have, i.e. from how many other nodes should it have load information? Possible answers range from only local knowledge to global knowledge. One would expect that more information should yield better decisions. But Casavant/Kuhl [4] found that a higher amount of information even results in lower efficiency because of the additional time needed to collect all the information. Concerning the choice of the parameters no investigations have been undertaken and the parameters are often chosen randomly. Ferrari/Zhou [7] claim to analyse the influence of varying load parameters carefully and to use mean values of different queue lengths of the operating system. As this is a crucial point for the efficiency of every load management algorithm our work concentrates on the question which parameters are relevant for decision making.

Before we describe our approach in detail we will first take a brief look at mechanisms for moving load from one node to another.

In existing load management methods two types of objects for load movement can be identified: data structures and processes. Migration of data segments is directly integrated into the application program itself as it can not be handled without detailed knowledge about the interaction of data and program. In many cases this mechanism yields very good performance values, but it exhibits a significant lack of portability. Thus, with every new program the mechanism must be implemented again. Only integration of the mechanism into the operation

www.manaraa.com

system will lead to a load management system which is invisible to the user and independent of his style of writing programs.

The common level of abstraction at which object movement appears to be efficient is the process level. Movement of smaller units of an application program, e.g. procedures or even statements, will not result in performance gains as the transfer of these units between nodes needs too much time compared to the execution time of the units. Modern operating systems for parallel computers with distributed memory, for example Mach and Chorus already provide mechanisms for moving processes (or their corresponding constructions like actors etc.) and thus for the integration of load management schemes (an overview of these systems can be found in [8] and [14]).

The following section describes our approach of a load management system which uses migration of running processes as a mechanism for balancing the load.

3 A Load Management Testbed

Nearly all load management methods follow the principle of a control loop (see Fig. 1). The system to be controlled is the parallel computer together with its operating system and the running application program. A measurement unit measures the current state of the system by monitoring the load at its resources (processor, communication network, memory). These values are compared by an evaluation unit which has to determine whether process migration is necessary. If so it sends a command to the migration unit to start load migration.

The following problems are inherent to control loops and must therefore be taken into account by load management methods:

- How can the reaction time of the control loop be optimized?
- How can overreactions be minimized?
- How can oscillating regulations be prevented?

The answer to these questions lies in the intelligence of the decision unit. Only a sound tuning of all parameters that influence the migration of processes will lead to a management method which can handle all crucial load situations.

From the logical structure of the load management process we can directly derive a component structure of an implementation. At least three components must be developed: a load measurement unit, a load evaluation unit and a migration unit. The various options with regard to placement of these units will be discussed later. We will first have a closer look at the components.

3.1 The Load Measurement Unit

The measurement unit is responsible for collecting load information. It receives commands from the load evaluation units, initiates and deactivates measurements and returns the results back to these units. Measures relevant for load

www.manaraa.com

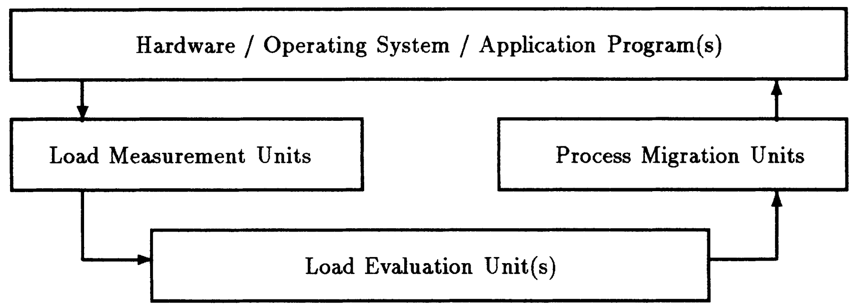


Fig. 1. Control loop of the load balancer testbed

management are the usage of each individual resource: CPU, communication network, and memory. A list of load parameters that can be measured by this unit is given in Table 1.

The measurement unit is integrated into the operating system kernel and acts as a monitoring system (see [2]). In our testbed the measurement unit is implemented in software but hardware solutions are also possible. The influence of the additional component to the running program is rather small: most of our software based measurements do not slow down the application by more than 1%. A reason for this can be found in the methodology of gathering runtime information without strongly influencing the measured object. All data is collected via counters that are activated by hooks integrated into the operating system.

In a distributed system there must be one load measurement unit on each node which is subject to the process of load management. In our testbed this unit is implemented as a kernel process integrated into the operating system.

Table 1. Load parameters of the measurement unit

Node measures	**Process measures**
Processor idle time	Time using CPU
Process ready queue length	Time in process ready queue
Sending/receiving queue length	Time in sending/receiving queue
Free memory	Used memory
Amount of data sent/received	Amount of data sent/received
Number of messages sent/received	Number of messages sent/received
	Number of page faults

www.manaraa.com

3.2 The Load Evaluation Unit

The load evaluation unit is the "brain" of the load management system. It receives performance values from the load measurement units, investigate the necessity of load migration and controls the migration unit. Its complexity is determined by the fact that it should be able to dynamically make appropriate decisions based on performance values of the past.

Three decisions have to be made before a migration can take place in the system:

1. Is there any load imbalance in the system, i.e. are there nodes with load varying significantly from the nodes' mean load?
2. If so, which are the nodes that have high and low load respectively?
3. Which processes are the best candidates to be moved from nodes with high load to nodes with low load?

Each decision is based on at least one load measure that is suitable for this purpose. Different load measures have to be considered before a process migration can be initiated. The evaluation of the existing load imbalance requires parameters that reflect the load situations on nodes, i.e. number of processes, idle–time on these nodes etc. Also, the decision unit must calculate some mean values describing the system as a whole. In order to find nodes with extreme load conditions we can use the same load parameters together with some information about the communication load on the nodes. Finally, the last decision requires process oriented data, e.g. like CPU–usage of individual processes, communication with other processes in the system, and memory usage.

The crucial problem for all load management methods is the question which parameters should be used and how they should be combined to reflect something like load of a node or load of a process. In the literature we find approaches that use only trivial parameters like the number of processes for all decisions and also methods that incorporate many parameters [15]. The approach of Stankovic/Sidhu uses a McCulloch Pitts neuron to calculate a single value (from few up to a dozen and more load parameters) that can be compared with some threshold to make binary decisions. Unfortunately, they did not investigate which parameters are indispensable and which are useless. As this issue is of utmost importance for the efficiency of load management our investigations will concentrate on finding those load measures that are relevant for decision making.

In order to be able to compare decentralized and centralized methods of load management the load evaluation unit is implemented as a user process. The number and localization of the load evaluation unit(s) can be defined via specification files. They are loaded and started before the application processes and can easily be deleted if the user desires another configuration.

3.3 The Migration Unit

The migration unit is activated by the evaluation unit and moves processes from source to destination nodes. In our particular environment processes are not

www.manaraa.com

migrated as a whole but as a set of single pages [1].

With this mechanism a migration candidate is at first stopped at the source node. Its process context and current code page are transferred to the destination node. There the process can directly be handed over to the scheduling mechanism. Once actived again, every page fault causes a transfer operation from pages of its address space. The locations of these pages are known to the kernel. If repeated migrations of the same process take place its pages may be distributed over several former source nodes. This scheme distributes the time necessary for migration over a longer period and thus avoids temporarily overloading the communication network. The approach has several advantages over the conventional approach where processes are migrated as one large block of code:

- The idle time of the migration candidate is minimal. This is important for not to disturbing the synchronization with other processes longer than absolutely necessary.
- The consistency problem is alleviated. As process migrations last only for a short period of time, the number of asynchronous messages arriving during this time interval is small. Thus bringing the system to a consistent state after migration does not produce excessive delays.
- In most cases only a part of the pages used by the process will have to be moved. Moving them only on demand therefore reduces communication.
- If a process is migrated more than once, it can collect its pages from all former nodes. Again, this leads to a gain of efficiency compared to conventional migration.

There is also a disadvantage of this method: the total amount of communication necessary to migrate a process in parts may be higher than for a method that migrates the process as a whole. This is caused by the startup time for every page transfer communication. Only empirical studys will show whether the advantages outweigh this overhead.

The migration component is integrated into the kernel of the operating system as it makes intensive use of datastructures describing processes and also needs the highest privilege level to manipulate the processes.

3.4 The Hardware Platform for the Testbed

The load management testbed has been implemented on an Intel iPSC/2 hypercube multiprocessor system. Our system currently has 32 nodes with 4 MByte main memory each. Its 80386 processor together with the communication network supports paging via communication links. The network provides virtually full connectivity, thus, the distance between sender and receiver does not influence the time necessary for communication and process migration.

As already mentioned, the load measurement component and the process migration component have been integrated into Intel's operating system kernel NX/2 whereas the load evaluation has been implemented as user processes to

www.manaraa.com

facilitate experimentation. Communication between components is performed via ordinary send/receive–calls.

3.5 Issues to be Studied with the Testbed

The possibility to configure the testbed allows the comparison of different alternatives concerning two important issues of load management: where should the components be located in the distributed system and which combination of load parameters is necessary to make good decisions?

We use a definition file for experimentation with the locations of the components. As measurement and migration components have to be located on each node in the system we can only specify the location of the load evaluation units and the number of nodes each load evaluation is responsible for. With a centralized approach we have only one load evaluation unit in the whole system, whereas with decentralized load management there may be an evaluation unit on each node. The issue is to find a tradeoff between gaining a good system overview and thus the ability to react quickly to any changes in load distribution and avoiding centralized components in a distributed computing environment.

The impact of load measures on the migration decisions is of crucial importance for the efficiency of load management. The parameters, taken into account, their relative weighting, and the method of combining them into single load information values can be specified in a description file. This allows not only a rapid comparison of different types of evaluation by the user but also by the machine itself. Test programs can be run automatically with varying sets of parameters to find combinations that yield optimal results

4 Results from Experimentation

It is obvious that the investigation of load management issues is not trivial due to the high number of parameters that influence the performance of the load management system. Parameters can be subdivided into two classes. The first class covers application dependent parameters, e.g. number and size of running processes and their interprocess communication structure. The second class characterizes the behaviour of the load management system, e.g. its physical distribution and decision making heuristics. Any sound analysis must cover a representative set of parameters to ensure that the load management is suitable for a given class of application programs.

At first we have to define the performance criteria of the load management system. With our multiprocessor system (as with most of the currently available multiprocessor systems) only space sharing and no time sharing mode is supported for the users. Consequently, the task of the load management system is to minimize the runtime of the individual programs. No multi user aspects like fairness between users or maximal throughput have to be taken into consideration. This makes performace evaluation easier because we only have to compare program runtimes with and without load management.

www.manaraa.com

4.1 Configuration of the Testbed

In our first set of experiments we tested only a small configuration for the load management system. We chose a configuration of four nodes with only one load evaluation unit managing all of them. This centralized approach can also be a starting point for a clustered configuration, where we have several load evaluation units each managing a small number of nodes.

For the heuristics for decision making we compared four different types. Type 1 bases its decisions only on the measurement of the idle times of the nodes, whereas type 2 takes only the lengths of the ready queues into consideration. Type 3 is a combination of both type 1 and type 2. Finally, type 4 also evaluates the lengths of the receive message queues. The results of type 4 will not be discussed in this paper because it did not improve the performance of types 1-3.

For each type we tested three different threshold values to trigger the process migration. The proper selection of threshold values is important for the performance of the system. Thresholds chosen too high keep the system too long in an unbalanced state, whereas low values lead to instabilities and increasing numbers of process migrations even for almost well balanced systems.

The selection of an appropriate migration candidate was not parametrized in our investigations. We chose a heuristic that takes the CPU-time of the processes and their time in the ready queue for decision making.

4.2 Selection of Test Programs

The load management system was not tested with real programs because their load imbalance can hardly be controlled. Instead we took one application program (calculation of Mandelbrot sets) and modified it in order to meet our requirements.

At first we want to distinguish two types of applications: process systems with and without communication. With the second type, where processes run independently, load imbalance occurs only in the final phase of the program execution. This is due to the fact, that some nodes did already finish their work, i.e. their processes, while others are still active. As long as there are at least as many processes as nodes in the system the load can be redistributed by the load management system. This is of course the easiest situation for our system. For the first type we assume that the processes perform a global synchronization from time to time. Load imbalance appears before each synchronization on those nodes where all processes already reached their synchronization operation and are now in a waiting state. This situation is typical for many real applications, especially for those with iteration algorithms. Of course this behaviour makes it more difficult for the load management system to yield good performance. In order to cover a representative set of programs we varied the synchronization frequencies from one every ten seconds up to one every half a second.

Our test applications were composed of 12 and 16 processes respectively. Each of them calculates one part of a specified Mandelbrot set. Because of the

www.manaraa.com

special nature of the solution algorithm the runtime of each process can not be determined in advance and all process runtimes are different. This fact can be used to simulate different degrees of load imbalance and to evaluate the corresponding performance of the load management system.

The simplest program version has independent processes. Synchronized versions of this application where generated by repeated synchronous calculations of smaller Mandelbrot sets or by a global synchronization after the calculation of each line of the processes' subsets.

We started with an initially equal distribution of the processes, where each node receives 3 or 4 processes. Without load management we determine the optimal mapping (yielding the shortest runtime) and the worst mapping (yielding the longest runtime). Note, that there are even worse mappings when the initial distribution may also be unequal. In addition we select seven more mappings with different runtimes in order to have a suitable distribution of test cases. These nine mappings resemble programs with load imbalances varying from almost zero up to a degree that nearly doubles the runtime of the program.

For each mapping we measure the program runtime with activated load management and compare it to the runtime without load management. The combination of all load management system configurations and all different program types leads to a set of almost 3.000 experiments. In this paper we only discuss a small part of the results. However, they are significant for the results obtained by a statistical analysis of the complete set of experiments.

4.3 Discussion of Results

Figure 2 shows part of the results in a very condensed form. The first plot shows results from a program system consisting of 12 processes. It calculates a Mandelbrot set of 400×400 points. The second plot represents a synchronous program version where 12 processes calculate the same Mandelbrot set (of size 100×100 points) ten times and perform a global synchronization after each calculation.

At first, let us have a look at the meaning of the various lines and symbols in the plots. Both plots show the runtimes in seconds for the nine different mappings. The solid horizontal lines represent the individual runtimes of the programs without load management system. In plot a) they vary from 90 seconds to 168 seconds, in plot b) from 84 seconds to 148 seconds. The runtimes of the programs with activated load management system are given by the centers of the triangles and rectangles. Each mapping has nine runtimes with load management corresponding to the different configurations of the system. Threshold values are indicated by a graphical symbol: normal triangles represent low threshold values, rectangles medium thresholds, and reverse triangles high thresholds. Decision heuristics are represented by the position of the graphical symbols in the plot: the first triple of each mapping (normal triangle, rectangle, reverse triangle) is produced by a heuristic of type 1, where only idle time is measured. The second triple corresponds to type 3 combining idle time and ready queue length. Finally, the last triple reflects type 2, where only ready queue lengths are taken

www.manaraa.com

into consideration. Dashed lines show for each mapping the mean value of the runtimes with activated load management system.

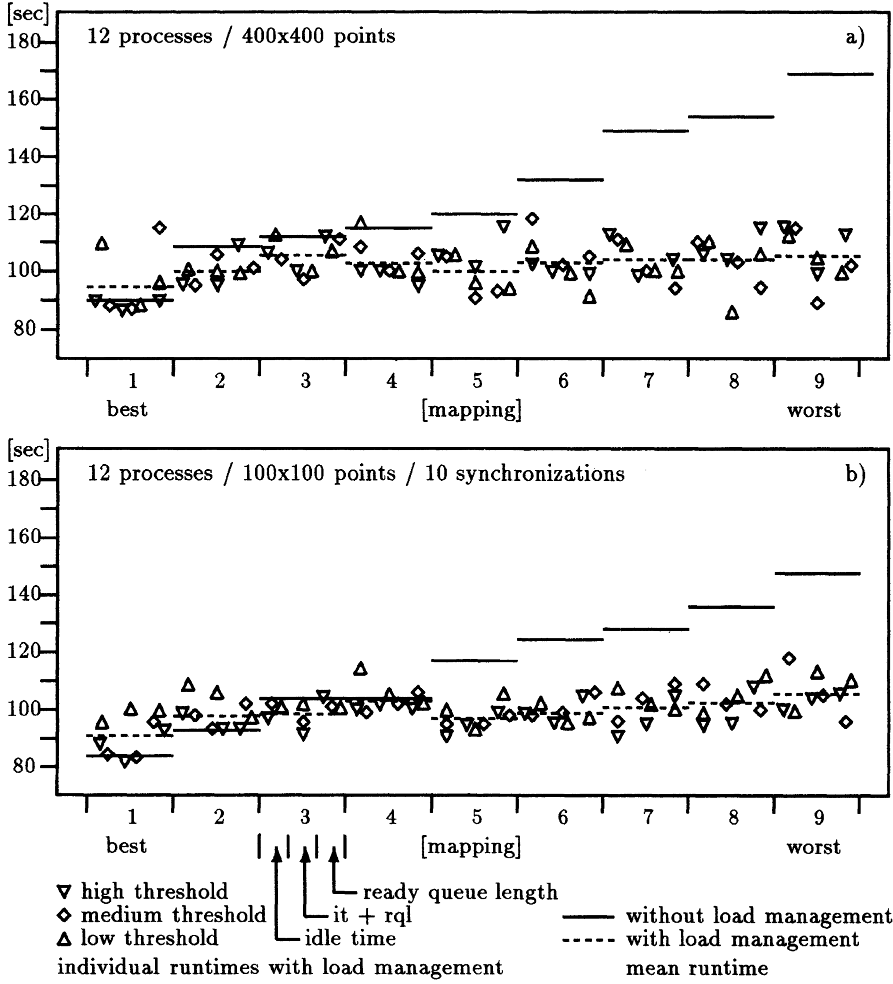


Fig. 2. Runtimes of test programs (with and without global synchronizations) without load management and with activated load management

Let us now discuss the conclusions that can be drawn from the results of these experiments. Several aspects have to be considered as we are interested in the overall efficiency of the system as well as in questions of how to tune the various parameters.

www.manaraa.com

First of all and most important we can see that the load management system balances most of the unbalanced load situations. With unsynchronized processes there is almost always a gain of performance when load management is activated. However, synchronized processes are more difficult to balance. As a consequence, the best two mappings have shorter runtimes with deactivated load management.

In the real world optimal mappings can usually not be determined because of the computing complexity or the fact that load imbalances occur during runtime. For this situation our results from Fig. 2 and other measurements indicate that the load management system will improve the runtime of most of the application programs. Nevertheless, there will also be a certain percentage of programs without internal load imbalance that will suffer from a prolongation of completion time.

Both plots show that the mean values of runtime with load management (given by dashed lines) form an almost horizontal line. This line is about 10 % higher than the value of the best mapping. Thus, the load management system is capable of balancing the load and of achieving a degree of quality which is independent of the chosen mapping. The height of this line is of course determined by the inherent overhead of the load management system itself and by the quality of decision making. Note, that the height will be even lower, if we eliminate all disadvantageous threshold/heuristic combinations. Not only the mean runtime is reduced but also the variance of runtimes for all mappings. For real applications this means that program completion times get more predictable and that the importance to determine clever initial mappings diminishes.

Comparing unsynchronized and synchronized process systems we find that efficiency for the latter is worse. This is not surprising because the situation is more complicated as idle times occur frequently and with a smaller extend. From further results we can see that the frequency of synchronization has almost no influence on efficiency. For synchronized process systems the point of intersection between the horizontal line of mean runtimes with activated load management and the curve for unmanaged programs is farther to the right on the x-axis. Thus, the percentage of mappings that can not be improved by load management increases. One of our future investigations will concentrate on how to minimize the time overhead added to application programs in these situations.

An additional problem has been noticed in the two plots. For synchronized process systems the difference between best and worst unbalanced mapping is smaller. As a consequence, the overall efficiency of the load management system is lower because it adds an almost constant overhead to all mappings.

Let us now have a more detailed look at the runtime differences caused by the three load evaluation heuristics. In general (considering all measured results) we can say that idle time as a single load measure does not yield optimal results. Instead we need at least a combination of idle time and ready queue length. Comparing all synchronized and unsynchronized programs we found that idle time is inappropriate if the number of synchronizations is low. Plot a) in Fig. 2 shows, that mean idle time yields the lowest efficiency. This is a surprising result, because with single blocks of idle time at the end of a program one would expect

www.manaraa.com

idle time to be the most adequate measure. However, using ready queue length instead leads to a better balanced system during the complete runtime. Process migrations even appear if there is no idle time at all but only a difference in ready queue length. At first, these migrations produce only additional costs, but finally result in shorter completion times. They are preventive in the sense that they already take place even if no loss of efficiency (i.e. idle times) occured.

Finally, the threshold value influences the efficiency of the load management system. Regarding Fig. 2 we can see that in many cases low thresholds result in longer runtimes. There are two reasons for this. First, lower thresholds produce a higher number of migrations and thus more overhead. If this overhead is not outweighed by a gain of performance completion times will increase. However, this effect alone can not explain the differences between high and low thresholds. The second reason is the instability caused by low thresholds. Migrations are initiated based on minor load differences on the nodes. This leads to a higher percentage of wrong migration decisions as small load fluctuations are often not significant for the long term behaviour of the process system. Considering all measured experiments we must conclude that medium to high thresholds yield optimal runtimes.

There is an additional result which can not be concluded from the two plots. If we increase granularity (i.e. take 16 processes instead of 12) the difference between best and worst mapping decreases because of the better inherent load balance of the process system. As already mentioned, these situations are problematic for the load management system. However the results are similar to that shown in Fig. 2 (only few mappings have better performance without load management). The reason for this can also be found in the granularity. Actually, load imbalances are smaller, making it difficult to manage them. However, this is compensated by a higher number of potential migration candidates. This facilitates an efficient fine tuning of the load situation. Considering all our results we found that granularity has no influence on the performance of the load management system.

5 Conclusion and Future Work

In this paper we investigated load management issues on multiprocessor systems with distributed memory. The performance criteria of the load management system is the minimization of the runtimes of the individual user programs. We implemented a testbed to investigate the question of decision making in the control loop of the load management system. Load migration is implemented via preemptive process migration where the pages of the processes' address spaces are only transferred on demand.

About 3.000 experiments with varying process system types and decision making heuristics were carried out. We found that the load management system is able to significantly improve most of the load situations that may appear in real programs. Simple load measures like idle time are not appropriate for decision making. Instead combinations of idle time and ready queue length should

www.manaraa.com

be used. The influence of communication based load measures is still an open question.

In our future work we will concentrate on comparing different physical configurations of the load management system. Especially scalability and performance matters will be investigated with respect to decentralized approaches with more that one load evaluation unit.

References

1. T. Bemmerl, A. Bode, O. Hansen, T. Ludwig *A Testbed for Dynamic Load Balancing on Distributed Memory Multiprocessors,* Esprit Project 2701 Parallel Universal Message-Passing Architecture, Working Paper 14, Work Package 4.5, Tech. University of Munich, August 1990.
2. T. Bemmerl, R. Lindhof, T. Treml *The Distributed Monitor System of TOPSYS,* Proceedings of the CONPAR 90 – VAPP IV Joint International Conference, Zurich, Switzerland, Sep. 1990, Lecture Notes in Computer Science, Vol. 457, pp. 756–765.
3. F. Bonomi, A. Kumar *Adaptive Optimal Load Balancing in a Heterogeneous Multiserver System with a Central Job Scheduler,* Proceedings of the Eighth International Conference on Distributed Computing Systems, San Jose, California, June 13–17, 1988, Computer Society Press, Washington, D.C., 1988, pp. 500–508
4. T.L. Casavant, J.G. Kuhl *Analysis of Three Dynamic Distributed Load-Balancing Strategies with Varying Global Information Requirements,* Proceedings of the International Conference on Distributed Computing Systems (Computer Society of the IEEE), 1987, pp. 185–192
5. P. Dikshit, S.K. Tripathi, P. Jalote *SAHAYOG: A Test Bed for Evaluating Dynamic Load-Sharing Policies,* Software – Practise and Experience, Vol. 19 (5), May 1989, pp. 411–435
6. D. Ferguson, Y. Yemini, C. Nicolaou *Microeconomic Algorithms for Load Balancing in Distributed Computer Systems,* Proceedings of the Eighth International Conference on Distributed Computing Systems, San Jose, California, June 13–17, 1988, Computer Society Press, Washington, D.C., 1988, pp. 491–499
7. D. Ferrari, S. Zhou *A Load Index for Dynamic Load Balancing,* Proceedings of the Sixth International Conference on Distributed Computing Systems, Cambridge, Massachusetts, May 19–23, 1986, IEEE Computer Society Press, Washington, D.C., 1986, pp. 684–690.
8. A. Goscinski *Distributed Operating Systems – The Logical Design,* Addison-Wesley: Sydney, 1991
9. O. Hansen, T. Ludwig, R. Milner, S. Baker *Load Balancing Strategies,* Esprit Project 2701, Parallel Universal Message-Passing Architecture, Deliverable, Number 4.5.1, Work Package 4.5, Tech. University Munich and RSRE, December 1990.
10. P. Krueger, M. Livny *The Diverse Objectives of Distributed Scheduling Policies,* Proceedings of the Seventh International Conference on Distributed Computing Systems, Berlin, West-Germany, September 21–25, 1987, ed. R. Popescu-Zeletin, G. Le Lann, K.H. (Kane) Kim, Computer Society Press, Washington, D.C., 1987
11. F.C.H. Lin, R.M. Keller *The Gradient Model Load Balancing Method* IEEE Transactions on Software Engineering, Vol.SE-13, Nr.1, Jan 1987
12. T. Ludwig *Automatische Lastverwaltung für Parallelrechner* to appear in: B.I.-Wissenschaftsverlag: Heidelberg, 1993.

www.manaraa.com

13. W. Shu, L.V. Kalé *Dynamic Scheduling of Medium–Grained Processes on Multicomputers*, Internal Report, Department of Computer Science, University of Illinois
14. J.M. Smith *A Survey of Process Migration*, Operating Systems Review, Vol. 22, Nr. 3, July 1988, pp. 28–40.
15. J.A. Stankovic, I.S. Sidhu *An Adaptive Bidding Algorithm For Processes, Clusters and Distributed Groups*, Proceedings of the Fourth International Conference on Distributed Computing Systems, San Francisco, California, May 14–18, 1984, IEEE Computer Society Press, Silver Spring, 1984, pp. 49–59.
16. M. Willebeek–LeMair, A.P. Reeves *Local vs. Global Strategies for Dynamic Load Balancing*, Proceedings of the 1990 International Conference on Parallel Processing, August 13–17, 1990, Vol. I Architecture, B.W. Wah (editor), The Pennsylvania State University Press, University Park, PA, 1990, pp. 569–570.
17. S. Zhou *A Trace–Driven Simulation Study of Dynamic Load Balancing*, IEEE Transactions on Software Engineering, Vol.14, Nr.9, Sep.1988,pp. 1327–1341

www.manaraa.com

Randomized Shared Memory—Concept and Efficiency of a Scalable Shared Memory Scheme

Hermann Hellwagner

Siemens AG · ZFE ST SN 21
P.O.Box 83 09 53, W–8000 Munich 83
E-Mail: hermann@christine.zfe.siemens.de

Abstract. Our work explores the practical relevance of Randomized Shared Memory (RSM), a theoretical concept that has been proven to enable an (asymptotically) optimally efficient implementation of scalable and universal shared memory in a distributed-memory parallel system. RSM (address hashing) pseudo-randomly distributes global memory addresses throughout the nodes' local memories. High memory access latencies are masked through massive parallelism. This paper introduces the basic principles and properties of RSM and analyzes its practical efficiency in terms of constant factors through simulation studies, assuming a state-of-the-art parallel architecture. Bottlenecks in the architecture are pointed out, and improvements are being made and their effects assessed quantitatively. The results show that RSM efficiency is encouragingly high, even in a non-optimized architecture. We propose architectural features to support RSM and conclude that RSM may indeed be a feasible shared-memory implementation in future massively parallel computers.

1 Introduction

Within the class of parallel MIMD computers, systems with shared memory are widely preferred over their distributed-memory counterparts. This is due to more convenient software development (programming models) and tool support (e.g. automatic parallelization, debugging, load balancing) facilitated by the system-wide global memory.

Traditionally, most shared-memory parallel machines have been built around a common bus and with physically common memory. Due to the use of centralized resources, these systems do not scale beyond the range of tens of processors at most. In the design of massively parallel computers, an attempt is currently being made to offer the convenience of a shared-memory image while maintaining the superior scalability of distributed-memory hardware.

Distributed Shared Memory (DSM) has emerged as the most promising approach to achieve this goal [7, 13]. DSM systems typically employ some

www.manaraa.com

form of *caching* to reduce global memory access latencies and achieve acceptable performance. Initially, most DSM schemes have been implemented in software, extending conventional virtual memory management.

Recently however, a number of hardware-based DSM schemes have been proposed, which are destined to be *scalable* into the range of at least hundreds of processors [3, 4, 6, 10, 12, 15, 16]. The key elements of these proposals are scalable cache coherence protocols, which usually are extensions of conventional multiprocessor cache protocols. The approaches pursued in these designs are manifold: weak coherency models, hierarchical or cluster-based configuratons, distributed directories, and/or cache-only memory architectures. Some of these systems are currently being built or already operational. It will be interesting to see whether they succeed in providing truly scalable shared memory.

In this paper, a radically different approach to scalable shared memory will be explored. The approach is based upon recent work in theoretical computer science which has disclosed ways how to implement *provably scalable and universally efficient* DSM [17, 18]. The proposal is based upon global *memory randomization (address hashing)*, that is, pseudo-random distribution of the global address space among the nodes' local memories. High memory access latencies are masked through massive parallelism. That is, this scheme builds upon latency hiding rather than latency reduction. In the following, the term *Randomized Shared Memory (RSM)* will be used to denote this proposal for shared memory realization.

In the first place, one would intuitively anticipate very poor performance. However, this scheme has been shown to optimally emulate (in an asymptotic sense) shared memory in distributed-memory machines, provided that software provides sufficient parallelism for effective latency hiding. RSM is therefore a potentially significant proposal and worthwhile to be investigated further.

This paper introduces the theoretical background and basic principles of RSM and then analyzes its practical efficiency in terms of constant factors through simulation studies, assuming a state-of-the-art parallel architecture. Inefficiencies of the architecture are pointed out, and improvements are being made and their effects assessed quantitatively. Finally, we propose architectural features to support RSM and conclude that RSM may indeed be a feasible shared-memory implementation in future massively parallel computers.

2 Randomized Shared Memory

RSM was introduced in the theoretical literature in the framework of studying emulations of an idealized shared-memory parallel computer, the

www.manaraa.com

Parallel Random Access Machine (PRAM), on realistic distributed-memory hardware.

The PRAM is a theoretical machine which abstracts from any architectural and hardware constraints. It offers a powerful, architecture-independent programming model without imposing limitations on parallelism, and therefore allows parallel computation to be studied *per se*. It has extensively been used to develop and study (efficient) parallel algorithms, to devise techniques for parallel algorithm design, and to develop a parallel complexity theory [5].

The PRAM model essentially consists of a (possibly infinite) number of processors, accessing and communicating via a global memory (possibly of infinite size). The processors operate in lock-step synchrony. In each step of the computation, every processor performs a read-memory, compute, write-memory cycle. Memory accesses are of uniform cost and take unit time. There is no notion of memory locality.

To make PRAM algorithms more readily applicable in practical problems and machines, theoreticians devised methods to emulate the PRAM on (models of) real machines. A key element of these methods is shared memory emulation in a distributed-memory system using *memory randomization*.

Fig. 1 illustrates the basic principle of memory randomization. The technique pseudo-randomly distributes global data and, at run time, global accesses to those data on a *per-word* basis throughout the system's memory modules, i.e. the nodes' local memories. Data in RSM are neither cached (replicated) nor migrated. As a result, most of the memory accesses are directed at remote memory units, which incurs high memory access latencies and generates intense network traffic. For practical purposes, randomization needs to be applied to global data only. Data that are private to a node can be held in a conventionally addressed portion of the node's local memory.

The purpose of hashing is to spread out global addresses and accesses as uniformly as possible across the memory units of the system, avoiding to overload individual units even if the memory accesses are arbitrarily non-uniform. Similarly, memory request/reply traffic in the interconnection network is randomized and the danger of hot spots or systematic blocking, which may severely impair machine performance, is reduced.

From a practitioner's point of view, the RSM approach is counter-intuitive. Since locality of reference cannot be exploited at all and high costs are associated with each memory access, one would intuitively anticipate very poor performance. However, recent theoretical work by Valiant on optimal PRAM emulation [17, 18] has rendered RSM potentially significant and feasible for practical use. The following features make RSM an interesting DSM implementation proposal:

www.manaraa.com

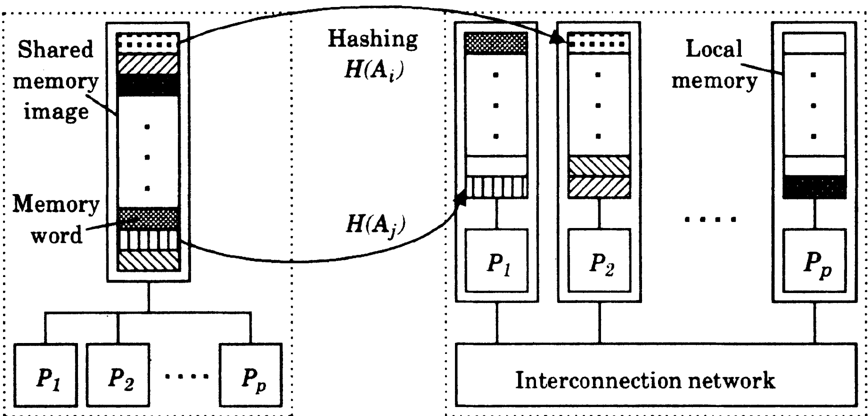


Fig. 1 RSM concept

- When memory access latencies are adequately masked through massive parallelism (see below), RSM is an *optimally efficient* DSM implementation (in an asymptotic sense).
- RSM *scales with constant efficiency*.
- RSM is *universally efficient*.

Valiant's major contribution has been to propose hiding memory access latencies through a high degree of parallelism exhibited by application programs (*excess parallelism* or *parallel slackness*). The idea is outlined in the sequel. A more detailled description of this concept appears in [8]. The mathematical treatment is given in [17]. The quantitive figures given below hold for the emulation of an exclusive-read, exclusive-write (EREW) PRAM. Results for other PRAM variants can be found in [17] as well.

Let p denote the number of processors of the real, distributed-memory machine that is to emulate an EREW PRAM. The PRAM executes synchronously in *steps*, as described above; a step takes unit time. Assume that a given PRAM algorithm utilizes v virtual PRAM processors. Given that $v = p \cdot \log p$, the PRAM algorithm can be executed with optimal asymptotic efficiency on the real machines as follows. (This is a lower bound, chosen here for sake of presentation; the result holds for any $v \geq p \cdot \log p$.)

The v virtual PRAM processors are evenly mapped to processes on the p physical processors. The PRAM shared memory is randomized throughout the real system's local memories. Each PRAM step is emulated by a *superstep* on the real machine. In each superstep, each node of the real machine executes $\log p$ processes concurrently. Whenever a process accesses global memory and the access is directed at a remote memory (through the hashing function), the requesting process is descheduled, and another process

المنارة للاستشارات
www.manaraa.com

resumes execution. In other words, frequent process switching is employed to hide the communications latencies involved in memory accesses.

All the processes together issue $O(p \cdot \log p)$ memory requests in a single superstep, $O(\log p)$ per node. Valiant has shown that there exist networks, e.g. hypercubes and butterfly topologies, which with very high probability can route all those requests and, therefore, service all the memory accesses in $O(\log p)$ time. Consequently, on each node of the real machine, the $\log p$ processes (PRAM virtual processors) are executed (emulated) in $O(\log p)$ time, which is optimal up to constant factors. At the end of each superstep, all processes *explicitly* engage in barrier synchronization. Since global synchrony is established at a much coarser grain than in a PRAM, the term *bulk-synchronous parallelism (BSP)* has been coined for this style of execution.

In effect, Valiant's analysis shows that any program written in a high-level shared-memory programming model can be emulated on the BSP model and thus on a real, distributed-memory system with only a *constant-factor loss of efficiency C*. The major prerequisite is that software provides sufficiently many processes (excess parallelism) for effective latency hiding.

It must be noted that fig. 1 gives a simplified picture of memory randomization. Only a single hash function is depicted which is assumed to yield both a node address and the physical address within that node's local memory. The theoretical analysis [17] requires two hashing steps and functions, *global* and *local* hashing. In addition, the hash functions must be *universal*, i.e. generate a provably (approximately) even distribution of global memory locations throughout the memory modules. Computing these functions involves evaluating a polynomial of degree $O(\log p)$.

For practical purposes, however, a *single, linear* hash function is assumed to be sufficient [1]. The universality property does no longer hold for such a function, but its hardware implementation is cheaper and the address translation is faster than for universal hashing. A linear hash function is therefore assumed for the RSM evaluation reported in this paper.

3 RSM Performance Evaluation

The actual practical relevance of RSM is determined by the constant-factor loss of efficiency C involved in the PRAM emulation. We have therefore assessed the practical efficiency of RSM in a state-of-the-art parallel architecture. In this section, we briefly outline the approach taken for RSM performance evaluation. A detailled description is given in [8].

Trace-driven simulation has been used to assess RSM efficiency. The simulations encompass in full detail the global memory access and global syn-

www.manaraa.com

chronization behaviour of application processes and all resulting activities, such as address translation (hashing), process scheduling, communications (message passing), actual memory accesses, and barrier synchronization events. All these operations are covered *on a per-process basis*. Fig. 2 illustrates the activities on a node when a process reads a global variable stored in a remote memory module.

The simulator for RSM performance analysis is an extension of an existing interconnection network simulator [9]. All the network properties, such as throughput and latency, and dynamic effects, like contention and queueing, are captured in close detail in the RSM simulations. This is important since RSM efficiency is, to a high degree, determined by network behaviour.

The *architectural model* is a distributed-memory parallel computer based on the new generation of transputer components, the Inmos T9000 processor and C104 routing switch [11]. On each node, there is a memory management unit, termed *RSM Server*, that implements the global memory abstraction for its processor. The interconnect of the architecture was chosen to be a multi-stage, Clos-type, four-fold replicated network performing adaptive wormhole routing [9].

The T9000 transputer was regarded as particularly well-suited for RSM simulations because of its hardware support for rapid process scheduling and communications. For example, process switching was initially assumed to take 1 µs only (40 cycles on a 40 MHz T9000). To make global memory management comparably fast, we modelled the RSM Server as a hardware unit similar in performance to the T9000 communications coprocessor. Fig. 2 exemplarily illustrates the functional and timing behaviour of the RSM Server (and of other functional units). RSM Server functionality as well as performance is specified in more detail in [8].

The RSM Servers collectively implement the common memory image on the distributed-memory hardware, relieving the CPUs from controlling global memory accesses. In our model, the RSM Servers are also responsible for global synchronizations. Initially, only a simple *message-based, centralized, two-tier* barrier scheme was implemented. It is obvious that such a scheme is not scalable. Since synchronization performance was not a focus of our investigations, it was initially regarded as sufficient for our purposes.

The *workloads* driving the simulations are either *application traces* or *synthetic reference patterns*. The traces capture the global memory access and synchronization behaviour of application kernels on a per-process basis. The kernels have been written according to the BSP model, i.e. with excess parallelism and in a superstep-wise fashion, and have been executed on small transputer arrays (up to 17 nodes) with shared memory emulated in software. The kernel routines fall into three classes:

www.manaraa.com

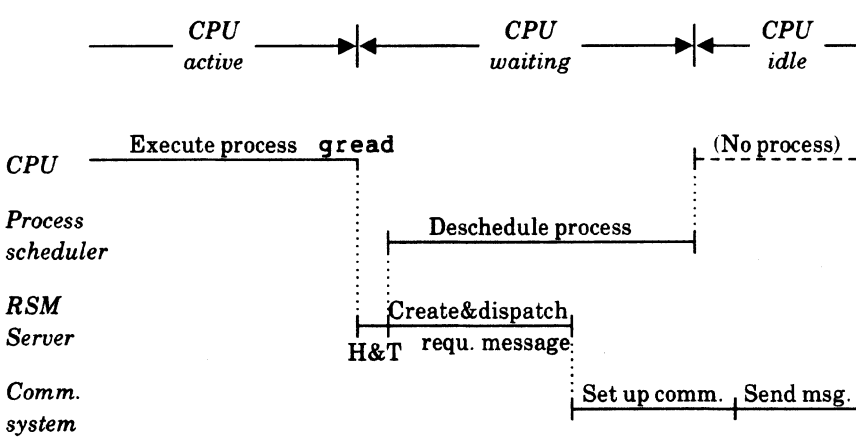


Fig. 2 Activities of a node's functional units on a global read operation (`gread`)

- *numerical kernels:* dense and sparse dot products, matrix-vector multiplication, matrix multiplication, matrix transpose, and 1D complex FFT;
- *low-level image processing operations*: thresholding, lowpass filtering, edge detection (Sobel), and simple growing and thinning operations;
- *PRAM-specific routines:* parallel prefix sum and list ranking.

The traces enable RSM performance figures to be obtained on the basis of *realistic* global memory access and synchronization behaviour. Their drawback is that they are confined to a fixed, small number of nodes.

The synthetic workloads are used for two purposes: to obtain RSM efficiency results for larger systems (i.e. to address the issue of scalability), and to more systematically investigate the impact of load characteristics (e.g. degree of excess parallelism) on RSM performance. To that end, the synthetic load generator allows variations in a number of parameters, most notably system size (number of nodes), excess parallelism (number of processes per node), and the rate (frequency) of global reads or writes. The spatial distribution of global memory accesses does not produce systematic conflicts. Synthetic loads will therefore yield best-case RSM performance figures.

The *evaluation criteria* for determining RSM efficiency are based upon considering three classes of CPU activities; see fig. 2. A CPU is *active* when an application process is currently executing. A CPU is *waiting* during global address translation (hashing), local access to a global data item, and process (de)scheduling. As shown in fig. 2, the main constituent of CPU waiting time is the process switching time (>90%). Finally, a CPU is *idle* when no application process is available to be executed. All processes are inactive, waiting for their global memory accesses or barrier synchronization re-

www.manaraa.com

quests to be serviced. Therefore, we differentiate between CPU *memory idle* and *sync idle* times.

RSM efficiency is expressed by splitting the *elapsed time* (simulated program run time) up into four parts:

- *average CPU active time and rate (percentage of elapsed time);*
- *average CPU waiting time and rate;*
- *average CPU memory idle time and rate;*
- *average CPU sync idle time and rate.*

The first criterion enables the *constant-factor loss of efficiency C* to be defined as

$$C := 1 / (\text{average CPU active rate}).$$

It is thus the most important measure of RSM efficiency. The latter three criteria allow the major sources of inefficiencies to be identified and assessed quantitatively.

In an idealized machine (e.g., a PRAM) with ideal global memory, zero-time process switching and perfectly synchronous operation, the CPUs would be fully active, yielding $C=1$. $C>1$ in a real system is caused by the global memory emulation, process switching time, barrier synchronization latencies, and potentially insufficient degree of excess parallelism. Notice that a message-passing version of a given program or an implementation on physically shared memory would also have some $C>1$.

4 RSM Efficiency

In this section, the major results on RSM efficiency are reported. First, the basic results that have already been described in [8] are summarized. They allow an initial value of C to be determined and the major sources of inefficiencies to be identified. In a further step, the effects of architectural improvements to mitigate these inefficiencies are investigated.

4.1 Basic Results

Fig. 3 summarizes representative initial results on RSM efficiency, which are based on the architectural model as described in the previous section. More results and the details on the traces and synthetic loads underlying these simulations are given in [8].

The graphs indicate that RSM (more precisely, the specific RSM implementation model described above) induces a typical CPU active rate (efficiency) of 10% to 20%, which means that the constant-factor loss of efficiency C is in the range of 5 to 10.

www.manaraa.com

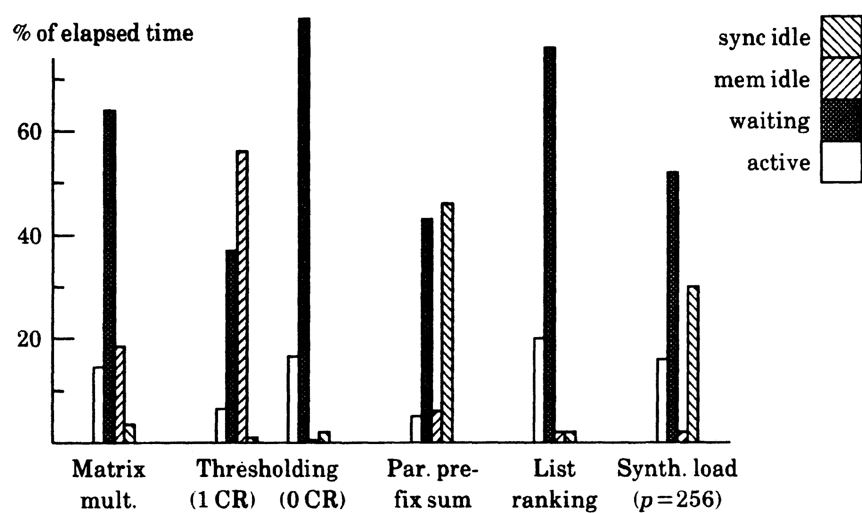


Fig. 3 Distribution of CPU time for sample loads

This is regarded as encouragingly high, since high global memory access rates have been assumed for the simulations. For example, the matrix multiplication program/trace has been assumed to issue a global memory access after every 15 local processor cycles (on an average). Various experiments indicate that the CPU efficiency is highly sensitive to the global access rate [8]. Thus, quite good CPU efficiency can be expected when global requests occur at a coarser grain. Furthermore, the architectural model is not optimized for RSM support. Several sources of inefficiencies can be identified, either due to architectural bottlenecks or specific application characteristics.

The results clearly show that the process switching time (although modelled to take 40 processor cycles only) is the major bottleneck in the emulation. For loads which do not have significant application-specific inefficiencies, the processors spend most of their time (>60%) waiting for address translation and, most notably, process scheduling to take place. Reducing process switching time must therefore be considered the single most important measure to support RSM. This issue is dealt with in subsection 4.3.

For the matrix multiplication trace, the CPU memory idle time accounts for about 20% of the simulated run time. This is due to an insufficient degree of parallelism. For the specific global memory access rate chosen in this simulation, the four processes per node (in a 17-node parallel system) cannot fully mask the ensuing memory latencies. It must be emphasized that the CPU memory idle rate closely depends on the global memory access frequency. In other words, in a program or simulation with a lower ac-

المنارة للاستشارات
www.manaraa.com

cess rate assumed, four processes may well suffice to satisfactorily hide the latencies.

In contrast, the extremely high CPU memory idle rate of the thresholding program is caused by the application performing a massively concurrent read (CR) operation. Several thousand processes concurrently access a common variable (the threshold value). This leads to massive congestion at the memory module holding the variable and in the interconnection network ("hot spot" behaviour). The read accesses can only be serviced serially by the responsible RSM Server. Despite the massive parallelism provided by the program (400 processes per node), the CPUs run out of active processes and become memory idle eventually. As a result, CPU efficiency becomes poor. The CR operation is eliminated in the second version of this program, leading to acceptable efficiency and negligible memory idle time.

For the parallel prefix operation, the CPU sync idle time is the dominant constituent. This is due to global synchronizations occuring at a very high frequency, with an average of less than one global memory access being performed per process and superstep.

High CPU sync idle times also occur in larger systems, as shown by the synthetic load example in fig. 3. A system of size $p = 256$ nodes, 16 processes per node and more than 300.000 global memory accesses, but only 10 supersteps have been chosen for this simulation. Clearly, the centralized barrier synchronization scheme implemented in the initial model seriously impairs performance for larger systems and does not even scale into the range of hundreds of processors. Improved barrier implementations and their effect on performance and scalability are therefore investigated in subsection 4.2.

Experiments with varying degree of excess parallelism in synthetic loads have also shown that, in larger systems, almost exactly $\log p$ processes per node suffice to fully hide global memory latencies [8]. This closely corresponds to the theoretical results. Clearly, this one-to-one correspondence holds for the specific architectural model and simulation parameters underlying these experiments only. Changing the assumptions would cause a constant-factor increase or decrease of the required degree of excess parallelism.

In summary, the initial performance figures indicate that up to 20% efficiency of idealized shared memory can be attained in an RSM-based parallel architecture that is principally implementable with state-of-the-art technology. This result holds in case sufficient parallelism is available to mask memory latencies, and for high global memory access rates assumed. Higher efficiency could be expected for global memory accesses occurring less frequently. However, quite high global access rates must be expected in fine-grained parallel shared-memory programs specifically written for RSM-based systems.

www.manaraa.com

Two major architectural bottlenecks were identified, limiting or impairing RSM performance and scalability: the process switching time and the message-based, centralized barrier synchronization scheme. Improvements of these two factors will therefore be investigated subsequently.

Apart from these two factors, the interconnection network becomes a severe bottleneck in case of concurrent global read (or write) accesses being issued by the application program. A combining network would cope well with such load conditions.

No other principal impediments to a scalable and efficient RSM implementation have been found under favourable circumstances, i.e. with sufficent degree of excess parallelism available and global memory access behaviour without systematic conflicts. Thus, the results principally confirm the desirable theoretical properties and indicate encouraging practical efficiency of RSM.

4.2 Improved Barrier Synchronization

An improved barrier scheme, which arranges the nodes participating in global synchronizations in a binary tree, has been implemented. In the course of a global synchronization, sync request messages travel up the tree. When having received sync request messages from both its subtrees, the root knows that all nodes have joined the synchronization event and sends sync acknowledge messages down the tree. Thus, messages are propagated up to log p levels up and down the synchronization tree.

For comparison purposes, an idealized barrier scheme has also been implemented. This scheme assumes dedicated synchronization hardware which enables any node to propagate a sync request signal to a master node in 1 µs only. Conversely, the master node can broadcast a sync acknowledge signal to all participants within 1 µs as well, independent of system size.

Fig. 4 depicts the results of simulations that allow the effects of different barrier schemes to be assessed. The experiments have been performed with system size p varying from 64 to 512 nodes, and with 16 processes per node, 10 supersteps, and the number of accesses *per node* remaining constant.

The left graph shows that, with increasing system size, total program run time increases proportionally to sync idle time. Moreover, it becomes evident that sync idle time increases linearly with system size for the centralized barrier scheme, and logarithmically for the tree-style barrier. This is emphasized by the right graph.

The results also indicate that, while the barrier tree improves over the centralized scheme in terms of performance and scalability, it still entails a high percentage of sync idle time, up to about 25% for $p = 512$. This renders this type of message-based synchronization impractical as well.

www.manaraa.com

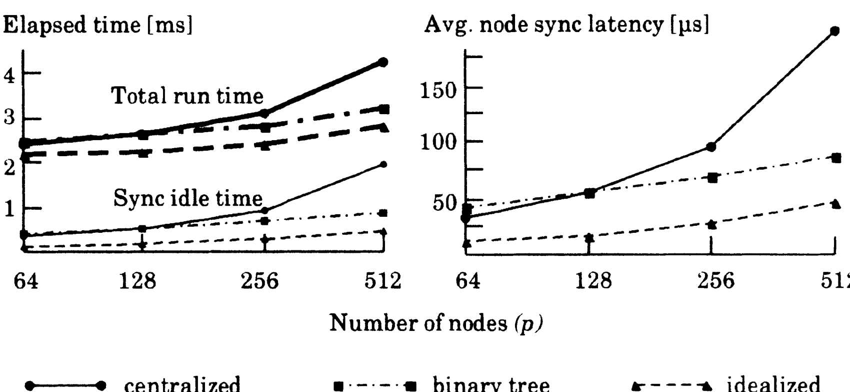


Fig. 4 Comparison of different barrier synchronization implementations

The idealized barrier implementation should be expected to scale with constant efficiency, i.e. the average node sync latency should remain constant with increasing system size. The node sync latency is defined as the time passing between a node sending out the sync request message (or signal) and receiving the corresponding sync acknowledge message (or signal).

In the first place, it is quite surprising that the average node sync latency of the idealized barrier increases significantly with system size. The explanation is as follows. In these simulations, the overall number of processes increases proportionally with system size. Thus, the probability that the processes and, eventually, the nodes run out of synchrony increases. This is caused by the processes encountering different global read and write latencies which in turn are due to blocking effects in the network. As a consequence, "fast" nodes may have to wait quite a long time for "slow" nodes at a barrier, thus increasing the average node sync latency.

We conclude that in general the bulk-synchronous style of execution as proposed by Valiant is not well suited for highly parallel programs and computers because of the inefficiencies incurred by establishing global synchrony.

4.3 Reduced Process Switching Time

The basic architectural model was extended to allow the process switching time to be scaled down. Experiments have been performed with process switching reduced to as low as 20% of the original value, i.e. to 8 cycles.

Fig. 5 shows the effect of reducing process switching time on the run time of a synthetic load modelled according to the parameters of the matrix multiplication program/trace. This load allows the number of processes per node to be varied. This is important since with decreasing process scheduling time the degree of excess parallelism must increase for effective latency hiding. System size is $p = 17$ nodes, chosen such that sync idle time is negligi-

www.manaraa.com

ble. Increasing the number of processes is done such that each process is assigned proportionally less work; the overall amount of work remains constant.

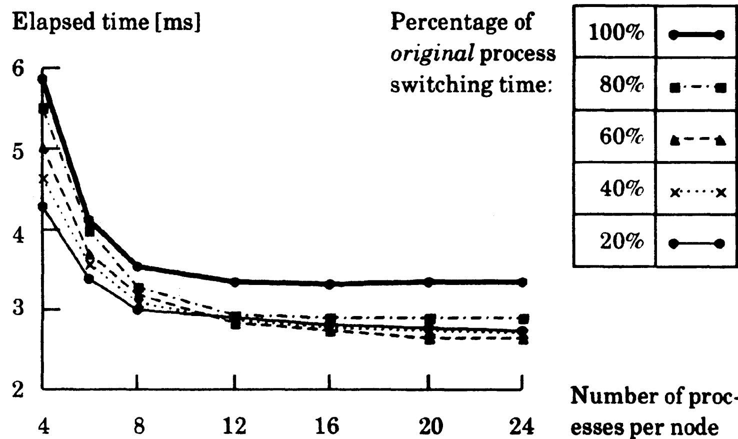


Fig. 5 Effect of reducing process switching time (synthetic matrix mult. load)

The results show that reducing process switching time to 80% and 60% of the original value does in fact reduce program run time, but not to the same degree. The reason is that, at a given number of processes per node, memory idle time increases with decreasing process switching time, eliminating part of the efficiency gained by lower waiting time.

More importantly, reducing process switching time to 40% and 20% of the original time, does not further decrease the run time; another factor of inefficiency emerges. Fig. 6 provides an explanation for this behaviour.

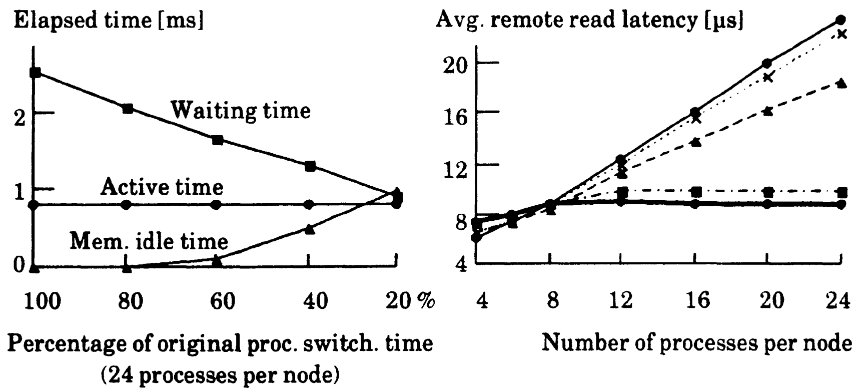


Fig. 6 Analysis of synthetic matrix multiplication load

www.manaraa.com

The left graph shows, for the program version with 24 processes per node, how the distribution of the CPU time evolves for different process scheduling times. While the waiting time decreases linearly, the memory idle time becomes significant for small process switching time. This indicates that there is a bottleneck in the memory system. Notice that for the smallest process switching time, CPU efficiency (active rate) is at about 30%.

The right graph shows that, for small process switching times, the global memory read latencies increase roughly linearly with the number of processes per node. The curves indicate that the interconnection network becomes a bottleneck. Many processes per node and rapid context switching induce many concurrent global memory accesses, ensuing heavy request and reply traffic in the network. Apparently, such heavy load leads to more intense internal blocking, resulting in higher message latencies. It must be noted that in this 17-node system a message is routed through a single C104 switch only. In larger systems with multi-stage networks, this inefficiency must be expected to have a more severe effect on RSM performance.

We therefore conclude that an isolated architectural improvement is of limited benefit only. In general, it must be obeyed that the architecture remains balanced when improvements are made.

5 Conclusions

In the course of optimal PRAM emulation, RSM has been shown to be an attractive approach to implement scalable, universal, and theoretically efficient shared memory. In this paper, we described the basic principles of RSM and analyzed its efficiency in terms of constant factors.

In principle, the practical efficiency of RSM was found to be encouragingly high, if sufficient parallelism was available to effectively mask memory access latencies. Up to 20% CPU efficiency could be achieved with a parallel system model which is principally implementable using today's technology.

Process switching time and message-based global synchronizations were identified as the principal architectural impediments to better performance and scalability. Major inefficiencies caused by application programs were found to be an insufficient degree of excess parallelism, very high global memory access rate, and concurrent reads (or writes) to global data.

Subsequently, two architectural improvements were investigated: reduced process switching time and improved barrier implementation. More efficient process switching was found to significantly increase performance, and CPU efficiencies of up to 30% were observed. However, this measure caused the interconnection network to become a bottleneck eventually. We

www.manaraa.com

conclude that architectural optimizations for RSM must, in order to be effective, keep the architecture balanced.

Barrier synchronization turned out to remain a significant source of inefficiency even when an idealized implementation was assumed. We conclude that the bulk-synchronous model of parallelism (BSP), which proposes that global synchrony is established periodically, is not well-suited for highly parallel computers. It must be noted that the principles of RSM, in particular the idea to hide high memory access latencies through massive parallelism, does not preclude other synchronization schemes to be implemented.

The results indicate that RSM may indeed be a feasible implementation of shared memory in future massively parallel computers. The following features are proposed to support RSM in these architectures:

- *extremely fast (or zero-time) process switching* as implemented by processors with multiple instruction streams (hardware contexts);
- a *high-bandwidth combining network* that is able to on-the-fly combine global memory reads (and preferably also writes); and
- an *efficient and scalable synchronization mechanism* supported by the architecture, such as provided by `fetch&op` or `(multi-)prefix` memory primitives [1] or synchronization bits associated with memory locations [2].

Several novel machine designs document increased interest in RSM and such architectural features [1, 2, 14]. Given such support, RSM should be expected to be an efficient DSM implementation and, in particular, scale to massively parallel systems with close to constant efficiency.

Acknowledgment

The work reported in this paper was funded in part under ESPRIT P2701 (*P*arallel *U*niversal *M*essage-passing *A*rchitectures).

References

[1] F. Abolhassan, J. Keller, W.J. Paul, *On the Cost-Effectiveness and Realization of the Theoretical PRAM Model*, Report SFB 124/D4, 09/1991, Univ. Saarbrücken, 1991.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, B. Smith, "The Tera Computer System", *ACM SIGARCH Computer Architecture News 18(3) (Proc. 1990 Int'l. Conf. on Supercomputing).*

[3] D. Chaiken, J. Kubiatowicz, A. Agarwal, "LimitLESS Directories: A Scalable Cache Coherence Scheme", *Proc. ASPLOS IV*, ACM, 1991.

www.manaraa.com

[4] D.R. Cheriton, H.A. Goosen, P.D. Boyle, *ParaDiGM: A Highly Scalable Shared-Memory Multi-Computer Architecture,* Report No. STAN-CS-90-1344, Standford University, Nov. 1990.

[5] A. Gibbons, W. Rytter, *Efficient Parallel Algorithms,* Cambridge University Press, 1988.

[6] E. Hagersten, A. Landin, S. Haridi, "DDM—A Cache-Only Memory Architecture", *COMPUTER 25(9),* 1992.

[7] H. Hellwagner, *A Survey of Virtually Shared Memory Schemes*, SFB Report No. 342/33/90 A, Techn. Univ. Munich, 1990.

[8] H. Hellwagner, "On the Practical Efficiency of Randomized Shared Memory", *Proc. CONPAR'92–VAPP V,* LNCS 634, Springer 1992.

[9] H. Hofestädt, A. Klein, E. Reyzl, "Performance Benefits from Locally Adaptive Interval Routing in Dynamically Switched Interconnection Networks", *Proc. EDMCC'2,* LNCS 487, Springer 1991.

[10] *IEEE Std 1596-1992 Scalable Coherent Interface (SCI),* IEEE CS Press, 1992.

[11] Inmos Ltd., *The T9000 Transputer Products Overview Manual,* First Edition 1991.

[12] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, M. Lam, "The Stanford DASH Multiprocessor", *COMPUTER 25(3),* 1992.

[13] B. Nitzberg, V. Lo: "Distributed Shared Memory: A Survey of Issues and Algorithms", *COMPUTER 24(8),* 1991.

[14] R.D. Rettberg, W.R. Crowther, P.P. Carvey, R.S. Tomlinson: "The Monarch Parallel Processor Hardware Design", *COMPUTER 23(4),* 1990.

[15] J. Rothnie, "Kendall Square Research Introduction to the KSR1", in: H.-W. Meuer (ed.), *Supercomputer'92,* Springer 1992.

[16] M. Thapar, B. Delagi, "Scalable Cache Coherence for Large Shared Memory Multiprocessors", *Proc. CONPAR'90–VAPP IV,* LNCS, Springer 1990.

[17] L.G. Valiant: "General Purpose Parallel Architectures", in J. van Leeuwen (ed.): *Handbook of Theoretical Computer Science*, North-Holland, Amsterdam, 1990.

[18] L.G. Valiant: "A Bridging Model for Parallel Computation", *Comm. ACM 33(8),* 1990.

www.manaraa.com

Methods for Exploitation of Fine-Grained Parallelism

Günter Böckle, Christof Störmann, Isolde Wildgruber
Siemens AG
Central Research and Development
Otto-Hahn-Ring 6, D-8000 München 83
E-Mail: boe@zfe.siemens.de

Abstract

Fine-grained parallelism is offered by an increasing number of processors. This kind of parallelism increases performance for all kinds of applications, including general-purpose code; most promising is the combination with coarse-grained parallelism. Unlike coarse-grained parallelism it can be exploited by automatic parallelization. This paper presents program analysis and transformation methods for exploitation of fine-grained parallelism, based on global instruction scheduling.

1 Introduction

Coarse-grained parallelism is well suited for programs with sufficient parallelism inherent to the algorithms they are based on. The parallelism can most efficiently been exploited by stating it explicitly in the program, while automatic parallelisation is still a research topic for coarse-grained systems. Only loops of the FORALL-style with suitable array privatization for distributed-memory systems can be parallelized satisfactorily. While fine-grained parallelism cannot offer the high degree of parallel execution as coarse-grained parallelism, the parallelization can be performed automatically and thus it is well suited for sequential programs and the sequential parts of parallel programs.

Fine-grained parallelism uses concurrent processing of machine instructions - a level which cannot be seen by a programmer and a performance potential which cannot be exploited at a higher level. For high-level source-code statements which are strictly sequential we still can find machine instructions compiled from these statements which can be executed in parallel. Thus, the two granularities of parallelism complement each other. There are already multiprocessor systems using both kinds of parallelism - coarse- and fine-grained. The Intel Paragon is a coarse-grained parallel system with nodes which offer fine-grained parallelism. For such multiprocessor systems the parallelization of both granularities not only benefit from each other by using the same methods, but combined coarse/fine-grained parallelization methods promise higher performance for application programs.

2 Architectures with Fine-Grained Parallelism

Fine-grained parallelism is becoming common in microprocessor development by offering several functional units per processor - an obvious trend, considering all the new superscalar

www.manaraa.com

processors announced or provided by most microprocessor companies. Currently, the exploitation of this parallelism is performed dynamically in hardware by fetching several instructions concurrently and issuing them to the corresponding processing elements. However, the degree of parallelism which can thus be achieved is quite limited (see [6]); for taking advantage of this parallelism, it has to be exploited by corresponding software methods.

In our group we have built a set of tools for the exploitation of fine-grained parallelism. The methods applied are based on program analyses and transformations by reordering the code statically at compile time so that machine instructions which can be executed in parallel are grouped together.

Such methods have been developed before for specific fine-grained parallel architectures, Very Long Instruction Word computers. These architectures offer several heterogeneous processing elements which are controlled by very long instruction words with one operation field for each processing element. Thus, these processing elements execute their machine operations, grouped in one instruction word, all in the same clock cycle. The machine operations are reordered at compile time and compacted to the very long instruction words. The methods used were derived from microcode compaction for horizontally microcoded machines.

3 Methods for Static Fine-Grained Parallelism Exploitation

The methods for fine-grained parallelization are based on reordering intermediate or machine code so that groups of instructions are formed which can be executed in parallel. The main requirement for parallel execution is data and control independence, two dependent instructions have to be executed successively. All instruction scheduling methods perform program analysis first to determine dependences between instructions.

Inside a basic block, i.e. a sequential part of the control flow, only data dependences have to be considered. Moving an instruction from one successor path of a branch instruction before the branch may change a variable read in the other path of the branch instruction and thus alter program semantics. Thus, for reordering instructions globally, i.e. beyond basic block boundaries, data-flow analysis is necessary, which can be quite time consuming. Therefore, many reordering methods use local scheduling, inside basic blocks only, although global scheduling has a higher parallelization potential.

Methods for local instruction scheduling have already been developed for the CDC 6600. Since 1981 several global scheduling methods have been published, mostly for horizontal microcode compaction (see [1] - [5], [7] - [11]). Two classes of global methods can be distinguished, one where a program's flow graph is partitioned into regions of traces or trees and a local scheduling method like List Scheduling is applied to these regions. At the interfaces between the regions the program semantics may be changed by these methods, thus it must be reconstructed by (usually quite costly) bookkeeping methods. The second class comprises methods applying scheduling to the program graph where the nodes contain independent operations which can be executed in parallel, and are attributed with dependence information. Semantics-preserving transformations for reordering are provided, and reordering is separated from strategies controlling reordering so that no bookkeeping is necessary.

3.1 Global Scheduling Methods Based on Local Scheduling

Trace Scheduling ([3], [4], [10]) was developed by J. Fisher for horizontal microcode compaction. A function of a program is separated into traces by first determining branch proba-

www.manaraa.com

bilities for each branch operation, using heuristics or profiling. Traces end e.g. at back edges of loops. A trace is thus a sequential part of the program and can be rearranged by List Scheduling. To preserve program semantics, bookkeeping is necessary where additional code is inserted at entries and exits of traces.

The ITSC (Improved Trace Scheduling) method ([11]) reduces the amount of code-size increase due to bookkeeping by separating a trace's operations into two sets with different scheduling priorities, starting with operations on the critical path and the operations dependent on those.

The tree-method ([7]) decreases the amount of code increase in Trace Scheduling by separating a program's flow graph into "top trees" which contain mainly nodes which are not join or leaf nodes and in "bottom trees" which contain mainly nodes which are not start nodes or fork nodes. First top trees are scheduled, then bottom trees.

The SRDAG method ([8]) allows for more parallelization potential. Single rooted directed acyclic graphs (SRDAGs) are considered for scheduling, not just sequential traces. However, bookkeeping is more complex than in Trace Scheduling.

3.2 Global Scheduling on the Program Graph

Percolation Scheduling ([9]) separates the algorithms for moving instructions from the algorithms controlling the application of these movements. Scheduling is performed on the program graph where the nodes contain operations which can be executed in parallel. Core transformations are used to move an instruction from one node to a preceding node and control algorithms determine the sequence of operations to be moved. Methods like Percolation Scheduling have a higher flexibility than methods of the other class, more kinds of moves are possible; in Trace Scheduling, moves across traces are not possible while there are no trace borders in Percolation Scheduling. However, resource allocation is quite difficult in methods of the second class.

In [1] a version of Trace Scheduling was developed, based on Percolation Scheduling's core transformations. The complex bookkeeping phase of Trace Scheduling can be omitted because the core transformations insert correction code where necessary while moving operations.

In [2], Percolation Scheduling was extended for architectures with speculative execution of instructions. In this model the operations in a program graph node form a tree with the conditional branch operations as nodes and the other operations at the edges. Restrictions due to limited resources are considered for scheduling in this method, too. For each node the operations which can be moved there are determined and the program graph's nodes are filled by choosing the best of these operations.

A method using an extended program dependence graph for global scheduling is Region Scheduling, described in [5]. A program is separated into regions consisting of one or several basic blocks and in these regions loop transformations and instruction reordering are performed.

4 Program Analysis

A prerequisite for instruction scheduling is sufficient knowledge about dependences of the instruction to be moved. We developed and implemented interprocedural data dependence and data-flow analysis methods for our tools. In our system model, all processing is per-

www.manaraa.com

formed in symbolic registers, program variables are either memory addresses or directly mapped to symbolic registers. For each operation the registers read and written, as well as the memory locations read and written are determined and accumulated for all operations in a program graph node. These data are also accumulated for all functions and the information propagated to the operations (and nodes) calling the functions. Thus, we have complete knowledge about data dependence at all relevant places, the operations, the nodes, and the functions.

4.1 Analysis of Symbolic Registers

For moving operations beyond conditional branches, data-flow information is required. If an operation is moved above a conditional branch, it must not write to any location (register or memory) read by any successor in the other path emanating from the branch operation. This means, it must not write to any register or memory variable live in the conditional branch's successors. This data-flow information is gathered in two steps, first intra- then interprocedurally. First, the sets use(n) and def(n) of a program graph node n are determined, initially as the set of registers read in n resp. the set of registers written in n which are not in use(n):

use(n) = rreads(n) with rreads(n) = {registers read in node n}

def(n) = rwrites(n) - rreads(n) with rwrites(n) = {registers written in n}

These sets are then enlarged interprocedurally by adding information about functions called in n:

$$\text{use}(n) = \text{use}(n) \cup (\text{live_at_entry}(f) - \text{def}(n)),\ f \text{ called in } n$$

$$\text{def}(n) = \text{def}(n) \cup ((\text{dead}(f) \cap \text{must_mod}(f)) - \text{use}(n)),\ f \text{ called in } n$$

The set dead(f) is the set of registers dead at the entry of a function f and must_mod(f) is the set of registers written on each path through f. The sets use(n) and def(n) are needed to determine our actual target, the set in(n) of variables live at entry of a node n, which is initialized with use(n) and enlarged by standard data-flow analysis using the interprocedurally determined sets use(n) and def(n). For the determination of in(n), the standard set out(n) of registers live at the exit of a node is used. Interprocedural analyses show the flow of liveness information in both directions - from a called function to the caller and back; the set out(n) is used to collect this information:

$$\text{live_at_exit}(f) = \bigcup_{\forall g \text{ calling } f} \ \bigcup_{n \text{ in } g} \text{out}(n) \quad \text{for all nodes } n \text{ in } g \text{ containing a call to } f$$

Iterative data-flow analysis is performed most efficiently on a postorder traversal of the program graph.

4.2 Analysis of Memory Variables

The analysis of memory accesses is more complex than the analysis of registers; for each register access the register number is known while for memory accesses the address need not be known at compile time. Thus, the main task of memory access analysis is to find the memory address accessed.

Each operation accessing memory gets attributed with information about the corresponding memory variable. This information comprises the variable's type, value, ambiguity, and an

www.manaraa.com

expression describing the value. The type of a memory variable is determined according to its declaration in the source program as: local, global, array, pointer, all; a memory access where the target cannot be determined is of type "all". A memory access may be ambiguous if several addresses may be referenced e.g. in: "if (cond) i = 3 else i = 4; a[i] = ...". In such a case the operation gets the attribute "ambiguous", otherwise "unique". The expression determining an address is part of the operation's dependence information, as well as the address' value if it can be determined. The dependence information about memory accesses is collected for nodes, too, but not interprocedurally yet. Interprocedural dependence analysis for memory variables requires alias analysis, additionally.

Rules can be determined for moving an operation accessing memory across another memory access operation depending on their attributes, i.e. if local or global variables are accessed:

local - global	can be moved except after a stack overflow.
local - local	can be moved if the addresses are different; for ambiguous accesses it has to be determined if the expressions for the addresses evaluate to disjoint sets of possible addresses.
local - array	can be moved
local - pointer	may be moved if either the local variable cannot be a pointer target or if it is one, but the pointer cannot refer to this variable.
local - all:	cannot be moved.

Similar rules apply to the other combinations of memory accessing operations.

For moving an operation accessing an array across another operation of this kind, where at least one of both writes to memory, memory reference disambiguation has to be implemented. For moving operations accessing via pointers or, moving across such operations, pointer analysis is necessary.

5 Multicycle Operations

In addition to the program analysis methods, resource usage has to be considered for instruction scheduling. The processing elements (PEs), as the most important resources, are considered by representing each cycle in the program graph.

According to the philosophy of Reduced Instruction Set Computers, most operations of a RISC instruction set need one cycle to complete the execution phase of their pipeline stage. However, there are operations in many current architectures, e.g. mul/div, load/store, and several floating-point operations, which need more than one cycle for their execution phase. Therefore, if n cycles (n>1) are required to execute an operation on processing element PE_i, we shall have to consider during scheduling and PE assignment that PE_i has to be reserved during n succeeding cycles for this operation.

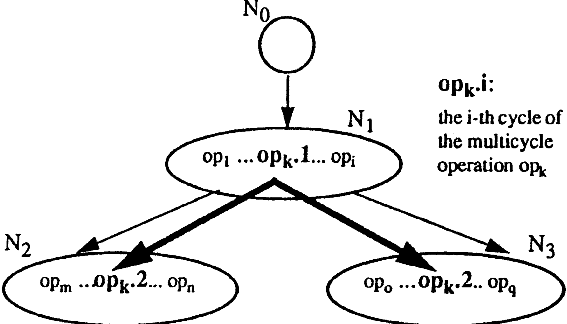
As mentioned above, Percolation Scheduling is performed on program graph nodes ([9]) containing operations being data independent and executable in parallel. So, each node represents a Very Long Instruction Word and all these operations within an instruction will be executed in one machine cycle. Using this data structure, we have to insert fill-ins, representations (called multicycle-rep) for each of those n cycles, during data-dependence analysis to

www.manaraa.com

ensure correct scheduling. All multicycle-reps get the same information about data flow (use(op), def(op)) and data dependences (reads/writes of op's register and memory locations). After assignment of a n-cycle (n>1) operation op to PE_i, the first cycle op.1 is the i-th operation within the instruction word and all n-1 following instructions will have a *noop* in the i-th place.

If there is such a n-cycle operation op in node N, there will be two possibilities for inserting representations:

1. Creating n-1 new nodes each containing one single multicycle-rep op.i ($2 \leq i \leq n$). The first cycle op.1 remains in N. If these new nodes cannot be deleted during scheduling, the new instructions generated from these nodes will enlarge the code of the target program and possibly cause lower performance.
2. To avoid creating new nodes which could become a barrier for scheduling, we decided to enter the representations in successor nodes where possible. So, op.1 remains in N and representations for the second up to the n-th cycle are entered in successors N_{succ} of N in distance d ($1 \leq d \leq n-1$) provided that there are no data dependences between operations of N_{succ} and the representation op.i. See figure 1 as an example for the insertion of a two-cycle operation op_k.



Let $op_k \in N_1$ be a two-cycle operation:
Insert op.1 for the first cycle in N_1 and representatives for the second cycle in N_1's successors in distance 1 ($\rightarrow$ in N_2, N_3)
Precondition: op_k is data independent on $op_m ... op_n$ and $op_o ... op_q$

Figure 1: Insertion of multicycle representations

To perform these insertions, data dependences between the multicycle-rep and all operations of node N_{in} in which the multicycle-rep should be entered are checked. For each data-dependent multicycle-rep a new program graph node is created containing this representative. In the following cases the creation of new nodes cannot be avoided, too:

- Node N_{in} is a leaf but there are still k representations for k further cycles ($1 \leq k \leq n-1$) to enter. This causes the creation of k new nodes containing only op.i ($n-k+1 \leq i \leq n$).
- Node N_{in} has a call (jump_and_link) operation and op.i ($i < n$) is entered in N_{in}: before executing instructions of the called function all operations started with or just before the call have to be finished. So, n-i new nodes have to be inserted into the program graph.

المنارة للاستشارات
www.manaraa.com

- If a branch or call operation itself needs more than one cycle execution time, new nodes with multicycle-reps will have to be created to ensure correct branching immediately after the last cycle.

Aggregate multicycle-reps may be used to represent a sequence of nodes containing only a multicycle representation as single operation, if there are no other operations executable in parallel to this sequence. When applying one of the core transformations move_op resp. move_cj to these aggregate nodes, they have to be expanded step by step.

6 Percolation Scheduling

Program transformations are used to reorder the operations so that a high degree of parallelism can be achieved; we use Percolation Scheduling (PS) for this purpose. The front-end of our scheduler creates the program graph where the nodes are attributed with the data-dependence and data-flow information determined according to section 4. The objects of reordering are the machine operations in the program graph nodes.

Percolation Scheduling tries to move operations in the program graph as highly up as possible, filling the nodes according to available resources, and deleting empty nodes. PS consists of several levels; we have extended and structured the definition in [9] where only level 0 is described and the other levels are just mentioned abstractly.

Level 0: This level of PS comprises a set of core transformations for moving operations between adjacent nodes, deleting nodes, and merging operations.

Level 1: Control tactics specifying how the core transformations are applied to a set of operations or nodes inside a window around a node N specified by a control strategy (level 2).

Level 2: Control strategies determining the sequence of nodes or operations to which the core transformations are applied using particular control tactics from level 1.

Level 3 up: The higher levels of PS comprise methods for higher-level constructs, e.g. methods for loop transformations such as software pipelining.

The methods are applied according to decreasing levels, i.e. starting with the highest level.

6.1 The Core Transformations

The core transformations comprise

- Move_cj for moving a conditional branch operation to a preceding node
- Move_op for moving other operations to a preceding node
- Unify for merging copies of the same operation
- Delete for deleting empty nodes.

The core transformations check all necessary data and flow dependences and insert corrective code where necessary. In our implementation the unification of operations which are copies of the same original is integrated in the move_op and move_cj transformations (for performance reasons). In addition to the methods of [9] we perform the unification of conditional branches, too, because in some program graphs a high number of unnecessary copies

www.manaraa.com

of a conditional branch may occur. Below, the move_op and the move_cj transformations are described in detail, as well as move_multicycle_op, a transformation we developed for moving multicycle operations in the program graph; this transformations comprises the unification of operations, too. The delete transformation which just deletes empty nodes and adjusts the graph's edges is not described further.

6.1.1 Move_op

Figure 2 shows the move_op transformation. This core transformation moves an operation op'_i from a node N to a preceding node M. The operation can be moved if there are no read_after_write, write_after_read or write_after_write data dependences and if op'_i does not write to a variable live at the beginning of N3 (off-live dependence). A copy N' of node N is needed if N has another predecessor besides M, e.g. N2 in figure 2.

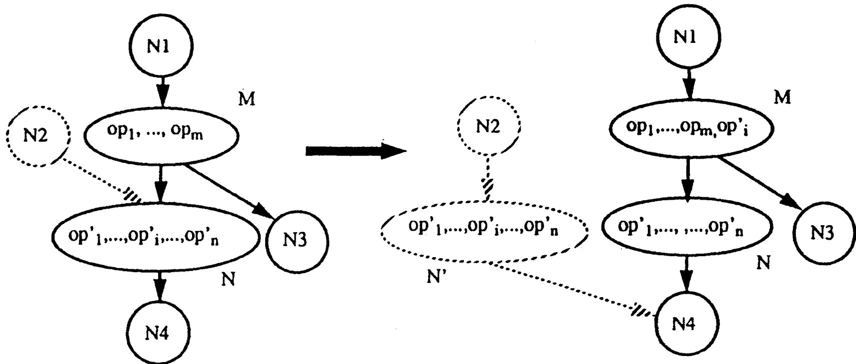


Figure 2: Move_op

6.1.2 Move_cj

For processors with multiway branches, such as VLIW or conditional-execution architectures, several conditional branches may reside in the same node. For supporting these architectures, the operations inside a program-graph node are structured in a tree. The nodes are the branch operations and the edges contain the operations on the paths emanating from a branch operation.

The data dependences checked for move_cj are only read_after_write dependences because a conditional branch does not write to a register (except the PC which is not considered here) or to memory. Figure 3 shows the trees of operations inside a program graph node and how a conditional jump cj is moved from a node N to a preceding node M. The operation cj has a left subtree T_f (building the FALSE-path) and a right subtree T_t (the TRUE- path). Node N is copied twice, to the nodes N_t and N_f. In N_t the FALSE-subtree T_f of cj is omitted together with cj and in N_f the TRUE-subtree T_t of cj is omitted together with cj. The conditional jump itself is moved to the bottom of the tree in node M and M has now N_f and N_t as successors instead of N. The node N can be deleted if it has only M as predecessor.

www.manaraa.com

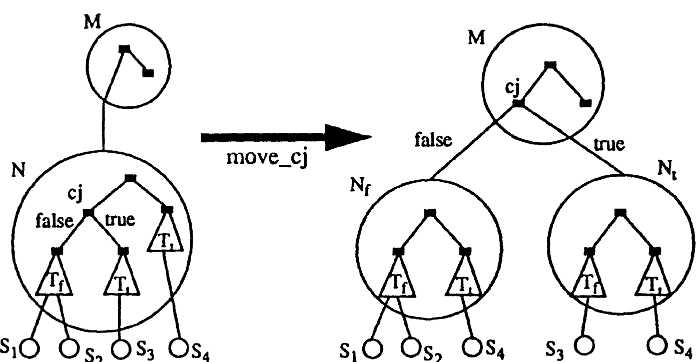


Figure 3: Move_cj

6.1.3 Move_multicycle_op

This core transformation is an extension of move_op: instead of moving a single operation from one node to its predecessor, a whole sequence of multicycle representations from op.1 to op.n is moved. See figure 4 showing an example for moving a two-cycle operation op from node N to M and notice the integrated *unify* of op.1 in N respectively in N_4. In our example, if op.1 can be displaced from N resp. N_4 to M, each multicycle-rep op.i ($2 \le i \le n$) will be moved to the predecessor node N_{i-1} containing op.i-1 before the move.

Generally, let op $\in$ N be a n-cycle operation (n>1) with representations op.3 to op.n inserted in successors of N_2, N_3, and N_5. Before moving any multicycle-rep, data- and off-live-dependences of op.1 on all operations of M have to be checked with respect to registers and variables. If there are no dependences and if there are sufficient resources to execute op.1 in parallel to all operations of M, move the whole sequence op.1 to op.n in the following manner:

- Preservation of the semantic correctness:
  If N has not only M as predecessor ($\exists\, N_{pred} \neq M$):
  Copy the whole multicycle-tree (i.e. all nodes containing multicycle-reps op.i, $1 \le i \le n$) with all its edges and adjust the edges from all predecessors N_{pred}, $N_{pred} \neq M$, to N: from N_{pred} to the root of the multicycle-tree copy. In figure 4, the tree with nodes N, N_2, N_3 is copied, and the edge from N1 to N is adjusted from N1 to N'.
- Treatment of M:
  Take all op.1 which are in successor nodes of M - these copies result from previous applications of *move_multicycle_op* - and insert one multicycle-rep op.1 in M - corresponding to the core transformation *unify*. In figure 4., op.1 in N and in N_4 are unified and moved to M.
  Update data-dependence and liveness information of M and allocate resources for executing op.1 in M.

المنارة للاستشارات
www.manaraa.com

- Treatment of all multicycle-reps from op.2 to op.n; $\forall$ i, $2 \le i \le n$: N_i contains op.i:
  If the predecessor node N_{i-1} containing op.i-1 (for i=2, N_{i-1} = N) has only one successor N_i, move op.i to N_{i-1}. In our example, N_4 has only the successor N_5; therefore, move op.2 from N_5 to N_4.
  Otherwise, if N_{i-1} has two or more successors, remove op.i in each N_i and insert one multicycle-rep op.i in N_{i-1}. Dependence, liveness and resource checks are not necessary here, because they were already done when the representations op.i were inserted for the first time. In figure 4, N has two successors, N_2 and N_3, each containing op.2. Remove op.2 from N_2 and N_3, and enter a single op.2 into N.

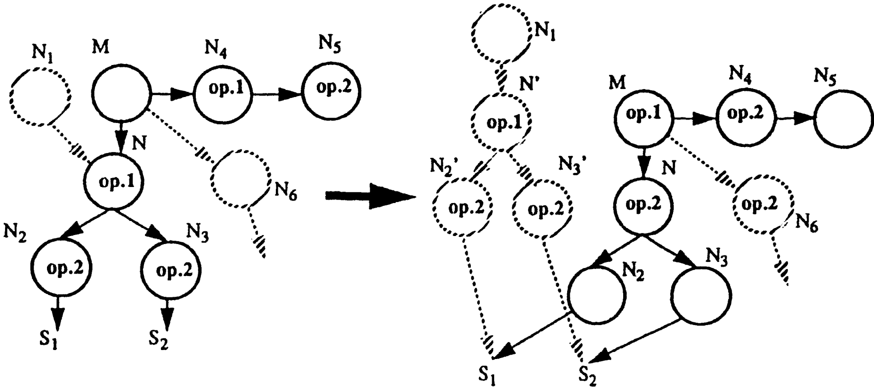


Figure 4: Move_multicycle_op

- Treatment of all nodes N_n containing op.n originally:
  After op.n has been moved to pred(N_n), update data-dependence and liveness information of N_n and release all resources which were reserved for op.n in N_n; in our example, do this for N_2, N_3, and N_5.
- Preservation of the semantic correctness after moving:
  If there are successors of M which do not contain op.1, e.g. N_6 in figure 4.:
  Try to insert multicycle-reps in distance d from M ($1 \le d \le n-1$), as described in section 5. In our example, insert op.2 in N_6.

If node N has more than one predecessor M, we shall have to copy entire multicycle-trees which can increase the number of program graph nodes significantly. Therefore, the control strategy may restrict the application of *move_multicycle_op*: the multicycle operation will only be moved, if op.1 can be inserted in all predecessors of N.

The decision whether a multicycle operation should be moved belongs to the problems solved by the control strategies described below.

www.manaraa.com

7 Controlling Mechanisms

As mentioned above, the PS core transformations allow using various control mechanisms. Therefore, our task is to develop control strategies applying core transformations in a suitable order to create nearly optimal schedules.

In our first version we do not provide moves across loop boundaries. To prevent those moves we define regions of the program graph, separated by loop boundaries, and allow movements only within regions. Hence the regions, control strategies are applied to, are acyclic subgraphs of the program graph.

We distinguish between two basic approaches of control, the node-oriented and the operation-oriented approach. Both approaches consider layer one and two of PS. Layer one contains a control tactics scheme describing local moving rules applied to a node or operation. Layer two contains the actual control strategy specifying the sequence how control tactics are applied to nodes or operations.

Scheduling all movable operations is a task of high complexity and the quantity of core transformation applications (up to many tenthousands) may exceed acceptable compiler runtime. Therefore, we provide windows of arbitrary size to limit the range of the control tactics. Both, window size and number of passes for application of control strategies can be varied.

Some of the control strategies and control tactics are introduced below.

7.1 Control Strategies

Control strategies (layer 2) determine the sequence of nodes to be considered for scheduling. Such a sequence may be chosen as:

- *path-oriented*: Prioritizing scheduling along critical paths reduces the number of cases where schedule lengths are unnecessarily increased by early placement of uncritical operations. The "critical path" can be chosen as:

  - simply the longest path along nodes of a program graph or
  - the most probable path along nodes or
  - the longest chain of data-dependent operations; a path is built by the nodes containing these operations

  The last kind of path corresponds to the operation-oriented approach, the previous kinds to the node-oriented approach. When the nodes' execution probabilities are determined by clever heuristics or profiling, taking the most probable path seems to be most promising. Otherwise, the longest path along data-dependent operations is a preferable choice.

- *bottom up*: Nodes are visited upwards from the region's leaves to its top node. Candidates for current bottoms are nodes whose successors are either

  - already visited (resp. filled) or
  - not in the same region or
  - only reachable via a back edge.

  The bottom node may be determined with

  - left-downward DFS (depth-first search),

www.manaraa.com

- right-downward DFS or
- critical-path-downward DFS

In the last method, "critical path" can be viewed as longest path or as most probable path through nodes. The advantage of this method with respect to the pure path-oriented strategy is a higher probability for the application of the core transformation unify, because the successors of the current nodes are guaranteed to be treated before them. Thus, an operation and its copy have likely reached the current node's dominator on their way up through "diamond structures" and are reunifiable.

- *top down:* Nodes are visited downwards from the region's top node to its leaves. The traversal may be performed analogously by several kinds of DFS.

Several passes of the strategies may be applied, even including different strategies. The bottom-up strategy supports moves of data-independent operations up to the region's top within one pass. In the top-down strategy, the movement of those operations is restricted by the window size of the control tactics. However, top down is able to move long chains of data-dependent operations, while in bottom up the chain length is restricted by the window size of the control tactics.

7.2 Control Tactics

The control tactics layer lies between the strategy layer and the core-transformation layer. The tactics specify rules how to apply core transformations within a window attached to a given node N or operation op. This node N or operation op are specified by the control strategy, as well as the sequence of control tactics applied.

7.2.1 Node-Oriented Tactics

Node-oriented tactics are for instance pull_n or push_n (see figure 5).

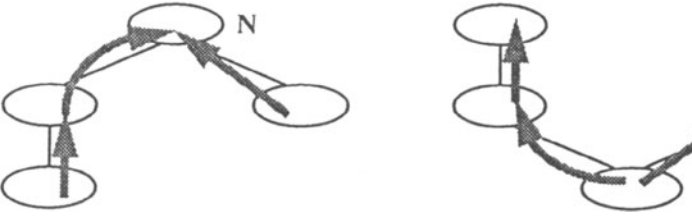


Figure 5: pull with window size 2 **push with window size 2**

- *Pull_n* performs moves in a window reaching from the strategy-selected node N to all successors of N within a distance of n. It tries to move all operations up to the window's top node N. Pull_n may be realized in a way so that all the operations movable to N are determined and the optimal one is chosen as proposed for the VLIW-Scheduler of IBM at Yorktown Heights (see [2]).

www.manaraa.com

- *Push_n* works in the opposite manner. The window includes the strategy-selected node N and all predecessors with a distance not greater than n to N. These tactics try to move all operations from N within the window as highly up as possible.

While data dependences have to be checked between op and each node on its way up, resource constraints have to be checked only between op and its destination node. In this manner, operations can be passed even through nodes which are full with respect to resources.

Pull_n is a suitable method to fill nodes with operations under resource constraints. Push_n aims at emptying nodes. Thus it reduces compiler runtime by early deletion of nodes.

7.2.2 Operation-Oriented Tactics

Some examples for operation-oriented tactics are shown in figure 6.

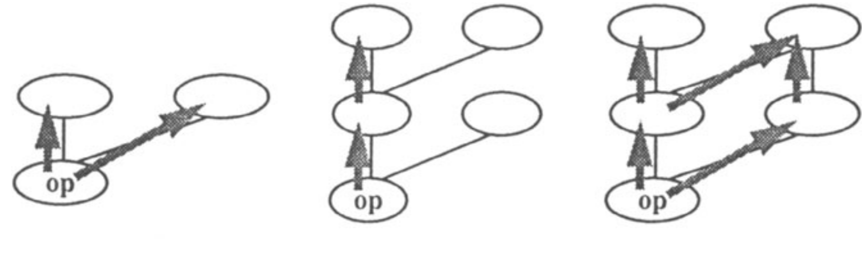


Figure 6: move_to_all_preds migrate_within_trace migrate

- *Move_to_all_preds* moves an operation op of node N simultaneously to all predecessors of N. If it can be performed, a copy node is not required.
- *Migrate_within_trace* moves an operation op along a trace uppermost in the region. The different kinds of critical paths mentioned above are considered as traces.
- *Migrate* takes an operation op and moves it upmost in the region. These tactics are used in Compact_Global ([1]) and in [2].

For *migrate* and *migrate_within_trace* we may specify a window to restrict moving distances to a predefined length.

Additional tactics may be considered for particular patterns in the program graph, e.g. for diamond structures or chains of if-then-constructs (stairs). In the case of diamond structures a node M contains a conditional jump and both paths emanating from M rejoin later in a succeeding node N, typical for "if-then-else" constructs. Moving an operation op upward from N causes the creation of one (or more) copy node(s). We try to move op directly to M without actually inserting it in the intermediate nodes in the diamond structure. Thus, the number of copy nodes is reduced. Blocking in intermediate nodes due to resource shortage can be avoided, too.

المنارة للاستشارات

www.manaraa.com

8 Conclusion

Fine-grained parallelism is offered by most new microprocessors. The methods and tools described above offer the way to exploit this parallelism so that it can be used for application software. Interprocedural program analysis and global instruction scheduling methods described above are the base for this exploitation. Percolation Scheduling has been enhanced by representations and transformations for multicycle operations and by scheduling tactics and control strategies directing how to apply Percolation Scheduling's core transformations. With these new scheduling tactics and control algorithms we have specified methods for the new levels 1 and 2 of Percolation Scheduling.

Parallelization methods for the upper levels of Percolation Scheduling, mainly for loop handling will further enhance performance. This is one of the areas where methods for coarse- and fine-grained parallelization complement each other and lead to higher performance, by a combination of coarse- and fine-grained loop parallelization. There and in other areas, both methods can complement each other and lead to higher system performance.

9 References

[1] A.Aiken:
Compaction-Based Parallelization
PhD Diss., Techn. Report No. TR88-922, Cornell University, Ithaca, NY, June 1988

[2] K.Ebcioglu, A.Nicolau:
A Global Resource-constrained Parallelization Technique
Proc. 3rd Int. Conf. on Supercomputing, Crete, June 1989, pp. 154-163

[3] J.R.Ellis:
Bulldog: A Compiler for VLIW Architectures
MIT Press, 1985

[4] J.A.Fisher:
Trace Scheduling: A Technique for Global Microcode Compaction
IEEE Transaction on Computers, July 1981, pp. 478-490

[5] R.Gupta:
A Reconfigurable LIW Architecture and its Compiler
PhD Dissertation, University of Pittsburgh, 1987, Order No. 8807357

[6] N.P.Jouppi:
The Nonuniform Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance
IEEE Trans. on Comp. Vol. 38, No. 12, Dec. 1989

[7] J. Lah, D.E.Atkins:
Tree Compaction of Microprograms
ACM SIGMICRO 16th Annual Workshop, pp.23-33, Oct. 1983

www.manaraa.com

[8] J.L. Linn:
SRDAG Compaction - A Generalization of Trace Scheduling to Increase the Use of Global Context Information
ACM SIGMICRO 16th Annual Workshop, pp.11-22, Oct. 1983

[9] A.Nicolau:
Percolation Scheduling: A Parallel Compilation Technique
Technical Report TR 85-678, Cornell University, Ithaca, NY, May 1985

[10] C. Störmann, G. Piepenbrock:
Erzeugung parallelen Codes für VLIW-Architekturen durch globale Kompaktierung
Diplomarbeit, TU München, May 1990

[11] B.Su, S.Ding, L.Jin:
An Improvement of Trace Scheduling for Global Microcode Compaction
ACM SIGMICRO 17th Annual Workshop, pp.78-85, Oct. 1984

www.manaraa.com

Causality Based Proof of a Distributed Shared Memory System

Dominik Gomm   Ekkart Kindler

Technische Universität München
Institut für Informatik
Arcisstr. 21
W-8000 München 2
Germany

1 Introduction

The specification and verification of distributed systems calls for techniques tuned to the particular area of application. In this paper we introduce a specification and verification technique which exploits the order (causality) in which different events of distributed systems must occur. The technique is illustrated by applying it to a distributed shared memory system (DSM-system). We model a DSM-system by means of Petri net protocols for a DSM-system and prove that the executions of the protocols respect the specified ordering of events.
During recent years DSM-systems have gained great importance, because

- on the one hand it is easy to build hardware for distributed systems with physically *distributed memory*,
- on the other hand the concept of a *shared memory* is easier to understand when developing parallel programs.

In order to combine the advantages of both concepts, DSM-systems in a virtual way provide a shared memory on the basis of a physically distributed memory [7, 1]. Similar problems have for long been investigated in the fields of cache coherence protocols [11] and distributed database management systems [2]. In [8] it is investigated in which way DSM-concepts can be incorporated into concurrency control of distributed database management systems.
There are many different approaches to build efficient DSM-systems. An overview on these approaches is given in [6]. In this paper we follow the *caching* approach of [3]: Variables are assigned to pages which are located on different nodes (page owners). In order to reduce expensive accesses to variables which are located at different nodes, there may be *copies* of the *original* page at other nodes. When there are several copies of the same variable, the consistence of the copies must be guaranteed. To formalize the notion of consistence, *data coherence concepts* have been introduced. [3] employs the concept of *weak coherence* [1].

www.manaraa.com

The paper is organized as follows: Section 2 formalizes the concept of weak coherence within the framework of causalities. In Sect. 3 we introduce the Petri net protocols as an abstract operational model of the DSM-system in [3]. In Sect. 4 we apply our methods to prove its correctness with respect to the specification.

2 Weak Coherence

In this section we introduce the basic notions which are necessary for a formal treatment of weak coherence.

2.1 Control Flow and Read Relation

From an abstract point of view an execution of a distributed program is a partial order of events, which is called *control flow*. The control flow describes which events must occur *causally* before others. Typically, there are a number of (sequential) threads interacting with each other by communication events as shown in Fig. 1. Some events

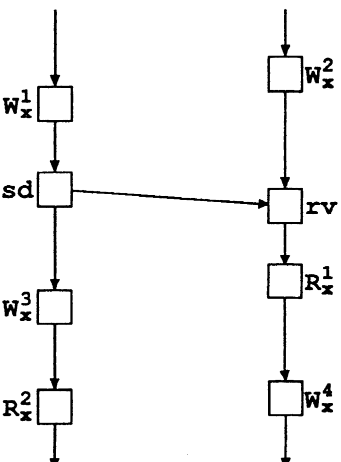


Figure 1: Example of an execution

are ordered by the control flow (e.g. W_x^1 and R_x^1). But, if the relative speed of the two threads is unknown, there will be pairs of events (of different threads) for which no order can be fixed. For example consider R_x^1 and W_x^3 in Fig. 1. Events which are not ordered by the control flow are called *concurrent*.

Notation 1 If two events x and y are ordered by the control flow this is denoted by $x \rightarrow y$. Graphically we represent the control flow as in Fig. 1. Note that $\rightarrow$ is transitive, and thus transitive arcs can be omitted in the graphical representation. Sometimes we indicate the transitive relationship by dashed arcs.

www.manaraa.com

In the context of memory coherence we are only interested in the occurrence of read and write events (denoted by R_x and W_x, respectively, for accesses to variable x), and in the occurrence of communication events which establish a control flow between different threads. In this paper we use asynchronous send and receive events (denoted by sd and rv, respectively). Additionally, we have to know which values are written and read by the different write and read events. Since a read event returns the value written by exactly one write event[1], this information can be formalized by a relation on write and read events, without mentioning values at all.

Notation 2 When a read event R_x reads the value written by write event W_x, we write $W_x \leadsto R_x$ and call this relation *read relation.* The read relation satisfies the following conditions:

1. For every read event R_x there exists exactly one write event W_x such that $W_x \leadsto R_x$.
2. For every two events W_x and R_x, $R_x \rightarrow W_x$ implies $W_x \not\leadsto R_x$.

Informally, condition 1 says that a read event returns exactly one value; condition 2 says that no read event returns a value which will be written in its future.

Graphically the read relation is represented by thick arrows as shown below:

An *execution* of a distributed program consists of a control flow and a corresponding read relation.

2.2 Definition of Weak Coherence

For the control flow shown in Fig. 1 a read relation from W_x^1 to R_x^2 would indicate that R_x^2 reads the obsolete value written by W_x^1. W_x^1 is obsolete w.r.t. R_x^2, because there is another write event (W_x^3) that occurs causally between W_x^1 and R_x^2.
In order to rule out such executions, we formalize the concept of weak coherence:

Definition 3 (Weak Coherence)

An execution is *weakly coherent* iff for every W_x, W'_x, and R_x, $W_x \rightarrow W'_x \rightarrow R_x$ implies $W_x \not\leadsto R_x$.

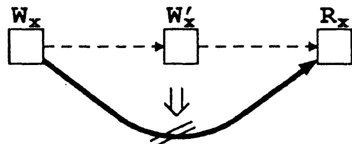


Figure 2: Definition of weak coherence

Figure 2 shows a graphical representation of this definition. In combination with the properties of the read relation this characterization of weak coherence is equivalent

[1]Assume that there is an initial event which writes an undefined value to variable x.

www.manaraa.com

to the definition given in [1], and covers the list of requirements for a specification of weak coherence in [3].

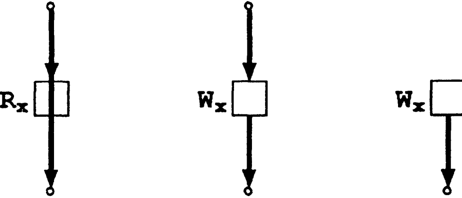
The control flow of Fig. 1 becomes a weakly coherent execution iff R_x^1 reads the value written by W_x^1, W_x^2, or W_x^3, and R_x^2 reads the values of W_x^2, W_x^3, or W_x^4. This shows that in addition to the values of the immediately preceding write events, Def. 3 allows that a read event reads the value of a concurrent write operation.

2.3 Physical Realization of Read Relation

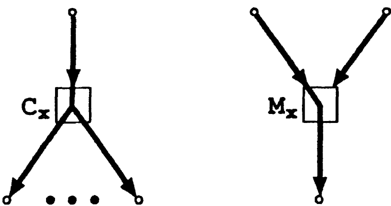
The read relation $W_x \rightsquigarrow R_x$ of an execution is only an abstract relation which is not justified by the physical propagation of data. In real systems the read relation must be realized by a combination of ***data propagation events.*** In addition to read and write events of the execution we allow events for making copies of variables, and for combining the values of different write events to the same variable. The following definition shows in which way these operations establish the read relation.

Definition 4 (Propagation Events)

A read event does not change the value of a variable. This is captured by the fact that the read relation 'passes through' a read event as shown in the graphical representation below. A write event, however, changes the value of variable x, and therefore the read relation does not 'pass through' a write event.



In addition to the events which belong to a program's execution there are propagation events which are part of the hardware or software environment in which the program is executed. There is a copy operation C_x which produces copies of variable x



and there is a merge operation M_x which combines the value of two write operations, i.e. which propagates the value of one and overwrites the value of the other.

The arcs of these propagation events correspond to real data paths. Therefore, when combining propagation events at the points $\rightarrow\!\circ$ $\circ\!\rightarrow$, these paths refine the read relation of an execution. In Fig. 3 R_x^1 and R_x^2 read the value of W_x^1, i.e. $W_x^1 \rightsquigarrow R_x^1$ and $W_x^1 \rightsquigarrow R_x^2$. R_x^3 reads the value of W_x^3, but not of W_x^2 or W_x^1.

www.manaraa.com

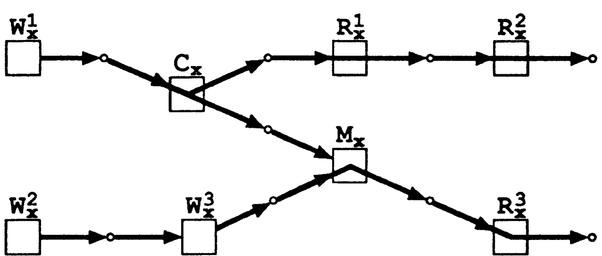


Figure 3: Example of linked propagation operations

2.4 Control Flow, Information Flow, and Weak Coherence

A weakly coherent system could be implemented according to the previous definitions using the propagation events to establish a weakly coherent execution. However, Fig. 3 shows that the read relation is not transitive at write and merge events. Thus, the read relation is not a causality. Since causality based specifications are easier to be proven correct[2], we will give another characterization of weak coherence which is based on the transitive closure of the read relation. We call this transitive (and reflexive) closure *information flow* and denote it by $\leadsto^*$. In graphical representations we use dashed thick lines to indicate information flow:

W_x ☐ - - - - - - - - - -▸ ☐ R_x

In the following we present two properties which are essentially based on the different kinds of causalities, and which imply weak coherence. The first property guarantees that according to the flow of information every read operation R_x has 'knowledge' of all write operations W_x which precede R_x according to control flow.

Definition 5 (Information Availability)

An execution in which the read relation is refined by propagation operations (*refined execution* for short) satisfies the *information availability* property iff for every read operation R_x and every write operation W_x holds: $W_x \rightarrow R_x$ implies $W_x \leadsto^* R_x$.

Figure 4 shows a graphical representation of this definition.

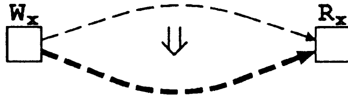


Figure 4: Information availability

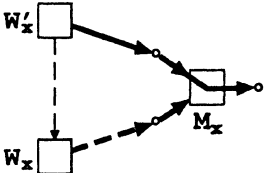
In order to guarantee that information availability implies weak coherence we need an additional property which requires that only correct values (i.e. newer ones according to control flow) are propagated by the merge events. This is formalized by the following property.

[2]see Section 4

www.manaraa.com

Definition 6 (Correct Merge)

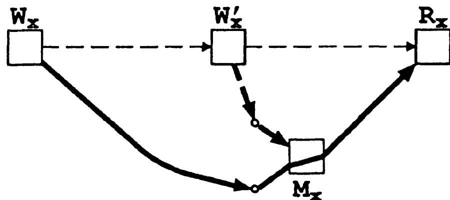
A refined execution satisfies the *correct merge* property iff there are no write events W_x and W'_x and no merge event M_x such that the following relations hold:



The combination of the properties of Def. 5 and 6 is a sufficient condition for weak coherence.

Proposition 7 Every refined execution which satisfies the correct merge property and the information availability property is weakly coherent.

Proof: Suppose the execution satisfies information availability, but is not weakly coherent. Then according to Def. 3 there exist W_x, W'_x and R_x such that $W_x \rightarrow W'_x \rightarrow R_x$ and $W_x \rightsquigarrow R_x$. By Def. 5 and 4 there exists a merge event M_x such that the following relations hold:



Then the correct merge property does not hold. □

2.5 Page Concept

Implementing weak coherence by a caching concept is only efficient when variables are collected in pages. The following considerations show that the above definitions can be easily adapted to the page concept.

The write event W_y^p to a variable y on page p comprises of a collection of events: There is one write event on variable y, and there are copy events for all other variables (see

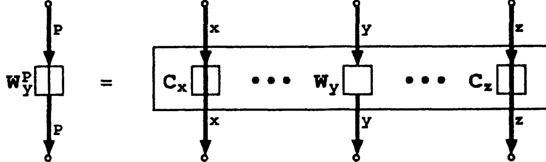


Figure 5: Write event on variable y of page p

Fig. 5). Similarly, we restrict the merge operation such that a merge event on page level corresponds to a merge event on a single variable (see Fig. 6). A read event

www.manaraa.com

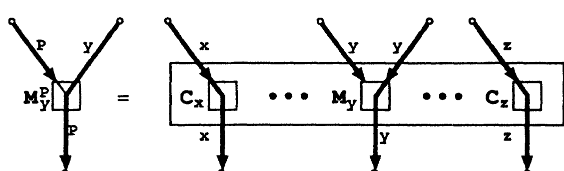


Figure 6: Merge event on variable y of page p

on page level can be modelled as a collection of copy events and one read event on variable level. Copy events on page level comprise to a collection of copy events.

If there is an information flow on page level, then there is an information flow w.r.t. every individual variable of that page. This can be used in proving the information availability property on page level. For proving the correct merge property for a merge event M_y^p we must consider only the dataflow of variable y. For the other variables x of page p M_x^p is only a copy operation for which nothing must be proved.

3 Model of a Weakly Coherent System

In this section we present a model of a *weakly coherent system.* Since we only introduced weakly coherent executions, we will briefly mention weakly coherent systems in the following.

3.1 Weakly Coherent Systems

A weakly coherent system should execute distributed programs such that the corresponding program executions are weakly coherent. We do not want to formalize the notion of executing a program by a system here, because this would be tedious. Rather, we simulate the execution of an arbitrary distributed program by nondeterministically choosing write, read, and synchronization operations for each thread of the distributed program.

3.2 Modelling the System

The system is described by means of protocols which determine the behaviour of the operations mentioned above. The protocols are formally represented by Petri nets[3] for the following reasons:

- Petri nets have a partial order semantics which explicitly expresses causality and concurrency.
- When using Petri nets, not only events but also states can be modelled. States are needed to model the state of objects like pages, messages or copies.
- Petri nets have expressive analysis techniques, some of which will be used in the proof.

[3]For a formal introduction to Petri nets we refer to [9] and [10].

www.manaraa.com

Before introducing the protocols we describe the algorithm which guarantees weak coherence. We assume that each sequential thread of a program is mapped on a node and that the memory is partitioned into pages which are distributed over the nodes, i.e. each *original page* is located at exactly one node, called its *owner*. Each variable can be accessed by an address $((n, v), o)$ with node number n, page number $p = (n, v)$ and an offset o for the relative address of x on page p.
Because we follow the caching approach, there will be *copies* of original pages to speed up subsequent accesses to pages which are owned by other nodes. When copies become obsolete because of synchronization events, these copies should not be read again. Therefore, before a send operation of a thread is executed, the system sends invalidation messages to all possible holders of obsolete copies. If a node receives such an invalidation it destroys the corresponding copy.
This algorithm is given in four protocols, one for each operation a thread may perform: *read, write, send*, and *receive*, called *user operations*. Usually, Petri nets do not distinguish different causalities. Figure 7.1 shows the representation of the usual arcs in Petri nets. The arcs representing control flow and the information flow are indicated by the arc types shown in Fig. 7.2.

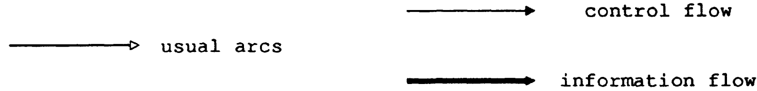


Figure 7.1: Usual arcs in the protocol

Figure 7.2: Specially marked arcs

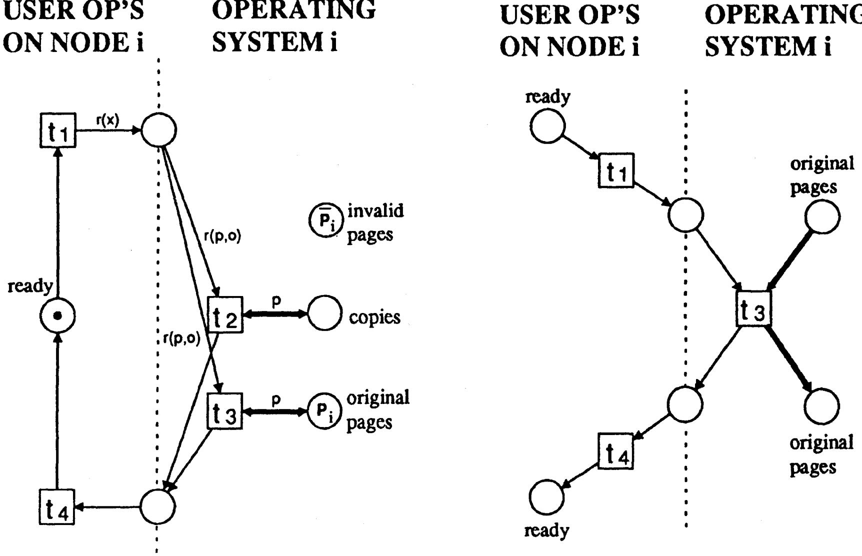


Figure 8.1: Protocol for reading I

Figure 8.2: A run of the protocol

The first protocol for reading a value is split into two figures for the sake of comprehensibility. Figure 8.1 shows how to read from a page in the local memory. Initially,

www.manaraa.com

a thread is *ready* and may perform an operation. Each node has some original pages P_i, all other pages $\overline{P_i}$ are invalid. If a thread on a node i wants to read a value x, it calls the local operating system (t_1) which reads the corresponding value from page p if p is available as a copy (t_2) or as original page (t_3). After that, control is given back to the thread which may then perform the next user operation (t_4). The behaviour of a protocol is described by its runs. Figure 8.2 depicts a run reading from an original page.

If neither a copy nor the original of page (n, v) is accessable, the node requests a copy of the owner n of the original page and waits until it receives this copy. Then it performs the read operation and stores the copy for future read events as shown in Fig. 9.

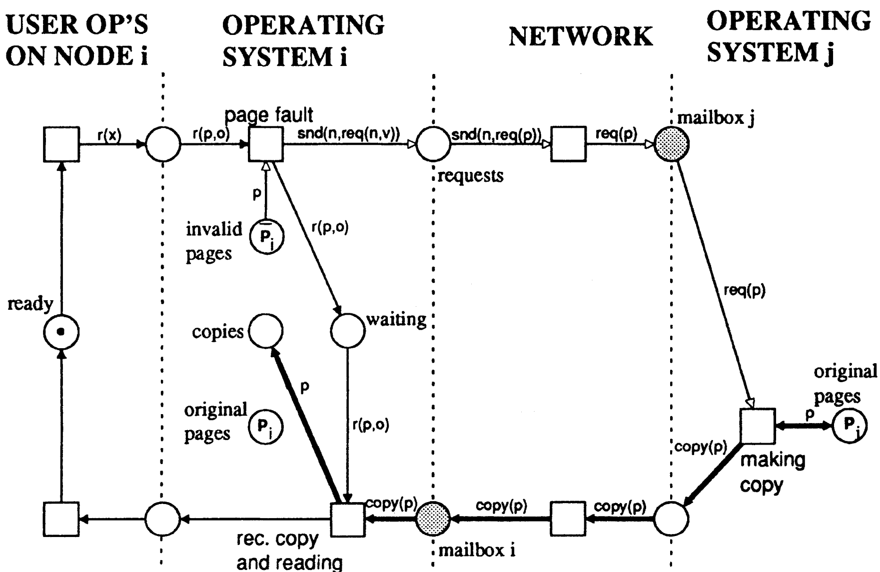


Figure 9: Protocol for reading II

The protocol for writing a value x to a page p is given in Fig. 10. Each write access[4] to a variable $x = (p, o)$ modifies the original page p, either locally by writing directly to it, or by sending an update $up(p, o)$ to the owner n of p. If there is a local copy of page p, it will be modified as well. When the update is registered in the mailbox of the owner, the thread may proceed.

Thus, all the original pages are continously updated, but copies only contain local modifications and may become obsolete with respect to the originals. Therefore, copies are invalidated whenever threads synchronize. This is specified by the protocol in Fig. 11.

[4]Actually, we do not model the change of a value, since by the read relation we only need to know which read operations read the written value.

www.manaraa.com

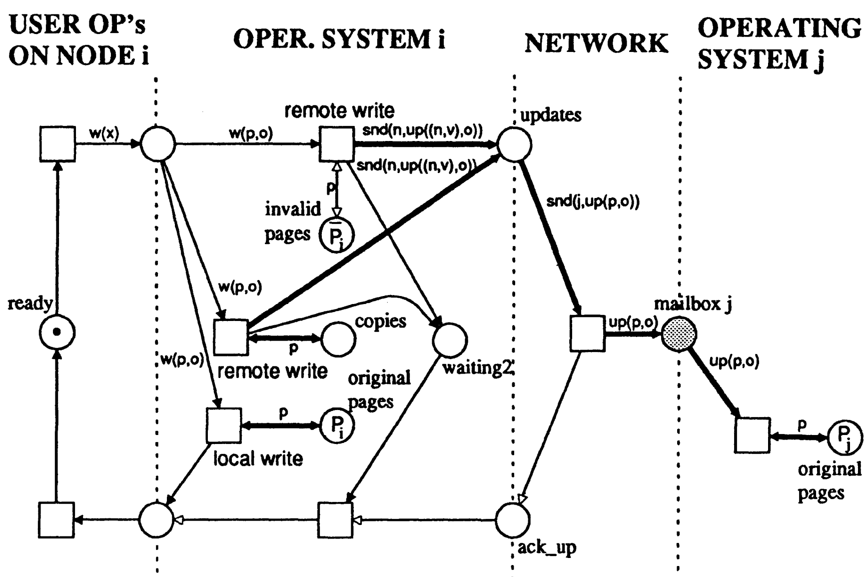


Figure 10: Protocol for writing

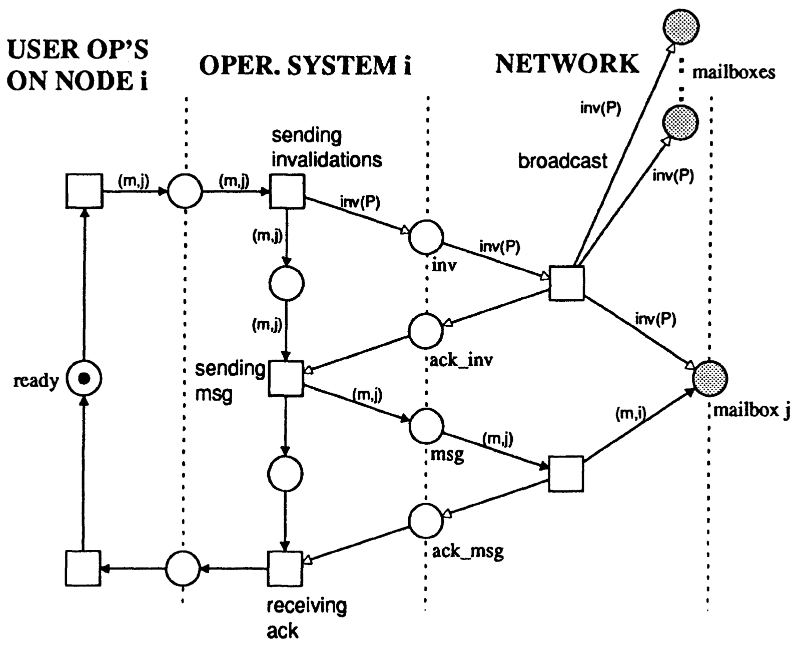


Figure 11: Protocol for the sending of synchronization messages

www.manaraa.com

So-called *invalidation messages* for a set of pages P are broadcasted to all other nodes causally before sending a synchronization message. P contains all the pages that were updated on node i since the last synchronization message. The effect of the protocol is that a synchronization message can only be sent to node j provided that all invalidations are written to the other nodes' mailboxes. In order to ensure that invalidation messages are processed prior to the corresponding synchronization message, mailboxes are defined as FIFO buffers. As a shorthand for FIFO buffers we use shaded places.

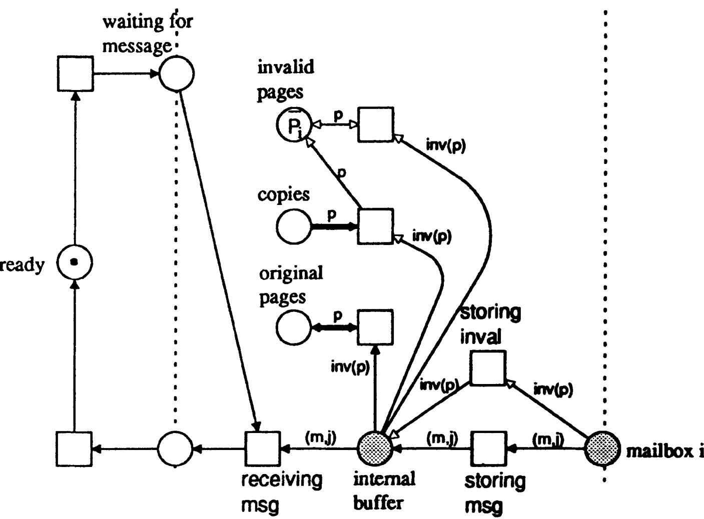


Figure 12: Protocol for the receipt of synchronization messages

Figure 12 shows the protocol for the receipt of a synchronization message m from another node j and the reaction on invalidation messages $inv(P)$: *copies* of pages $p \in P$ are set *invalid*. Both message types are stored in an internal buffer to prevent the system from deadlocks of the following kind. Assume that ahead of the mailbox there is a synchronization message from another node j, but the operating system expects a copy for reading (as shown in Fig. 9). This copy can never be received because of the FIFO property of the mailbox. Thus, messages are shifted into a buffer, and therefore, invalidations are shifted as well to retain the order between both message types.

4 Proving the Weakly Coherent System

In order to prove that the modelled system is weakly coherent we show that it satisfies Def. 5 and 6. Thus, the correctness proof is in two parts. Subsection 4.1 shows that the protocols satisfy the information availability property, in Subsect. 4.2 we prove the correct merge property. Both proofs are based on the partially ordered runs of the protocols.

www.manaraa.com

4.1 Proving the Information Availability Requirement

In order to prove the information availability requirement from Def. 5, all combinations of *write* accesses preceding *read* accesses must be considered. They are completely listed in [4]. Here, we only prove one case, because all the cases can be proved by the same technique.

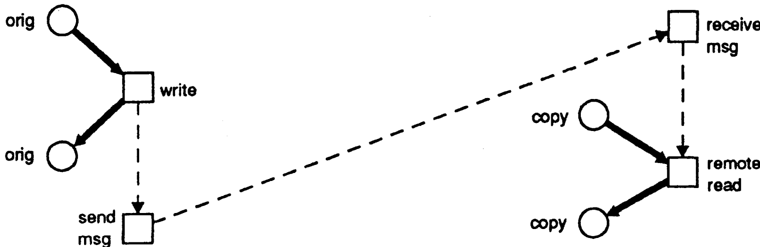


Figure 13: Premise: *write* precedes *read* by message passing

We consider the situation of a local write operation W to a variable x on a page p that precedes a remote read operation R to the same variable of another thread as given in Fig. 13. Using the following arguments we will show that there is an information flow from W to R.

1. The control flow $W \rightarrow R$ given by premise is refined according to the send and receive protocols. W and R are not yet related by the information flow in this run (Fig. 15).

2. In a second step we deduce an order between events taking messages out of mailboxes. To this end, we use the following properties of shaded places (mailboxes) in a run:
   a) Input accesses are totally ordered.
   b) Output accesses respect the order of the input accesses.
   This is graphically depicted in Fig. 14.

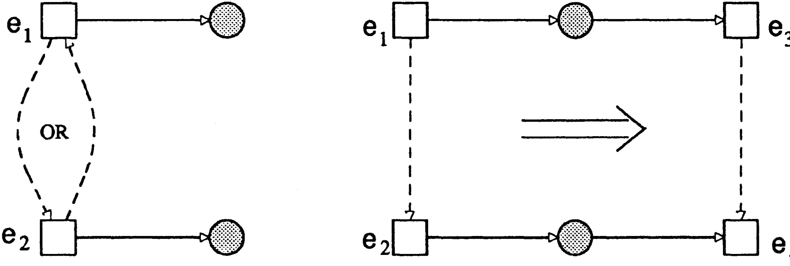


Figure 14: Semantic definition of mailboxes

3. Different occurrences of the same places in the run are set into causal relation by using *regular place invariants.* A regular place invariant is a set of places in

www.manaraa.com

the protocols such that the sum of tokens on all these places is always 1. Its places cannot be concurrent in a run, because they cannot be marked at the same time. By that, we can conclude that any two occurences of such places are causally ordered in one or the other direction.

4. Because of the acyclicity of partially ordered runs we can rule out one of these two possible orders such that an information flow from the *write* to the *read* event can be deduced.

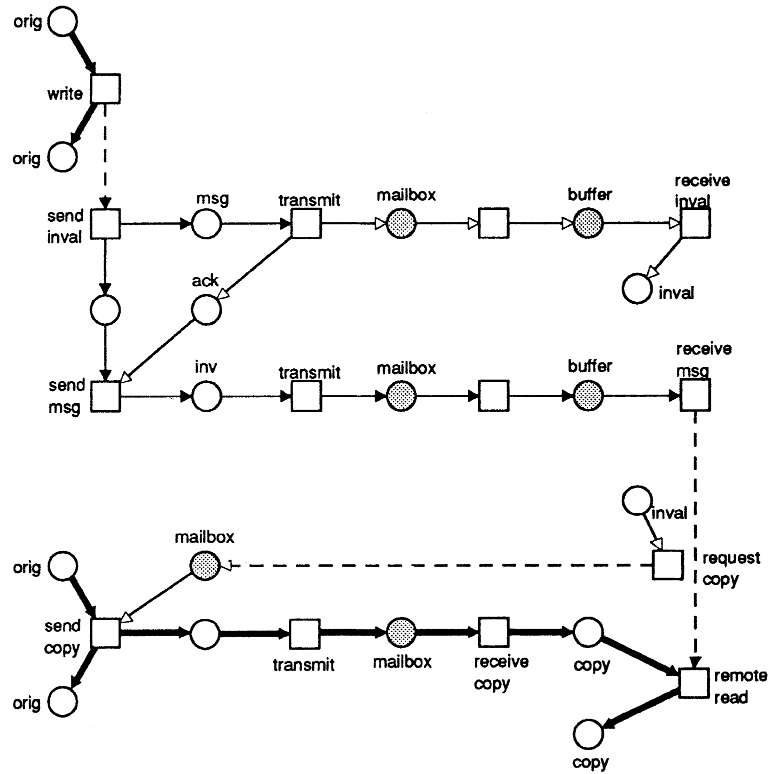


Figure 15: Deterministic extension of the run

Proof of the run in Fig. 13 We deterministically extend the run depicted in 13 as fixed by the protocols. This extension is shown in Fig. 15. Prior to the transmission of the synchronization message an invalidation message (concerning page p and others) is sent. Both messages are sent to the mailbox of the reading node and shifted into the buffer. The read operation is extended by a copy request cf. the protocol in Fig. 9, because initially there are no copies which can be read.

By the following arguments we can prove that this run satisfies the information availability requirement. This is depicted in Fig. 16. Since e_1 occurs causally before e_2 and

المنارة للاستشارات

www.manaraa.com

mailboxes and buffers respect the FIFO order of messages, the invalidation is received causally before the synchronization message (1).

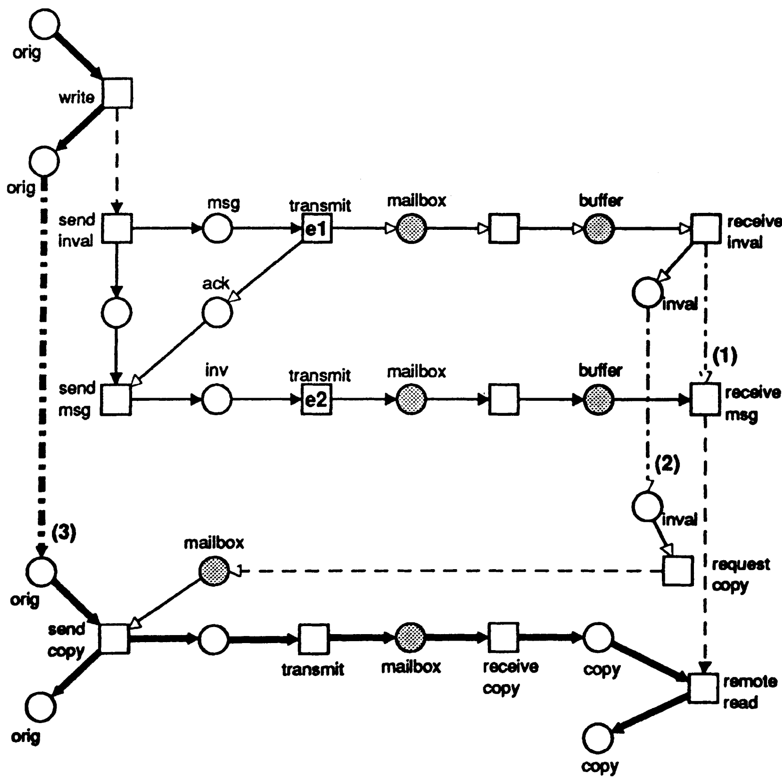


Figure 16: Proof of the information availability

Figure 17.1 shows a regular place invariant I for each page on a node which is not owned by that node: it is either invalid or there is a copy or the node is waiting for a copy. By that, we know that the occurrences of *inval* and *copy* on the reading node are causally ordered. This causality cannot order *copy* prior to *inval* because this would establish a cycle in the partially ordered run. Thus, they are ordered as depicted by (2).

Using the very same argument we can prove that information flows transitively from the *write* to the *read* event. By the regular place invariant J in Fig. 17.2 we know that the occurrences of *orig* on the writing node are totally ordered. This order is indicated by (3). Again, assuming this order in the opposite direction (from the lowest to the highest of the four occurrences of *orig*) would yield a cycle. Because all arcs of the place invariant correspond to information flow, causality (3) constitutes an information flow from the *write* to the *read* event. Thus, the information availability requirement holds.

www.manaraa.com

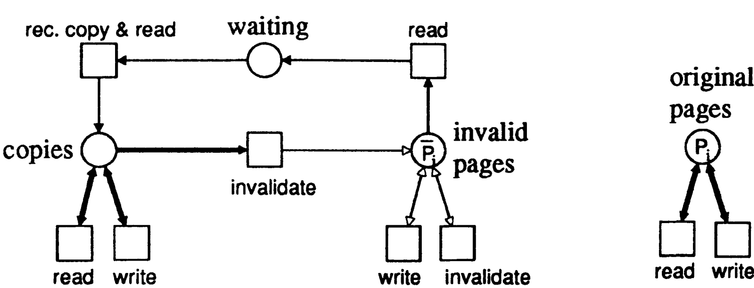


Figure 17.1: Regular place invariant I   Figure 17.2: Regular place invariant J

4.2 Proving the Correct Merge Property

The correct merge property can be shown by analogous arguments. Due to the lack of space we only give a sketch of that proof. Figure 18 shows the refined premise of that property: A remote write event W_x occurs causally before another remote write event W'_x. Both updates are merged in the original page by M_x^p and $M_x'^p$. We will prove that M_x^p occurs not causally after $M_x'^p$ w.r.t. information flow. This is sufficient, because a new value overwrites the old one from the page as specified in Fig. 6.

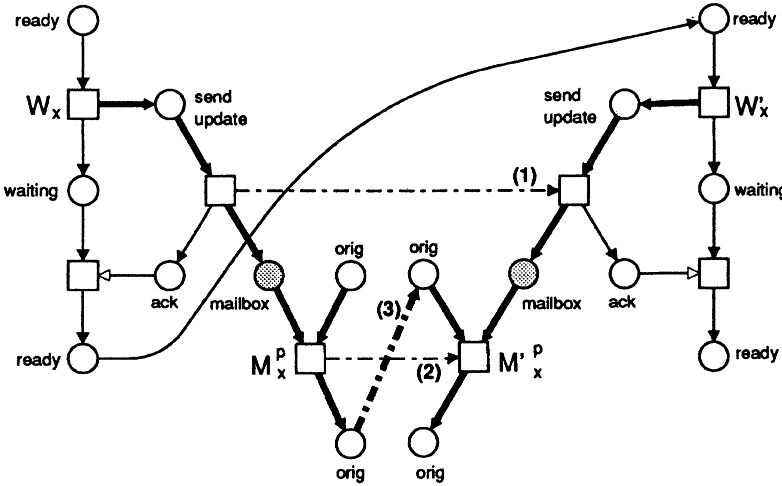


Figure 18: Proof of the correct merge property

The acknowledgement forces W_x to be written to the owner's mailbox prior to W'_x (1). Because of the FIFO property of the mailbox, the merge events are ordered as depicted by (2). From that we know that the causality between the occurrences of *orig* – which exists by the regular place invariant from Fig. 17 – orders M_x^p before $M_x'^p$ w.r.t. information flow.

www.manaraa.com

5 Discussion

We have given a protocol based model for a weakly coherent DSM-system, and proved this model to satisfy the specification which is based on different causalities. Moreover, we proved that the specification implies the one given in [3].

One should remark that this specification only consists of a safety property. Thus, even a system that deadlocks any user operation satisfies this specification, because it prevents any control flow. In that case nothing must be shown. In [4] we specified an additional liveness requirement and proved the protocols to be live by standard methods of temporal logic.

The proposed model determines all conceptual aspects to yield a correct system. For example, omitting acknowledgements or using mailboxes without FIFO property would destroy correctness. On the other side, we have not yet fixed an *implementation*. The correct model still allows several different implementations, but all of them are proved to work properly. For example, to achieve correctness we demanded that invalidation messages are received prior to synchronization messages, but we have not determined when to invalidate pages. Invalidations may be executed as soon as they are ahead of the buffer or as late as possible, i.e. when the node is waiting for a synchronization message. This is modelled by a *conflict* in the protocols allowing for a variety of correct implementations. Depending on the hardware or the communication frequency of the application tasks the most efficient strategy may be implemented.

We have laid stress on the adequacy of Petri nets and their partial order semantics for the treatment of weak coherence. The same formal method may be applied with benefit to other systems executing distributed programs. For example, we applied this method in [5] for proving a DSM-model which allows for the migration of original pages. More generally, it can be applied to cache coherence protocols [11] and in the field of concurrency control in distributed databases.

Acknowledgements

The ideas in this paper have been developed jointly during a cooperation of the projects A3 and C1 within the SFB 342 at TU Munich. We want to thank L. Borrmann and P. Istravinos (Siemens AG, Munich), who advised us in the concept of weak coherency and provided us with several existing implementations of weakly coherent systems. Moreover, we are grateful to Rolf Walter for some useful comments on earlier versions of this paper and to Dieter Barnard for carefully reading the manuscript.

References

[1] J. K. Bennet, J.B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *2nd ACM SIGPLAN Symposium on Principles and Practise of Parallel Programming*. ACM, March 1990.

[2] P.A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

www.manaraa.com

[3] Lothar Borrmann and Martin Herdieckershoff. A coherency model for virtually shared memory. In *International Conference on Parallel Processing*, August 1990.

[4] Dominik Gomm and Ekkart Kindler. Causality based specification and correctness proof of a virtually shared memory scheme. SFB-Bericht 342/6/91 B, Technische Universität München, August 1991.

[5] Dominik Gomm and Ekkart Kindler. A weakly coherent virtually shared memory scheme: Formal specification and analysis. SFB-Bericht 342/5/91 B, Technische Universität München, August 1991.

[6] Herrmann Hellwagner. A survey of virtually shared memory schemes. SFB-Bericht 342/33/90 A, Technische Universität München, December 1990.

[7] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[8] Andreas Listl and Markus Pawlowski. Parallel cache management of RDBMS. SFB-Bericht 342/18/92 A, Technische Universität München, August 1992.

[9] Wolfgang Reisig. *Petri Nets*, *EATCS Monographs on Theoretical Computer Science*, volume 4. Springer-Verlag, 1985.

[10] Wolfgang Reisig. Petri nets and algebraic specifications. *Theoretical Computer Science*, 80:1–34, May 1991.

[11] A.J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

www.manaraa.com

Object- and Memory-Management Architecture

– A Concept for Open, Object-Oriented Operating Systems –

Jürgen Kleinöder
kleinoeder@informatik.uni-erlangen.de

University of Erlangen-Nürnberg
IMMD 4, Martensstr. 1
D-W8520 Erlangen, Germany

Abstract. Object- and memory management are central components of an operating system for an object-oriented system. This paper describes the functionality of the components object store, object cache and object space and the perspectives resulting from this model for an object- and memory management. If the operating system itself is designed in an object-oriented manner, the question is how to manage the operating system objects. The answer is a hierarchical structure, which will be explained with the example object-store. The possibility to provide objects with operating system functionality within the scope of an application leads to an open operating system architecture. The interaction between application system and operating system may result in reflective and recursive relations between objects.

1 Introduction

File system and memory management are two important elements of traditional operating systems. The concept of *persistent objects* makes a file system in an operating system for object-oriented applications obsolete — a conventional file can be substituted by a persistent object with methods like *read, write* and *seek*. The file-system is replaced by the object management. Objects have to be stored on disk storage like ordinary files, but at execution time, they have to exist in main memory, too. Thus, object- and memory management are important parts of an operating system for an object-oriented system.

Object management and memory management of an object-oriented operating system are implemented with objects — which themselves have to be managed. A layered construction of the operating system will be proposed to avoid the cycle in this argumentation. In addition, such a system offers a simple possibility for adding enhancements and modifications: it is possible to instantiate objects which implement the functionality of an object management and which manage application objects. Furthermore, the propagating of specific object-interfaces in lower operating system layers may make it possible to influence algorithms and strategies of the operating system. Operating systems, being enhanceable or adaptable for the needs of specific applications, are called *open operating systems*.

Object- and memory management are arranged into the components *secondary storage*, *main memory management* and *organization of virtual address spaces*. Secondary storage is used as *object store* while main memory serves as cache for those parts, which are actually needed for the execution (*object cache*). To be able to ex-

www.manaraa.com

ecute an application, the objects have to be mapped into virtual address spaces (*object spaces*).

Object store and object cache may be placed on different nodes in a distributed system. Moving objects between different object caches is imaginable and may be a basis for object migration mechanisms.

New administration strategies, protection mechanisms, support for fault tolerance etc. may be added to a system by instantiation of new object stores, object caches or object spaces.

The introduced object- and memory architecture is a model for operating system components, built according to the *PM system architecture*. Within the PM project at the department of computer science (IMMD IV) of the University Erlangen-Nuernberg we develop foundations of object-oriented operating systems and applications in distributed systems. The PM system architecture deals with the structure of object-oriented operating systems being adaptable to special needs of applications and to hardware architectures.

Chapter 2 will give a short introduction into the two main parts of the PM project: PM object model and PM system architecture. Chapter 3 reviews aspects of memory management structuring as it can be found in existing traditional and modern operating systems. The organization of the management of secondary storage, main memory and virtual address spaces as well as the tasks of these operating system components, are outlined in chapter 4. How these concepts may be structured according to the PM system architecture is demonstrated by an example in chapter 5. The paper ends with a short note about a prototype implementation, a conclusion and some remarks about topics to be investigated in the future.

2 The PM Project — An Overview

Within the PM Project we examine object-oriented structuring and programming of operating systems and applications for distributed systems. The topics *PM object model* — an object-oriented programming model for distributed systems — and *PM system architecture* — structuring of open, object-oriented operating systems — actually form the two focal points of our project. The work is part of the project B2 "Design and Implementation of an Hardware-Architecture- and Application-Class-Adaptable Multiprocessor-Operating-System" of the Sonderforschungsbereich 182 "Multiprocessor- and Network-Configurations".

2.1 PM Object Model

The following topics are actually treated within the framework PM object model:

- *Structuring Mechanisms for a programming language*, which allow a separated description of types and classes to elaborate the differences between the description of object properties and interfaces and the implementation of such objects. Furthermore the concept of inheritance is reduced to aggregation by which improvements in the cases of dynamic object replacement and orthogonal distribution of objects have been achieved [6].

www.manaraa.com

- *Configuration* — To achieve a high degree of reusability it is not desirable to bring into the programming phase of classes information about the kind of relationships between objects and their distribution in the scope of one application. The relations between objects and the distribution of the objects should be described in a separate step — the configuration of an application. A separate configuration language has been developed to achieve this [5].
- *Synchronization* — Concurrency within objects is regarded as a fundamental concept of the PM object model. The works on the topic *synchronization* deal with the problem of how to separate the implementation of the synchronization and the implementation of the classes — again with the aim to avoid restriction of reusability wherever possible [8].

2.2 PM System Architecture

The structuring-concepts, offered by an object-oriented programming model, are very helpful in designing an operating system, but are not sufficient to bring order into the complexity of such a software system — additional structuring guidelines are needed.

The PM system architecture describes the construction of an operating- and runtime support system by the means of *meta-layers* and the interfaces between an application system and meta-system.

By the construction with several meta-layers the abstractions of the programming model are developed step by step. The power of the abstractions is increased from layer to layer. The PM operating system layers are programmed based on the PM object model [7]. Each layer can be regarded as an application layer in the sense of the object model, where a subset of the abstractions of the object model are available, but are partly less powerful. The underlaying layer — implementing theses abstractions — is called *meta-layer* (Fig. 2.1). The implemented abstractions are called *meta-abstractions* [16] (see also the meta-space concepts in Apertos / Muse [12], [13], [14], [15]). From the standpoint of an application object, the objects within the meta-layer are *meta-objects*. The level of abstraction for programming operating-system mechanisms increases, according to the position of the programming in this meta-stratification. In this way programming comfort is improved and adaptation, enhancement and maintenance of operating-system software becomes easier.

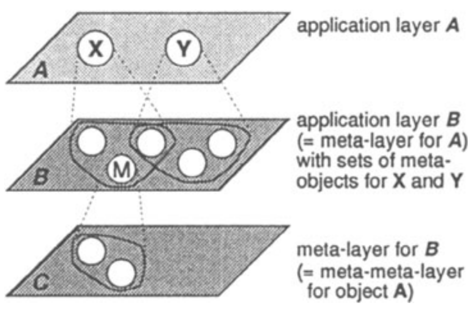


Fig. 2.1 Application layers and meta-layers

Based upon a complete implementation of all abstractions of the PM object model, it is possible to realize objects which implement improvements of this object

www.manaraa.com

model or which implement a run-time support system for a totally different programming model. Such objects form a new meta-layer for a new application layer. The operating system becomes expandable this way (Fig. 2.2).

A common PM object model for application layer and meta-layer — although with restrictions within the meta-layer — makes the interfaces inside the meta-layer available for the objects at the application layer (Fig. 2.2). By this, the possibility

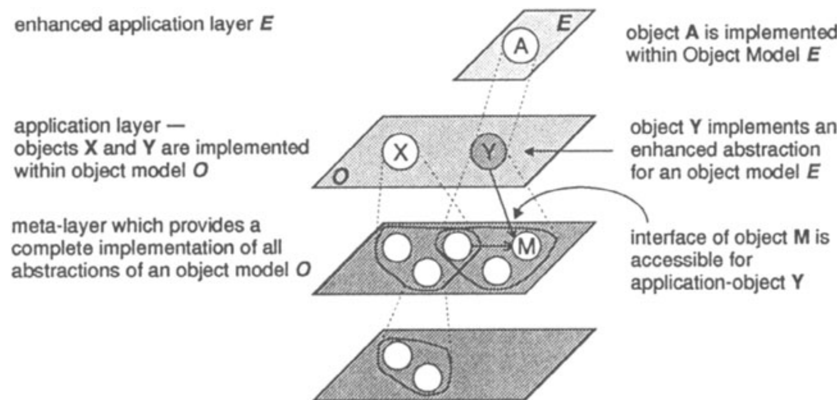


Fig. 2.2 Application-specific meta-layer and open meta-layer-interface

for applying application specific modifications to the meta-system — and thus to the operating system — can be provided. The operating system becomes an *open* System.

Object-oriented structuring, meta-layering and the concept of an open operating system, where operating-system objects are protected by the means of the invocation interfaces and not by incompatible interfaces, are the essence of *general purpose operating systems*, which will be adaptable both to application demands and to different hardware architectures.

Two special cases of interaction between application system and meta-system have to be emphasized: *Reflection* — an application object communicates with its meta-object and thus influences its own implementation — and *meta-recursion* — an object uses an abstraction which it implements itself.

3 Memory-Management Architectures

3.1 Traditional Operating Systems and Micro-Kernels

For traditional operating systems, the tasks of memory management can be grouped into three categories:

- Secondary storage (normally realized by file systems and swap spaces)
- Main memory management

www.manaraa.com

- Management of virtual address spaces (often joined tightly with the notion of a process)

In modern operating systems designed for high portability (e. g. MACH [10] or Chorus [2]), one has to distinguish within a memory management system between an architecture-dependent part, which should be as small as possible, and an architecture-independent part.

Furthermore in micro-kernel architectures there exists a kernel-predefined part of the memory management system and optionally additional components implemented in application space (e. g. *external pagers* in MACH [11]). External memory management modules are much easier exchanged or adapted to special demands of applications.

A further criterion for classifying memory management architectures is the cooperation between main memory and secondary storage. A concept, widely distributed in older operating systems, is *swapping*: Pages or segments of main memory are transferred to *swap space* if main memory runs short. Newer operating systems consider the main memory as *cache* for data stored on secondary storage. Generally all data needed for the execution of an application is placed on secondary storage and is transferred, as needed, to main memory at execution time. It is shifted back to its place on secondary storage once the main memory runs short or the execution terminates (*file mapping concept*).

3.2 Object-Oriented Operating Systems

Object-oriented systems allow further simplifications. Applications consist of a set of cooperating objects. Some of these objects should outlive the execution of the application — they are called *persistent objects*. Persistent objects make the notion of a file obsolete. A traditional file is a normal persistent object with methods like *read, write* or *seek*. The part of the file system within the operating system is substituted by the management of persistent objects — the object store.

The concept of a virtual address space is not much different from such a concept in traditional operating systems. In some systems it is separated from the notion of activity — the thread concept (e. g. in Clouds [3]) in contradiction to systems like UNIX, where a process is the unification of a thread and a virtual address space.

The cooperation between main memory management and object store can be grouped into two categories:

- Persistent objects are deactivated, frozen and transferred to the object store. Normally this is achieved by explicit statements in the users program (see e. g. Arjuna [4]).
- Main memory is considered as cache for objects which are placed in an object store.

www.manaraa.com

3.3 Conclusion

The aspects of different memory management systems mentioned in the above sections can be combined to a horizontal and vertical structuring:

- Horizontal division:
  - Secondary storage management
  - Main memory management
  - Management of virtual address spaces

- Vertical division:
  - Architecture-independent components (as few as possible)
  - Architecture-independent management
  - Programming model dependent interfaces (e. g. to support the notion of an object)

4 Structure of the PM Object- and Memory Management

This chapter describes abstractions for the three components identified in the horizontal division in the previous section. How important it can be to enhance or modify such operating system mechanisms will be demonstrated by examples of very different demands from applications. An implementation of theses abstractions can serve as an operating- and run-time support system for an object-oriented programming model, like the PM object model. Besides this, it will be possible to support other programming models or emulations of other operating systems, based upon the described mechanisms.

4.1 Object Store

In the phase of programming and configuring objects, demands for the storage management of these objects are defined as well. Organization and storage of objects should be handled according to these demands and should not be influenced by demands coming out of the hardware architecture or the execution environment of the application (e. g. a distribution configuration for the objects of the application).

Object stores are introduced for accomplishing the described tasks in a PM system. The functionality of an object store covers the following topics:

- Management and storage of classes, instances and their states in a certain quality for a certain or uncertain time period.
- Management of object groups which have been linked statically at compile time (called *system objects* or *distribution units*).
- Creation of object groups out of single objects and/or other object groups according to certain criteria.
- Management and interpretation of attributes and management directives (configuration, protection demands, execution experiences, statistic data, etc.) of objects and object groups, which describe instructions for the object storage.

www.manaraa.com

- Management of attributes and management directives for the organization of the mapping of objects or object groups into an execution environment and for the execution itself, but without interpreting them.

Besides these objectmodel-oriented functionalities, object stores have to implement mechanisms, known from base layers of file systems in traditional operating systems:

- Buffering in- and output
- Management of secondary storage
- Control of the disk-hardware

The specific demands for object store functionality may be very different for various applications:

- fast storage
- stable storage
- long term storage
- short term storage
- architecture-independent storage-format
- efficient in- and output of large amounts of data
- efficient in- and output and storage-utilization for small data-units
- special protection-mechanisms
- no protection-mechanisms

These examples give distinct illustrations for the reason of demanding that an adaptation to the individual needs of an application has to be made possible. Often the particular criteria are not compatible and it is necessary to make compromises — however such compromises often have to be very different for different demands.

4.2 Object Space and Application Space

The organization of objects of an application inside object stores is not proper for a concrete execution of the application. Data may not be available in the main memory of a node of the distributed system and the execution of operations and the addressing of data by a processor require mapping into a virtual address space of the node of the processor.

Additionally, if there is not just a single node but a distributed system available for the execution of an application, it will be necessary to plan and coordinate the distribution of the application objects within that distributed system. The object store defines a location for each object. These predefined locations of several cooperating objects are not necessarily compatible with the criteria given for an efficient execution under consideration of all directives (like protection, configuration load balancing, etc.).

Thus an execution environment is necessary, which allows to organize all objects of an application in a manner which is proper for an optimal execution. Tasks of such an execution environment may be, among others:

www.manaraa.com

- Representation of objects in a format appropriate to the demands of the execution control allowing a highly efficient execution.
- Organization of resources and objects in a manner guaranteeing that all directives made about object relationships (e. g. about protection of the objects) are met. Such directives may be stored in the object store together with the objects.
- Providing mechanisms to support method invocations between objects which have not already been bound at compilation time.

The most common abstraction of such an environment is called an *application space*. Application spaces are location-transparent things which may accommodate objects of applications. Configuration-attributes of such objects give directives to an application space. These directives explain how to distribute the objects to node-specific *object spaces* which build an address- and protection-spaces.

The main task of an application space is the planning and coordination of the object-distribution during the execution-time of the application. The basis for these decisions are the configuration information and protection demands of the application objects and data; determined by the operating system at execution time.

Thus, an application space has to deal with the following topics:

- Creation of object spaces
- Finding the object stores of application objects
- Finding and — if necessary — creating object caches (see section 4.3) for application objects
- Run-time-control for an application
  - Which objects are mapped into which object spaces?
  - Initiation of object-migration because of dynamic reconfiguration- or load balancing mechanisms

Object spaces can be considered as an abstraction of virtual address spaces. Inside an object space, application objects are supplied with run-time support which is specific to the programming-environment.

Examples for such run-time support are:

- Dynamic binding for local method invocations
- Forwarding of non-local method invocations by RPC-mechanisms
- Transformation and forwarding of instructions of the programming environment for the operating- and run-time support system (checkpoints, transaction results, instructions to initiate object migration)
- Dynamic loading of target-objects in case of a method invocation
- Supervising of local method invocations for certain objects

It is imaginable to have even different object- and application spaces, depending on the hardware architecture and the environment needed for a certain application — some examples:

www.manaraa.com

- Forward planning of the mapping of objects into object spaces to avoid fault situations in the case of method invocations
  - for applications whose distribution can be statically ascertained and which do not need dynamic changes of its distribution during execution time
  - for hardware-architectures, which do not support an efficient trap-handling (e. g. several of the modern RISC-Processors [1])
- Lazy evaluation techniques for avoiding unnecessary mapping of objects into object spaces in the case of dynamic object-migration during execution time
  - for applications whose object-distribution can not be statically ascertained
  - for hardware-architectures, which support an efficient trap-handling

Thus, it should be possible to have different realizations of object space and application space, depending on the needs of the respective application.

4.3 Object Cache

Application space and object space are only responsible for the mapping of objects into nodes and then into regions of virtual address spaces of those nodes. They are not responsible for transferring and positioning the objects into the main memory of the respective node. Of course, an execution of operations of objects is only possible, if the code and data are available in the main memory and addressable by the processor.

Mechanisms to manage the main memory of a node, for transferring objects between main memory and object store and for the supply of information, needed for a valid mapping within an object space are still missing.

The cooperation between main memory management and object store should be realized according to the *file mapping concept* (see section 3.1). Thus, the main memory acts as a cache for the object store and that is why it is called *object cache*. As in all page- or segment-oriented memory-architectures, it is possible to cache object data in units of segments or pages — so it is not necessary to keep all objects of an application completely inside the object cache during the whole execution time.

An object cache is not bound to a specific application. Its most important tasks include the following topics:

- The guarantee of the protection attributes, which are stored in the object store together with the object (Which object spaces are allowed to map the object?)
- Caching of often-needed management data of the object to reduce the accesses to the object store and to improve execution time
- Mapping of objects into object spaces
- Update of object-images in the object stores
- In- and output of objects or parts of objects between main memory and object stores

Especially to be able to adapt the cache-algorithms to special properties of object or object-data (e. g. large, coherent data-regions for images), it can be advantageous

www.manaraa.com

to have different implementations of object caches, adapted to the specific properties of the applications (different paging-strategies, larger blocks for I/O, etc.).

4.4 Conclusion and Perspectives of the Model

With the PM object- and memory management, objects are created and stored into object stores. To be able to invoke methods of an object during the execution of an application, objects are mapped into object spaces. Real addressing of object-data by the means of a processor-instruction is only possible if the data is available in main memory of the respective node. Main memory acts as cache for object data in an object store. (Fig. 4.1)

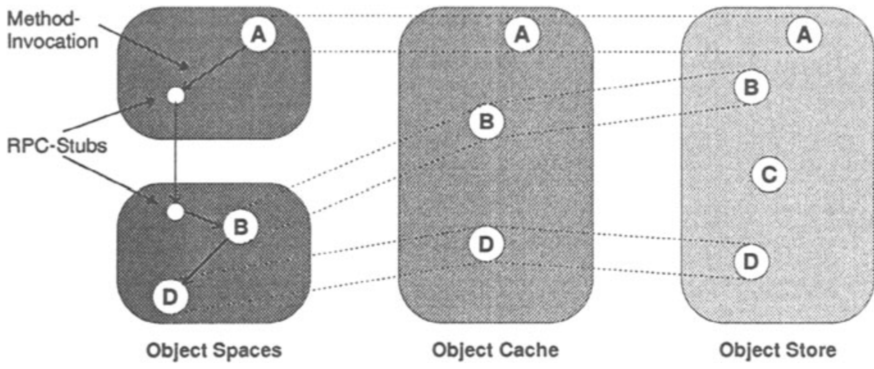


Fig. 4.1 Object store, object cache and object spaces — overview

While object cache and object space always have to reside on the same node (collocation relationship, see [5]), the object store may reside on a different node within the distributed system.

It is imaginable to cache objects in several object caches on different nodes of the distributed system. This can be seen as a realization of *distributed shared memory* on the basis of objects. Of course it will be necessary to run cache-coherence-protocols between such object caches to avoid inconsistent object-states. Besides the caching of an object into several caches it may also be possible to move objects between caches on different nodes. This is equivalent to an object-migration at runtime and may be a basis for the implementation of dynamic load-balancing mechanisms. Mapping of an object into several object spaces is equivalent to the concept of *shared memory* in traditional operating systems. (Fig. 4.2)

These outlined examples demonstrate how a lot of operating system mechanisms for distributed systems can be covered conceptionally with the introduced model for an object- and memory management system.

www.manaraa.com

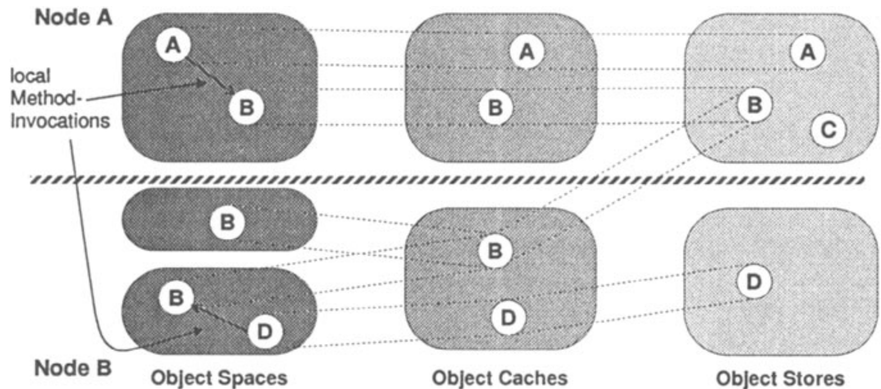


Fig. 4.2 Caching of an object in two object caches and mapping of an object into two object spaces

5 Construction of the PM Object- and Memory Management within the PM System Architecture

The structure of object store, object cache and object space described above corresponds on the whole to horizontal division of a memory management, as described in section 3.3. The PM system architecture (see section 2.2) defines guidelines of how to build such components in an object oriented manner. This will be illustrated in the following by the example "object store".

To express it simply: an object store is a meta-object, which is capable of storing application objects. Analogous of this concept, an object cache is a meta-object, which is able to cache objects. In addition, an object space is a meta-object which acts as an execution environment for objects. These meta-objects are a part of an object-oriented constructed operating system.

5.1 Hierarchical Meta-Systems — Example Object Store

Naturally the question arises of how, for instance, the object *object store* is stored. As objects can only be stored in object stores, an object store — this time an object of the next lower meta-layer — is needed to accomplish this task. As already mentioned in section 2.2, the power of the abstractions should decrease, the lower the meta-layer is in which it is implemented. In the case of an object store this can mean, for instance, that there are no protection mechanisms implemented or that objects inside such an object space are not allowed to change their size dynamically. Even this simpler object store can be implemented as an object and it is stored in an even simpler object store — a meta-object in the next lower meta-layer (Fig. 5.1).

www.manaraa.com

While all the *higher* object stores are realized as architecture-independent objects, it is imaginable to have a lowest object store which implements the architecture-dependent functions and which is implemented outside the object-oriented programming model. However, although it is able to store objects and has an interface like an ordinary object, it is not really an object — and thus it does not need to be stored in an object store. Instead it may come into existence during the boot-phase of the system.

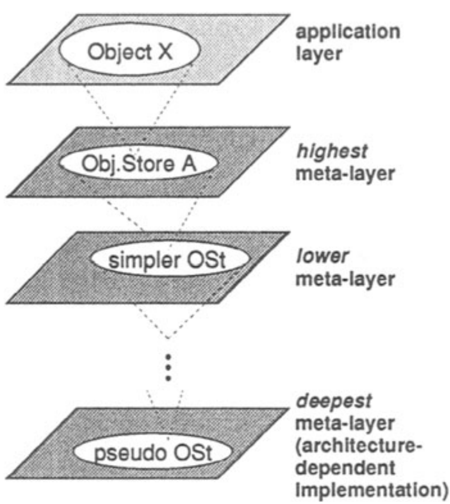


Fig. 5.1 Example of a meta-hierarchy of object-stores

The example *object store* shows how operating system components can be constructed by using the object-oriented paradigm. It also shows how the abstractions provided by the operating system for the object-oriented programming model are already used for the operating system objects themselves — even if with reduced functionality. The number of meta-layers an operating system will have is not fixed. The number can be different, even between applications on the same node. It is possible to create an object, within the context of an application, which fulfills the functionality of an object store. Other objects of the same application may be stored in that object store — this means, that an additional meta-layer exists for that application. Just like this the number of meta-layers may be different for different function-groups of the operating system.

The structuring of memory management systems, as described in section 3.3, can be adopted only with restrictions. Of course it is desirable to separate architecture-dependent and architecture-independent parts of the implementation. But to be able to implement the operating system itself by using an object-oriented programming model, it is necessary to have it supported already by the lowest layer of the operating system.

5.2 Open Operating System Architecture

As it has already been mentioned in section 5.1, it is possible to create objects within the context of an application, which do offer operating system functionality to other objects of that application. In this way, the operating system can be adapted to special demands of certain applications without interfering with other applications. Thus, the operating system is hierarchically expandable (Fig. 5.2).

As application objects and operating system objects can be implemented by using the same object-oriented programming model, it is possible for application objects

www.manaraa.com

to invoke methods of objects of the operating system — under the condition that the application object has a reference for the operating system object. With such a method-invocation, the application object may transfer references to those application objects which may be used by the operating system object to realize its services — if it is programmed to do so. Thus, the used application objects serve to realize operating system functionality — they become indirect parts of the operating system (Fig. 5.3).

The border between application system and operating system fades away in this way. The protection of the operating system is no longer achieved by encapsulating it into a *kernel* but by a protected method-invocation and by not giving away references to operating system objects to unauthorized entities.

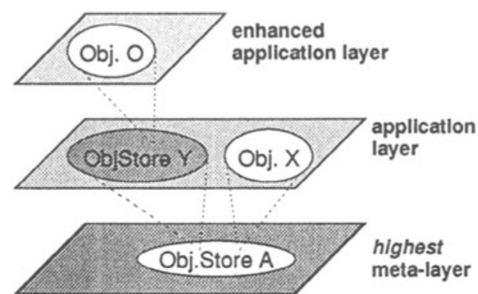


Fig. 5.2 Enhancement of the operating system by providing objects with operating system functionality at the application layer

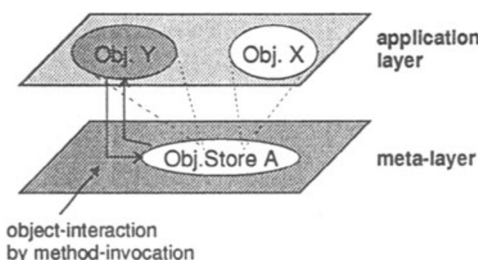


Fig. 5.3 Interaction between application objects and operating system objects

5.3 Reflection

In a system in which application objects and operating system objects may interact the possibility arises that an application object invokes methods of an operating system object, which implements just that application object — and is thus a meta-object.

This would mean, given the example of the object store, that an object interacts with its own object store. In the context of such an interaction the object can give directives to change its own storage structure — for instance by a migration into a different object store which implements redundant storage. As an extreme example the application object would also be able to delete itself.

Interaction with the meta-system — that means with the actual implementation — is called *Reflection* (see also [9] or [7]).

In systems which allow interactions between application objects and meta-objects, reflection is not an additionally introduced mechanism but implicitly available. The example above already suggests which consequences can arise from un-

www.manaraa.com

controlled use of reflection. Therefore, the possibility for an explicit description method is demanded for the PM system to be able to describe the use of reflection within an application.

5.4 Meta-Recursion

Another interesting effect arises if cycles result out of the interaction between an object and its meta-system. This means an object becomes its own meta-object. Given again the example of the object store: An object with the functionality of an object store is created in the context of an application. Subsequently this object contacts its object store and initiates its migration into another object store — and passes its own reference as target-object-store-reference. The result would be that the object will manage itself afterwards. All this may not be critical at execution time, as long as the object is cached in an object cache. It seems clear that swapping out the object to the object store will have undesirable consequences. This would be comparable to the situation arising in a traditional operating system if the part which is responsible for paging were to be paged out.

The details of how to use meta-reflection in operating system programming have not been examined, yet.

6 Prototype-Implementation

We have developed a prototype of the components object store, object cache and object space based on MACH and have gained first experience with the proposed model. The prototype allows storage of objects, dynamic mapping in one or several object spaces and migration between object spaces. The object cache is implemented as external pager to the MACH kernel and the object store uses normal files to store its data.

To support objects being dynamically mapped into several object spaces, code and data have to be position-independent. Furthermore, special glue-code for method-invocations is needed. The first simple test-applications were generated by modifying the assembler-code. To be able to run larger applications we are currently developing a modified C-compiler to generate appropriate executables.

7 Conclusion and Future Work

We proposed a model of an architecture for an object- and memory management for an object-oriented operating system. The components object store, object cache and object space form abstractions for the management of secondary storage, main memory and virtual address spaces, as it can be found in traditional operating systems.

We have outlined how it is possible to realize mechanisms like distributed shared memory, shared memory and object-migration on the basis of these abstractions.

By realizing the different components as hierarchical meta-systems, it will be possible to increase programming comfort for the implementation of large parts of

www.manaraa.com

the operating system functions because even if the abstractions are not fully developed in lower layers of the operating system, the implementation can be still done on the basis of an object-oriented programming model and its abstractions. In contrast to this, the implementation of most other operating systems has to be done on the bare hardware.

In future work reflection and meta-recursion will be treated in more detail. Very importantly the description of reflective or meta-recursive relations between objects by constructs of the programming model has to be investigated.

Through the possibility of bringing new meta-objects into the system a new flavor of relationship between objects has come into being: one object implements the state or the behavior of another. Existing protection mechanisms cover only the control of normal object-relationships (one object invokes a method of another). Concepts which answer, for instance, the question "which object cache may cache which objects?" are not yet known. Future work in the PM project will try to give an answer to this problem.

References

1. Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska, "The Interaction of Architecture and Operating System Design", *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pp. 108-121, Santa Clara (CA, USA), published as *SIGPLAN Notices*, Vol. 26, No. 4, Apr. 1991.
2. Vadim Abrossimov, Marc Rozier, and Marc Shapiro, "Generic Virtual Memory Management for Operating System Kernels", *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pp. 123-136, Litchfield Park (AZ, USA), Dec. 1989.
3. P. Dasgupta, R. J. LeBlanc, M. Ahamed and U. Ramachandran, "The CLOUDS Distributed Operating System", *IEEE Computer*, Apr. 1991.
4. G. N. Dixon, G. D. Parrington, S. K. Shrivastava, and S. M. Wheater, "The Treatment of Persistent Objects in Arjuna", *Proceedings of ECOOP '89, the Third European Conference on Object-Oriented Programming*, pp. 169-189, Ed. S. Cook, Cambridge University Press, Nottingham (UK), Jul. 1989.
5. Michael Fäustle, *An Orthogonal Distribution-Language for Uniform Object-Oriented Laguages*, Internal Report IV-16/92, University Erlangen-Nuernberg: IMMD IV, Dec. 1992.
6. Franz J. Hauck, *Typisierte Vererbung, modelliert durch Aggregation*, Internal Report IV-15/92, University Erlangen-Nuernberg: IMMD IV, Nov. 1992.
7. Gregor Kiczales, Jim des Revières, and Daniel G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.
8. Rainer Pruy, *Cooperative Concurrency Control*, Internal Report IV-18/92, University Erlangen-Nuernberg: IMMD IV, Nov. 1992.

www.manaraa.com

9. Ramana Rao, "Implementational Reflection in Silica", *Proceedings of ECOOP '91, the Fifth European Conference on Object-Oriented Programming*, pp. 251-267, Geneva (Switzerland), Ed. P. America, Lecture Notes in Computer Science No. 512, Springer-Verlag, Jul. 1991.
10. Richard F. Rashid, Avadis Tevanian, Jr., Michael Wayne Young, David B. Golub, Robert V. Baron, David L. Black, William Bolosky, and J. Chew, "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", *IEEE Transactions on Computers*, Aug. 1988.
11. Avadis Tevanian, Jr., *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments*, PhD thesis, Technical Report CMU-CS-88-106, School of Computer Science, Carnegie Mellon University, Dec. 1987
12. Yasuhiko Yokote, "The Apertos Reflective Operating System: The Concept and Its Implementation", *OOPSLA '92 - Conference Proceedings*, pp. 414-434, Vancouver (BC, Canada), published as *SIGPLAN Notices*, Vol. 27, No. 10, Oct. 1992.
13. Yasuhiko Yokote, Atsushi Mitsuzawa, Nobuhisa Fujinami, and Mario Tokoro, "Reflective Object Management in the Muse Operating System", *Proceedings of the 1991 International Workshop on Object-Orientation in Operating Systems*, IEEE, Oct. 1991.
14. Yasuhiko Yokote, Fumio Teraoka, Atsushi Mitsuzawa, Nobuhisa Fujinami, and Mario Tokoro, "The Muse Object Architecture: A New Operating System Structuring Concept", *Operating Systems Review*, Vol. 25, No. 2, Apr. 1991.
15. Yasuhiko Yokote, Fumio Teraoka, Mario Tokoro, "A Reflective Architecture for an Object-Oriented Distributed Operating System", *Proceedings of ECOOP '89, the Third European Conference on Object-Oriented Programming*, pp. 89-106, Ed. S. Cook, Cambridge University Press, Nottingham (UK), Jul. 1989.
16. Peter Wegner, "Concepts and Paradigms of Object-Oriented Programming", *ACM OOPS Messenger*, No. 1, pp. 8-84, Jul. 1990.

www.manaraa.com

An Orthogonal Distribution Language for Uniform Object-Oriented Languages

Michael Fäustle
faust@informatik.uni-erlangen.de

University of Erlangen-Nürnberg
IMMD4, Martensstraße 1
D-W 8520 Erlangen, Germany

Abstract. On one side the complexity of the design of distributed application systems is usually reduced significantly by using a distributed programming model that abstracts from the actual distribution and maybe even from the actual decomposition of the application. On the other side the developer needs to express the static and dynamic cooperation properties of the distributed application and therefore needs to control its decomposition and distribution.

An orthogonal distribution language *(ODL)* is presented. It allows the developer to independently describe and adapt the decomposition and distribution of an application written in a uniform object-oriented and distribution-transparent language independently; i.e. without affecting the semantics of the application.

1 Introduction

A distributed application is a system that consists of cooperating components which run on the nodes of a distributed computer system. The *distribution of the application* is defined by the decomposition of the application into distributable units and their initial and dynamic assignment to the computing nodes; i.e. the *distribution of the units*.

A distributed programming model must abstract from the actual distribution of the application to reduce the programming complexity and to allow for reusability and optimization, which can then be *orthogonal*; i.e. the optimization does not affect the semantics of the application. This orthogonal adaptation of the distribution for optimization requires a separate programming language, an *orthogonal distribution language*. The decomposition of the application into distribution units, their initial distribution state and their distribution behavior can then be described and adapted to the requirements.

The need for such an orthogonal distribution language that should complement the distribution transparent programming model of the distributed operating system *LOCUS* has already been recognized by Popek and Walker (from [17]):

> "*One can think of the set of functions by which the distributed system provides service to applications as an effect language. [...] We argue that a* ***separate optimization language*** *should be created,* ***orthogonal*** *to the effect language. The optimization language is* ***semantics free****, in*

www.manaraa.com

> *the sense that whatever is stated in the optimization language cannot effect the outcome of a program; i.e., cannot change the result of any statement or series of statements in the effect language. The optimization language permits one to determine the location of resources, request that the system move a resource, etc.*" [Boldface not in the original]

Uniform object-oriented programming languages like *Smalltalk* [9] and *Emerald* [2] use only one kind of object definition. they minimalize the restrictions on the distribution, more specifically the decomposition of an application: The objects define the minimal decomposition of the application. They are therefore an ideal basis for applying an orthogonal distribution language.

The distribution language can then be used for statically optimizing the implementation of objects and for dynamically optimizing their cooperation. To make this task tractable for the programmer parts of the description must be inferred from analysis or monitoring of the application.

Chapter 2 discusses the requirements for the programming language and alternatives for the realization of a distribution language. The major aspects of the orthogonal distribution language are presented in chapter 3.

2 Describing the Distribution of an Application

The *distribution description* of an application defines the initial state and the dynamic behavior of a *distribution system.* Its initial state defines their initial distribution. Its behavior prescribes the change of its state. The distribution system reacts to events of the application (*local events*) or of the entire distributed system (*global events*). An actual distribution results (eventually) in an assignment to *loci*, like nodes, that are defined by a run-time system (*RTS).* This is a simple, but very useful generalization of the concept of distribution.

2.1 Goals

Two central goals for a distribution system are considered:

- "*Optimization*"
  The cooperation of objects solving a common task should be improved; e.g. the communication and management-overhead should be reduced through *collocation* (*clustering*) as well as the parallel execution of tasks should be enabled through *dislocation*. The run-time system can define additional kinds of loci that allow for optimization; e.g. clustering for transfer operations in virtual memory systems [22].
- *Functional properties*
  In a distributed computer system nodes are assumed to fail independently. The assignment of objects to nodes therefore affects their failure properties. The run

www.manaraa.com

time system can define additional kinds of loci that allow the expression of additional orthogonal functional properties; e.g. the encapsulation of a set of objects.

2.2 The Programming Language

An explicit orthogonal distribution description can only be used when the programming language is *logically distributed* ([3]) and therefore defines distributable units, but is *distribution transparent*, i.e. it abstracts from their concrete distribution (*distribution transparency*). The definition of these units implies restrictions on the possible decompositions of the application. To minimalize these restrictions the programming language should only allow the description of a minimal fine-grained decomposition that can be coarsened by the distribution description.

Uniform object-oriented languages like *Smalltalk* [9], *Self* [21] and *Emerald* [2] satisfy these requirements: an object, as a set of variables and operations, forms a minimal distribution unit and there is only one kind of object. Except the handling of errors that result from a distributed execution, these programming languages can be implemented as distributed languages without sacrificing distribution transparency and efficiency ([13]).

2.3 Integrating Distribution and Programming Language

An explicit distribution description can be achieved by using two approaches: an extension to the programming language or a separate distribution language. Most distributed programming languages provide some of the required extensions to describe the distribution; like constructs for the decomposition and the explicit assignment to nodes, even though the distribution transparency of the programming language allows for an orthogonal, and therefore adaptable, description in a separate distribution language.

The decision for the second approach is motivated by contrasting the effects of both approaches on the following areas:

- *Distribution transparency of the programming language*
  Combined approach: The distribution becomes part of the state of the application. Therefore distribution transparency canno longer be guaranteed.
  Separate approach: The distribution transparency is not affected.
- *Reusability of classes*
  Combined approach: Classes can be reused by instantion or inheritance. Both kinds of reuse are hindered. To reuse a class it not only must satisfy the requirements with respect to its functional behavior but with respect to its distribution behavior as well. When inheriting from a base class, the subclass must now consider the distribution properties of its parent class which leads to an undesirably strong coupling.
  Separate approach: Different distribution descriptions can be assigned to objects of the same class or to objects of a subclass, depending on their use. The reusability of classes is not compromised.

www.manaraa.com

- *Adaptability of the distribution description*
  Combined approach: The distribution description is given as part of the class description. An extension of this description requires changes to the class. Otherwise only parts of the existing description can be chosen; e.g. by parameterizing.
  Separate approach: The distribution properties of an object can be adapted statically and dynamically to their use in the application without access to the source code of the classes.
- *Complexity for the programmer*
  Combined approach: The programmer needs to know only one language. The required extensions to the language add a moderate amount of complexity.
  Separate approach: The programmer has to learn a new language with concepts that differ substantially from those of the programming language.
- *Expressiveness of the distribution language*
  Combined approach: The distribution description is severely limited by the concepts of the programming language. In an object-oriented language, for example, it is hard or impossible to describe the reaction to events.
  Separate approach: The distribution system can be described in an adequate description model. Only concerns for simplicity and efficiency limit the expressiveness of the distribution language. In addition, the distribution system must not interfere with the application: it can only monitor its state, not change it.

The approach taken to define a separate orthogonal distribution language was chosen for the following reasons:

- it avoids the reusability problems
- an adequate description model can be used
- it allows for an easier adaptation of the distribution description of an application

3 The Distribution Language

3.1 The Distribution Model

The distribution model forms the base for the definition of the state model of the distribution language. It describes what kinds of loci the run-time system defines as well as the permissible assignments to these loci. Only two kinds of loci are introduced whereas, in principle, any number could be used:

- *nodes* — The distribution on nodes can be used in two ways: to optimize and to describe failure properties of objects; i.e. dependent or independent failure. The following properties of nodes are assumed:
  - *independent failure:* Nodes fail independently.
  - *mobility:* Assignment to nodes can be changed at any time.
  - *unique assignment:*
    An object must be assigned to exactly one node at a time. Run-time systems that allow for the transparent replication of objects, i.e. *Amber* [6] and the system described in [15], are not considered.

www.manaraa.com

- *capsules* — Capsules allow for the description of the encapsulation of a set of objects. Objects that are assigned to a capsule are protected from objects assigned to a different capsule. This purposely resembles an address space that is independent from the assignment to nodes.
  An object can only initially be assigned to any number of capsules.

3.2 The State Model

The state model of the distribution system defines how an actual distribution is represented in the distribution system. Two observations motivate the choice of the model:

- *relative distribution*
  The key to an abstract problem-oriented distribution description is to abstract from the properties of the nodes and to consider them as freely exchangeable, i.e. there are no restrictions on the mobility of objects. The assignment of objects to nodes and capsules can then be described indirectly by the relative position of pairs of objects. Distribution associations between pairs of objects imply restrictions on the assignment to nodes and capsules. A consistent set of distribution associations characterizes a set of permissible assignments to nodes and capsules. It is the task of the run-time system to choose one particular assignment according to the goals of its global distribution system.
- *distribution of potential objects*
  The distribution system cannot influence the creation and destruction of objects, it can just observe it. Therefore distribution associations relate to potentially existing objects. The set of the actual existing objects determines which distribution associations can be effective. The set of distribution associations therefore defines a potential relative distribution of which the effective relative distribution is a subset.

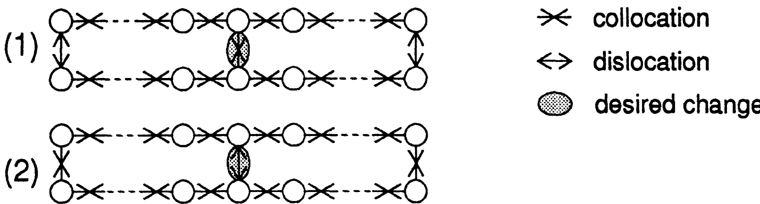
This kind of a relative potential distribution offers among others two advantages compared to a direct assignment to the loci in the run-time system:

- *problem-oriented* — The required distribution properties of objects can be directly specified in terms of object associations, an important concept in the object-oriented paradigm [18].
- *abstract* — The description does not unnecessarily restrict the distribution system of the run-time system. These distribution systems can coexist.

Three kinds of relative placements of objects must be expressible: collocation, dislocation and indifference. A collocation association prescribes the assignment to a common locus, a dislocation association prescribes the assignment to different loci and indifference association prescribes an independent assignment. The last one is necessary to explicitly override inferred associations. All three kinds of associations must be defined for every kind of locus, here nodes and capsules. To allow for a static optimization, mutable and immutable distribution associations must be distinguishable.

www.manaraa.com

Conflict during creation or reversion of associations



Conflict resolution by destruction of conflicting associations

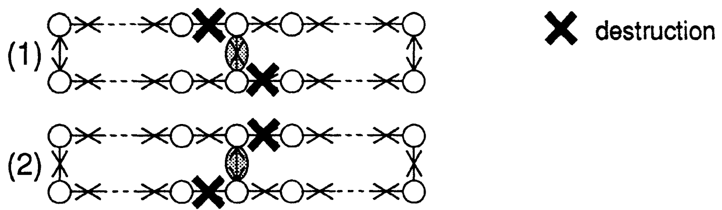


Fig. 3.1 Conflicts and their resolution

A relative distribution is completely determined if, between all pairs of objects for all kinds of loci, there exists a collocation or a dislocation association. The distribution associations could be used to describe distributions that are invalid in the distribution model and therefore cannot be realized by the run-time system. A relative distribution is *consistent* if it only describes valid distributions in the distribution model and therefore satisfies the following requirements:

- The distribution associations of one kind of locus are exclusive; e.g. two objects may not be collocated and dislocated at the same time.
- The distribution associations must be satisfiable by assigning each object to exactly one node. Therefore if one assumes that the associations in the transitive closure of the collocation association are implicitly defined then they must be consistent with the dislocation associations.

The distribution associations of capsules are immutable. Therefore changes are only possible for node associations. Changes of an association between a pair of objects may lead to an inconsistent relative distribution. *Destruction* of a distribution association means a change from a collocation or a dislocation into an indifference association. This change always leaves the relative distribution consistent. *Creation* of a distribution association means a change from an indifference association to a collocation or dislocation association. *Reversion* means a change from a collocation to a dislocation association or vice versa. The creation or reversion of a distribution association may lead to an inconsistent relative distribution. In this case the changes are in *conflict* with existing associations. A conflict can be solved by abandoning the change or by destructing the conflicting associations. Figure 3.1 shows possible conflicts and their resolution by destructing conflicting associations.

المنارة للاستشارات
www.manaraa.com

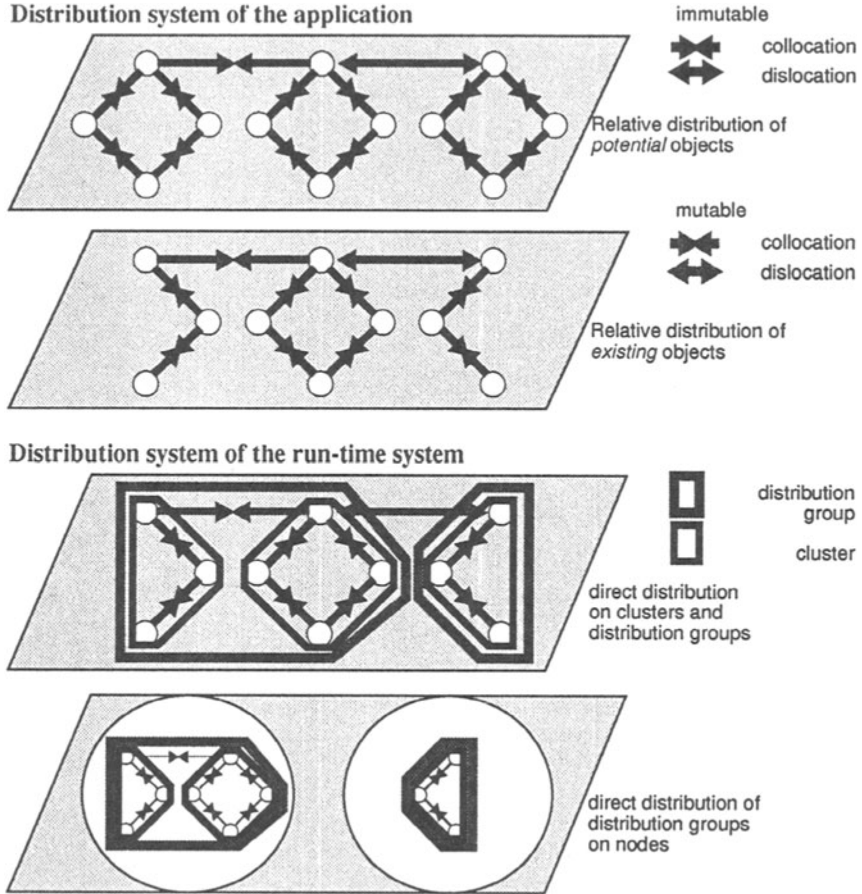


Fig. 3.2 Implementing the state model

3.3 Implementation Considerations for the State Model

The run-time system must implement the state model described above. *Cluster* and *distribution groups* are introduced to represent immutable and mutable collocation associations. They could be considered as a new kind of locus to which the run-time system assigns application objects, depending on their collocation associations. [20] and [10] discuss the concept and possible implementations of clusters and distribution groups. Figure 3.2 gives an overview of the transformation of the relative distribution of potential objects into a direct assignment of objects to clusters, of clusters to distributions groups and of distribution groups to nodes. This last assignment must respect the dislocation associations between distribution groups but it is otherwise controlled by the run-time system. The analogous diagrams for the distribution on capsules have been omitted for brevity.

www.manaraa.com

3.4 Inferring Distribution Associations

In order to reduce the overhead of describing the distribution system of an application it is essential —at least partially— to infer the description that can then be adapted and supplemented by an explicit description. This is only sensible for distribution associations referring to nodes because protection requirements can hardly be inferred. There are two kinds of information with respect to their collection time:

- *static (a-priori) information* — Static information can be used for static and dynamic optimizations and can be gathered by
  - textual analysis of the application systems code and configuration
    Information valid for all possible use cases can be collected and used to heuristically derive distribution associations.
  - monitoring of test runs of the application
    The information collected can hardly be generalized when the information is not collected in test runs for all relevant use cases. General heuristics are hard to identify. This is mostly useful to improve and validate the explicit distribution description.
- *dynamic information* — Dynamic information can be used for dynamic optimization only. The information is collected by monitoring an actual run of the application. Collecting information is not always possible. It involves substantial overhead and can hardly be used for prediction of the future behavior of objects. This is mostly useful for long term strategies like load balancing.

For the reasons given above only the textual analysis of the application code is discussed. Central to the distribution of an application is its decomposition into clusters through the specification of immutable collocation associations. A starting point for this decomposition is to look for object sets that are closed with respect to communication. In the object-oriented paradigm this translates into limitations on the flow of object-references in the system, also known as *aliasing* [1]. For a static immutable clustering of objects only invariant limitations can be considered. The simple heuristic considered here is found in one form or the other in nearly any distributed programming language: When an object only has one potential owner-object; i.e. one that owns a reference to it, then it is usually sensible to collocate them immutably. When an object has a few potential owner-objects that are all collocated immutably then it is again sensible to collocate the object immutably with its owner-objects. In this case a set of objects will be called an *island* following [12]. An Island can be more precisely defined as follows:

A set of objects that may use each other arbitrarily but cannot be used by objects not in this set —except one object called *gate*— is called an *island.* An island is called *open* if an object beside the gate object potentially uses an object not in the island, otherwise it is called *closed.*

An open island can be constructed starting from a given gate object as follows:

www.manaraa.com

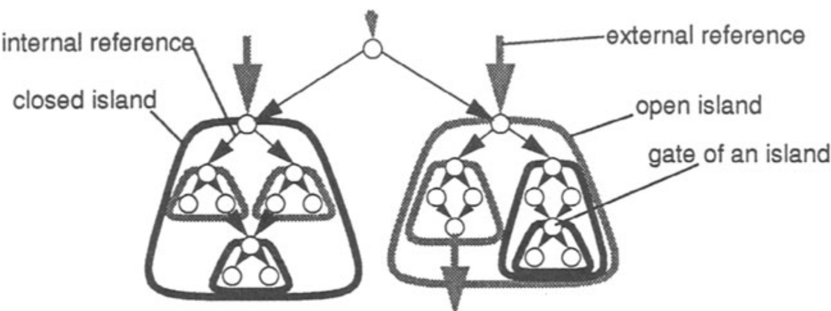


Fig. 3.3 A hierarchy of islands

- An object is called *private* if there is only exactly one potential owner-object during its lifetime. The transitive closure of *private* will be called *indirect private*. The gate object and all its private and indirect private objects are members of the initial set.
- All objects that can potentially only be used by the objects of the current set are added to the current set yielding a new set. This step is repeated until no more objects can be added. The resulting set is the island of the gate object.

The closed island for a given gate object can be constructed by successively eliminating objects that potentially use objects that are not in the island. The heuristic described above can now be used for islands: all objects in an island are collocated immutably. An island therefore corresponds to a (potential) cluster. The islands of an application system form a partial ordering with respect to set inclusion and define thus a hierarchical decomposition of the application into potential clusters. Figure 3.3 shows an example of such a hierarchy of open and closed islands.

The largest islands are used as a basis for the description of the actual decomposition by the programmer. Refinements are necessary when objects should be working on tasks in parallel or when the assumption underlying the heuristic does not hold, e.g. an object in an open island communicates more intensely with an external object than with the objects in its island.

3.5 Cooperations

The central goal of a dynamic distribution system is the "optimization" of the cooperation of objects. Objects cooperate by communicating or by concurrently working on a common task. The goal of the distribution system is to decrease the communication overhead by collocating objects and to increase the parallelism by dislocating objects.

The cooperation of a set of objects is modeled by dynamic *cooperation associations*, or short *cooperations*, to keep the concepts close to the object-oriented paradigm.The lifetime of a cooperation association for a set of objects is specified in terms of certain events in the application. The effects on the distribution are specified in terms of distribution associations on these objects.

www.manaraa.com

It is possible to infer simple cooperation associations by an automatic textual analysis of the application. One example is the cooperation between an *exclusive* object; i.e. an object with exactly one owner-object at any time, and its current owner-object. The effect of such a cooperation would be a mutable dynamic collocation of both objects, following the simple heuristic described in the previous chapter. It is not clear how further kinds of cooperations could be derived from an analysis or from a monitoring of the dynamic behavior of the application or how their lifetime and effects on the distribution could be inferred. A more precise description of cooperation scenarios as part of the programming language would improve the situation substantially.

The description of a cooperation association must specify the creation and destruction time, the involved objects, how to name them, the effects on the distribution associations of them as well as the type of objects it is applicable to. A cooperation class is used to describe the common properties of similar concrete cooperations that are then considered to be instances of this class. Cooperation classes can be bound to type-conform objects of the application. They can determine the cooperation this object can get involved in and therefore lastly its distribution behavior. Several cooperation classes may be statically or dynamically bound to an object. The description of a cooperation class consists of the following parts:

- *type of the cooperation class*
  The type of the cooperation class determines which objects it can be used upon. Objects of different type can observe different kinds of events. The type therefore impacts the lifetime specification.
- *lifetime specification of instances*
  The basic events of the application system are the creation and destruction of bindings to variables. The lifetime of a cooperation can be described as pairs of certain basic events; like the creation or the destruction of a binding of an object to an instance variable, the start or end of an invocation and even the creation or destruction of cooperations. The creation and destruction events can be made more specific by using conditionals on the state of the involved objects.
- *specification of the involved objects*
  The specification of a creation event not only has to determine the creation time of the cooperation but also the objects that are involved in it and a way to name them. An event like the begin of an invocation allows the naming of the caller, the callee and the argument objects. An event like the creation of a cooperation allows the naming of the objects involved in it.
  The objects in a cooperation play a certain *role* in it. The cooperation class describes all roles and how the objects actually involved are bound to them.
- *specification of the distribution associations*
  The effects of a cooperation on the distribution associations of the role objects must be specified. The resolution of conflicts and the acceptable migration-overhead for the creation and the maintenance of the distribution associations must be given.

Figure 3.4 shows the most important aspects of the creation and destruction of an instance of a cooperation class. A cooperation class bound to an object (called *ini-*

المنارة للاستشارات
www.manaraa.com

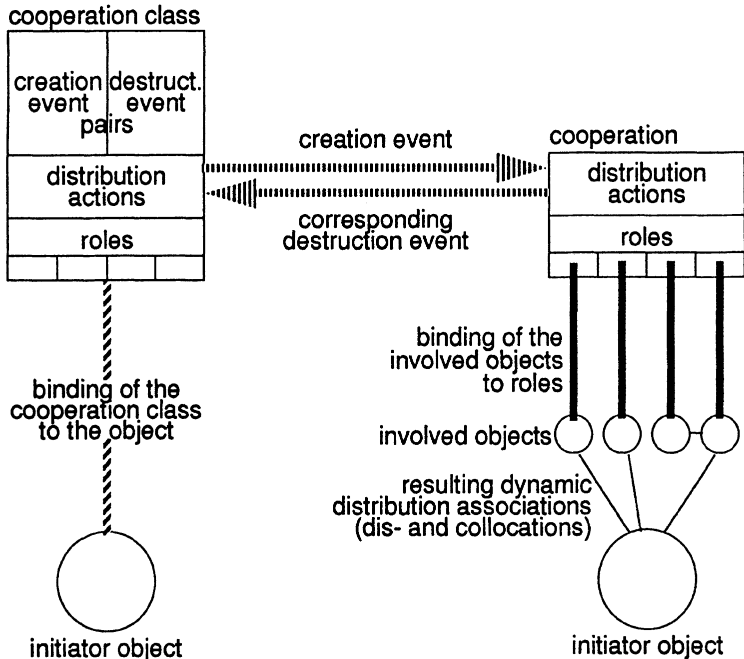


Fig. 3.4 Creation and destruction of a cooperation

tiator object) is instantiated if one of the creation events is observed within the object. The involved objects are bound to the roles and their distribution associations are created appropriately. When the destruction event that corresponds to the creation event is observed, all created distribution associations and the cooperation are destroyed. An object may be involved in several cooperations at a time and may even be their initiator.

4 Existing Approaches

Most existing distributed programming languages were created by extending an already existing programming language. The description of the decomposition into distribution units and their distribution is therefore part of the extensions made resulting in *non-uniform* distributed languages. The decomposition is achieved by explcitly declaring clusters, like *processor modules* in *Starmod* [8] or *guardians* in *Argus* [16], that can be assigned initially to nodes. This leads to the definition of relatively coarse-grain clusters that cannot be moved frequently because of the involved overhead. Mobility therefore is only considered for load balancing.

Object-oriented programming languages suggest a more fine-grained approach to distributed computing using *mobile* objects that can be migrated dynamically. *Emerald* was the first fully implemented language that pursued this approach. It uses a

www.manaraa.com

data-flow analysis to infer clusters and provides a sort of asymmetric collocation association called *attachment*. Objects can be migrated as an operation on it (*move*) or as part of the parameter-passing (*call-by-move*/*call-by-visit*) [14]. Systems like *Amber* [6] and [10] provide just run-time support.

MONACO [19] is an approach that allows at least a partially separated distribution description for object-oriented languages. *MONACO* is focussing on the automatic inference of class-based dynamic collocation rules by monitoring test runs of the application. Annotations in the classes are used to support this.

5 Conclusion

Developing distributed systems is a complex task. The distribution transparency of a distributed programming language reduces the complexity significantly but prevents using the distribution to optimize the cooperation and to achieve functional properties of the application. Existing approaches frequently give up distribution transparency and therefore reusability in favor of using the distribution. The adaptation of the distribution of an application to changes in the requirements of use or the distributed computer system can hardly be obtained.

The approach presented avoids these problems by using a separate orthogonal distribution language that does not compromise the distribution transparency of the programming language and therefore does not compromise reusability.

The generalization from the distribution of objects on nodes to the distribution on arbitrary loci, e.g. capsules, opens the possibility to describe a large number of functional properties or static and dynamic optimizations as long as they are orthogonal to the semantics of the application.

Currently the conceptual design of the distribution language has been completed. Its full description can be found in [11].

References

1. Hogg, John, Lea, Doug, Wills, Alan, deCahampeaux, Dennis, Holt, Richard, "The Geneva Convention on the Treatment of Aliasing - Followup Report on ECOOP '91 Workshop W3: Object-Oriented Formal Methods", *OOPS Messenger*, vol. 3, no. 2, pp. 11-16, April 92.
2. Black, A., Hutchinson, N., Jul, E., Levy, H., Carter, L., "Distribution and Abstract Types in Emerald", *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 65-76, Jan. 1987.
3. Bal, H.E., Steiner, J.G., Tanenbaum, A.S., "Programming Languages for Distributed Computing Systems", *ACM Computing Surveys*, vol. 21, no. 3, pp. 261-322, Sep. 1989.
4. Booch, G., *Object-Oriented Design with Applications*, Benjamin/Cummings, 1990.
5. Casavant, T.L, Kuhl, J. G., "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems", *IEEE Transactions on Software Engineering*, vol. 14, no. 2, pp. 141-154, Feb. 1988.

www.manaraa.com

6. Chase, J.S., Amador, F.G., Lazowska, E.D., Levy, H.M., Littlefield, R.J., "The Amber system: Parallel programming on a network of multiprocessors", *Proc. of the 12th ACM Symp. on Operating Systems Principles*, , Litchfield Park (AZ, USA), pp. 147-158, Dec. 1989.
7. Chen, P.-S. P., "The Entity-Relationship Model - Toward a Unified View of Data", *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 9-36, March 1976.
8. Cook, R.P., "*MOD-a language for distributed programming", *IEEE Transactions on Software Engineering*, vol. SE-6, no. 6, pp. 563-571, Nov. 1980.
9. Goldberg, A., Robson, D., *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.
10. El Habbash, A., Grimson, J., Horn, C., "Towards an efficient management of objects in a distributed environment", *Proc. of the Second International Symp. on Databases in Parallel and Distributed Systems*, Dublin, Ireland, 2-4 July 1990, pp. 181-190, Editors: Agrawal, R., Bell, D., IEEE Computer Soc. Pr., Los Alamitos, CA, USA, 1990.
11. Fäustle, M., *Beschreibung der Verteilung in objektorientierten Systemen*; Dissertation, Universität Erlangen-Nürnberg: IMMD; 1992
12. Hogg, J., "Islands: Aliasing Protection in Object-Oriented Languages", *Proc. of the Conference on Object-Oriented Programming, Systems, Languages, Applications, OOPSLA '91*, 6-11 Oct. 1991, Phoenix, Arizona, published as *ACM SIGPLAN Notices*, vol. 26, no. 11, pp. 271- 285, Nov. 1991.
13. Jul, E., Levy, H., Hutchinson, N., Black, A., "Fine-Grained Mobility in the Emerald System", *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 109-133, Feb. 1988.
14. Jul, E., *Object Mobility in a Distributed Object-Oriented System*, Ph.D. Thesis, Technical Report 88-12-06, Department of Computer Science, University of Washington (Seattle, WA 98195), Dec. 1988.
15. Kleinöder, J., *Objekt- und Speicherverwaltung in einer offenen, objektorientierten Betriebssystemarchitektur,* Dissertation, Universität Erlangen-Nürnberg: IMMD IV, 1992.
16. Liskov, B., "Distributed Programming in Argus", *Communications of the ACM*, vol. 31, no. 3, pp. 300-313, March 1988.
17. Popek, G.J., Walker, B.J., *The LOCUS Distributed System Architecture*, Computer Systems Series, The MIT Press, 1985.
18. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W., *Object-Oriented Modeling and Design*, Prentice-Hall International Editions, 1991.
19. Schill, A., "Mobility control in distributed object-oriented applications", *Proc. of the Eighth Annual International Phoenix Conference on Computers and Communications*, Scottsdale, AZ, USA, 22-24 March 1989, pp.395-399, IEEE Comput. Soc. Press, Washington, DC, USA, 1989.
20. Stankovic, J.A., Sidhu, I.S., "An Adaptive Bidding Algorithm for Processes, Cluster, Distributed Groups", *The 4th International Conference on Distributed Computing Systems*, San Fransisco, California, May 1984.

www.manaraa.com

21. Ungar, D., Smith, R.B., "Self: The Power of Simplicity", *Proc. of the Conference on Object-Oriented Programming: Systems, Languages, Applications, OOPSLA '87*, Orlando, Florida, published in *SIGPLAN Special Issue*, vol. 22, no. 12, pp. 227-242, Dec. 1987.
22. Williams, I., Wolczko, M., Hopkins, T., "Dynamic Grouping in an Object-Oriented Virtual Memory Hierarchy", *Proc. of ECOOP '87, the First European Conference on Object-Oriented Programming*, Paris, France, published as *Lecture Notes in ComputerScience,* eds. Goos, G.; Hartmanis, J., vol. 276, pp. 79-88, Springer Verlag, 1987.

www.manaraa.com

Towards the Implementation of a Uniform Object Model

Franz J. Hauck
hauck@immd4.informatik.uni-erlangen.de

University of Erlangen-Nürnberg
IMMD 4, Martensstraße 1
D-W 8520 Erlangen, Germany

Abstract. In most cases the programming models of distributed object-oriented systems have a two-stage or even a three-stage object model with different kinds of objects for values, distributable and non-distributable objects. This allows an efficient implementation for at least non-distributed objects, because traditional compilers can be used. This paper discusses some aspects of the implementation of a uniform object model that does not know of any distinction between distributable and non-distributable objects and allows an independent application description of the distribution of objects.

Instead of integrating distribution later into a non-distributed language our method takes the opposite approach. For the time being a distributed object model is implemented in a general, distributed, and, of course, inefficient way. With some additional information derived by user supplied declarations or automatically by a compiler the general implementation becomes more and more optimized. With assertions like *immutable* or *constantly* and *initially bound* objects the implementation can be optimized such that the non-distributed case is not worse than in traditional object-oriented languages.

1 Introduction

Usually distributed object-oriented systems are derived from non-distributed object-oriented languages, i.e. *Argus* from *CLU* [1], *Clouds* from *C++* – called *DC++* [2] – and from *Eiffel* – called *Distributed Eiffel* [3]. These approaches introduce a new kind of object which is distributable, all other objects of the base language used being non-distributable. The result is a two-stage object model with different semantics for each stage. The parameter-passing semantics depend on the kind of object passed, i.e. *call-by-copy* semantics (corresponds to *call-by-value*) for non-distributed objects and *call-by-reference* for distributed objects. Often, values form their own kind of objects resulting in a three-stage object model, e.g. in *DC++*.

This is an advantage because those object models are easy and quite efficient to implement, as the compiler of the base language can be used for all non-distributed kinds of objects and produces efficient code as well. Distributed objects are implemented by stub objects, which are non-distributed objects. These are passed by *call-by-copy* semantics which is in fact *call-by-reference* semantics for the distributed object.

www.manaraa.com

The different semantics for the different kinds of objects are the great disadvantage of these approaches. Often the kinds of objects are fixed by their respective classes, e.g. in *Argus*. There is a need for two classes, one for distributed and one for non-distributed objects. This impairs the reuse of code and it is not easy to see which semantics apply for a certain method call.

Another problem remains. The distribution of the objects is explicitly described in the program, thereby also affecting reusability. Yet, distribution is orthogonal to the programming of objects, with the exception of different failure models and different time behavior. Thus, we require a distribution description separate from the program description. To achieve this we need a uniform object model, as that is the only way to abstract completely from the distribution of objects.

Emerald is one approach with a uniform object model [4], but it has no separate description of distribution. Distribution is described inside an *Emerald* program with explicit statements.

This paper shows some aspects of implementations for uniform object models with a separate distribution description. Instead of extending a non-distributed language we choose the opposite method. In chapter 2 we introduce an object model for arbitrarily distributed objects. There is a separate description language for distribution. Chapter 3 presents a simple distributed implementation which is not very efficient. The efficiency is improved by suitable optimizations. Different aspects of the possible optimizations are shown. Chapter 4 contains a conclusion and ends with an outlook on future work.

These thoughts are originated from within the PM project of the IMMD 4, University of Erlangen-Nürnberg. The distributed object model is used there to model distributed operating systems.

2 A Distributed Object Model

This chapter introduces a distributed uniform object model for programming distributed applications or parts of operating systems. It defines *objects*, *methods*, *classes*, *types*, and *requests*.

2.1 Objects and Classes

Objects are the smallest distributable entities. They have behavior and state. The behavior is expressed with *methods*; operations which can be invoked by clients of an object. The state of an object *A* consists of references to other objects. Object *A* can be a client of all these objects and invoke any of their methods. References are bound to variables which can be accessed by the objects' methods only[1]. A graphical representation is shown in fig. 3.1. The rectangles represent objects, boxes represent variables and arrows represent references from variables to objects.

1. The access from a client to a variable can be allowed by using some special methods which can be derived automatically by a compiler.

www.manaraa.com

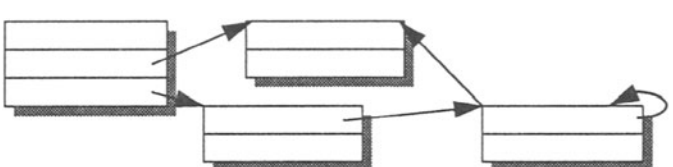


Fig. 2.1 A graphical representation of objects, variables and references

With the invocation of a method an independent virtual thread of activity is created executing the code of the method. After having done all computing and sending back the results to the client the thread of activity is destroyed. The thread of activity is called a *request*. During the execution of a method invocation all requests are seen as part of the state of the object which defines the method. The general parameter-passing semantics is *call-by-reference*. The *request* gets some references as parameters from the client and sends back some references as a result.

Classes are defined as sets of objects with the same behavior and identical structure of state. They constitute equivalence classes according to an equal relation of state and behavior. This extensional definition does not directly correspond to the definition of the term *class* in most programming languages. In a language the term *class* is almost intensionally defined. *Classes* are directly described and their objects are created from this description.

Languages like *Self* do not know classes as a concept of the language itself [5], but it is possible to identify the extensionally defined classes in these languages. For this paper it does not matter whether classes are concepts of a language or not. We refer in the next sections to the extensionally defined term.

2.2 Types

Types are equivalence classes of *classes* with the same abstract behavior. In most languages, like *C++* and *Eiffel*, the notion of type and class is the same, [6] and [7]. These languages do not know any mechanisms to declare two classes to be of the same type; not even when these classes are identically declared.

Type conformance is a subset relationship between types. This corresponds to the definition of Palsberg and Schwartzbach, [8] and [9]. Type conformance is also possible in *C++* and *Eiffel*, but only by using inheritance between classes. Inheriting classes are type-conforming to their base classes.

We propose that the notion of type is to be represented by some language concepts. A class[2] has to decide to which type it belongs. This results in an explicit separation of types and classes. It is necessary to make an explicit decision as to which type is supported by which class, because the compiler cannot decide which abstract behaviors are conforming.

2. Or an object (depending on if classes are defined as a language concept).

www.manaraa.com

2.3 Distribution

As mentioned above objects are the smallest entities for distribution. References can point to a distributed or to a local object. The method invocation on distributed objects is implemented by an RPC style communication. The semantics of a program is therefore independent of the possible distribution of the objects with the exception of different failure models and possible run-time differences. Thus, distribution is described in a separate programming model based on relative properties of distribution between the objects, called *collocations* and *dislocations*. Further information can be found in [10] and [11].

Objects can migrate arbitrarily in the system. This means that they can change their physical location at run-time, e.g. for an optimization of the total run-time of a program. The migration of objects is controlled by so called *cooperations*. These are part of the programming model of the distribution system and can be described there. Cooperations may dynamically create and destroy relationships of collocation or dislocation between objects and indirectly cause object migrations - see again [11].

3 Aspects of Implementation

One simple approach towards implementation of the described object model is exemplified in the following:

- Objects are implemented as continuous parts of memory containing all variables and methods.
- Each object has a unique identification called *OID* (object identifier), which has a system-wide uniqueness.
- References to other objects are stored in variables. These references are represented as *OID*s.
- Method invocations are handled by a run-time system which determines the actual position of an object by its *OID*.

An implementation like this is possible but certainly not efficient. The identifiers need a size of 96 to 128 bits to support a world wide distributed system. A variable needs 12 to 16 bytes of memory to store a reference. The search for objects will be very expensive, as objects may be located anywhere. Searching is also done for often used objects like integers or booleans.

It is immediately obvious that smaller identifiers would increase this efficiency. They would need less memory and allow faster searches for objects. The best case of an implementation of an OID is a pointer. There is no need for a search if the OID is a pointer to the memory space of an object. But in this case the object is not migratable without restrictions, as it must be located in the same address space.

As mentioned already an efficient implementation is possible with a multi-stage object model. Non-distributed objects are addressed by pointers and distributed objects are addressed by stub objects containing a large OID. These stub objects are only part of the implementation. This kind of implementation should be rejected be-

www.manaraa.com

Fig. 3.1 **Splitting the OID**

cause it cannot support a uniform object model. The following sections will show some aspects of optimizing the addressing of objects under certain circumstances.

Generally a more efficient implementation can be achieved by using smaller representations of the OIDs. The identifiers are to be stored in variables. But a variable is only bound to a subset of all objects during its lifetime. Thus, only the identifiers of those objects are to be stored in that variable. One approach could be to assign each variable a specific and different set of identifiers for all objects possibly bound to that variable. This is disadvantageous because assigning one variable to another is a more complex operation; the identifier of one object stored in one variable is different from the identifier of the same object stored in another variable. Trying to avoid all conversions of identifiers generally leads to a representation of identifiers which is as large as the unique OID.

Another drawback to using different identifiers for the same object is complex identity comparisons. They are needed as user level statements or as run-time statements for all kinds of aliasing-detection.

3.1 Constant Collocation

When the user specifies the distribution of an application in the distribution system there are objects which are constantly collocated; meaning their collocation relationship does not change during run-time. Using this information groups of objects can be derived which are all in a transitive collocation relationship. These groups are called *clusters* [11].

A simple way of using smaller identifiers is to split the OID into a Cluster ID (*CID*) and an identification local to a cluster (*LID*), see fig. 3.1. Inside a cluster local objects can be addressed by the LID only. The same objects are addressed from outside using the full OID. The CID allows a more efficient address resolution because there are fewer clusters than objects in the system.

Identity comparisons are simpler, because LIDs are part of the OIDs. References from inside a cluster assigned to variables outside are converted by simply adding the CID part to the LID.

The LID itself can be implemented by a pointer into the continuous memory space of a cluster. This pointer can even be an offset for some segmentation mechanism used to implement a position-independent addressing-scheme inside a cluster.

This kind of optimization can only be used for constantly collocated objects. Mobile objects form a cluster of their own and can only be addressed by a full OID. Variables inside a cluster which can be bound to objects of their own cluster and to objects outside of that cluster cannot use the efficient local addressing-scheme. They have to use the full OID address. The distinction between LID and a full OID would cause more overhead than the optimization of the local addressing would increase efficiency.

www.manaraa.com

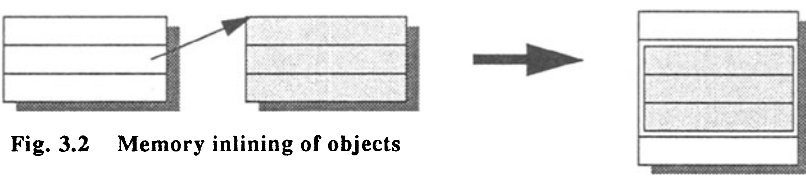


Fig. 3.2 Memory inlining of objects

3.2 Immutable Objects

Immutable objects only have constant bindings to other immutable objects. This implies that all their requests do not interfere. The result of a request depends only on its parameters. A request cannot store any information in an immutable object.

Immutable objects are easily shared. Shared immutable objects can be copied and each client can have its own copy without noticing it. The semantics of a program stay the same with shared or with copied immutable objects. Only correct identity comparisons between the copies must be realized. Immutable objects are candidates for a per cluster copy. Thus, they can always be addressed by a small LID. In our distribution system immutable objects can be collocated to two dislocated objects. This conflict is not solvable in the case of a mutable object.

This kind of optimization is very suitable for all kinds of value objects. Previously, in *CLU* and *Emerald,* values were modelled by immutable objects, [12] and [4]. There is no need to share integer values all over the system, as there can be copies for at least each cluster.

As the identifier of an immutable object has enough information about the object itself, there is no need to implement the object in the memory space in all cases. For further optimizations special identifiers, called *VID*s (value IDs), can be used for binding immutable objects. These identifiers do not address certain objects (in the implementation), they stand for them. VIDs can be made unique within the system and thus there is no need for a conversion when assigning a reference from one cluster to another. The representation of a VID can be very special, e. g. for integers it may be a bit-pattern. This bit-pattern allows for direct integer arithmetic operations.

3.3 Constantly and Initially Bound Variables

A variable is initially bound when a reference is bound at creation time. Constantly and initially bound variables are bound at the object creation. This binding cannot be changed. Such a situation can be used for an optimization called memory inlining. The variable does not contain an identifier of the bound object but the object itself (fig. 3.2). In the distributed object model this optimization is only possible with an additional constantly collocated relationship between the objects.

In *C++* all non-pointer variables are initially and constantly bound references. In *Eiffel* these kinds of objects are called *expanded objects*. In both languages the problem lies with the different kinds of objects having different parameter-passing semantics for non-pointers or expanded objects respectively; *call-by-copy* instead of *call-by-reference*. In our approach the memory inlining is a matter of an implementation and is independent of all semantics of the object model. There is no need to

www.manaraa.com

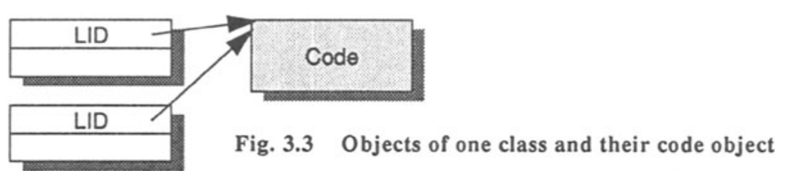


Fig. 3.3 Objects of one class and their code object

declare initially and constantly bound references at all, because they may also be derived by a compiler.

Inside a cluster inlined objects can be addressed by a LID. Outside of a cluster the usual OID is used. Initially and constantly bound, immutable objects can be treated like plain immutable objects; they can be copied freely. In this special case the new properties can be used to place the VIDs directly in the code and not necessarily in the variables. This procedure is the same as that done by compilers which implement constants integrated into machine instructions and which are not loaded from memory. Variables initially and constantly bound to immutable objects may not be represented in the memory space of the object when this property is known to all clients. This is the case, at least, for the code of the object's methods.

3.4 Classes

Classes are defined extensionally in chapter 2. Thus, they do not need a representation at run-time. Each object has its own methods, because only its own methods can access its local variables. In an implementation it does not make sense to have all objects with their own method code. The grouping of equal methods is recommended for efficient memory usage. The access of the method code to the variables of an object is implemented by segmentation mechanisms like index registers or hidden parameters.

The method code of the objects belonging to one class can be implemented by a special object visible to the implementation only. Each object of the class has a reference to that code object. It represents the class (fig. 3.3).

The code of a class is normally immutable. Thus, the code can easily be copied in each address space and be addressed locally. This is a must for executable code. The code object has a strongly collocated relationship to all the objects using the code. Because it is immutable, dislocation conflicts between these objects can be solved by different copies of the code.

The reference from each object to the code object is initially and constantly bound. Thus, there is no need for the reference when all clients know which class an object belongs to. Then the code of a method can be directly addressed by a client. The reference to the code object can be omitted when all clients know the class of the object.

www.manaraa.com

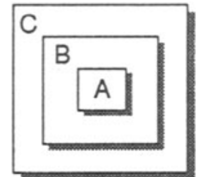


Fig. 3.4 **Class *C* inheriting from *B* and *A***

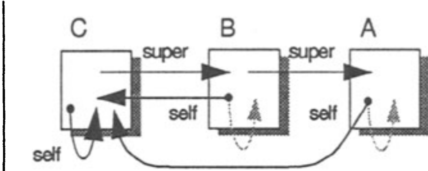


Fig. 3.5 **Bindings between differential classes**

3.5 Inheritance

Inheritance is a concept of composing classes. As types are separated from classes we need some kind of inheritance for both of them. Inheritance for types may be a concept of creating conforming types. This does not lie within the scope of this paper.

This section deals with class-based inheritance, but not subtyping. Inheritance between classes is used for refining and reusing the behavior of some base classes into subclasses. In most languages the subclass is seen as one unit containing the inherited properties of the base classes. See fig. 3.4 for a graphical representation of class *C* inheriting from class *B*, which inherits from class *A*.

We propose another view to inheritance. We identify subclasses as differential classes which describe all the changes of and refer to the base classes. Thus, the subclasses do not include properties of the base classes, they only refer to them. In fig. 3.5 the same situation seen in fig. 3.4 is presented, but the boxes represent differential classes. The objects of the classes in an inheritance hierarchy refer to each other by a reference. These references correspond to keywords like *self* and *super*[3]. *Self* refers to the last subclass which inherits no further. *Super* refers to the base class. These references are represented in fig. 3.5 by arrows. Grey arrows are *self* references which are rebound by the inheritance mechanism.

Using references of the objects of inheriting classes to model inheritance has its advantages for implementation. There is no need for a special implementation mechanism to deal with inherited classes. The implementation treats differential classes as normal classes. The creation of objects of these classes also causes the creation of objects of the base classes with certain bindings for their *self* and *super* variables. The bindings are initially and constantly bound and all objects are collocated, i.e. the bindings are addressed by LIDs and memory-inlining can be done. This is quite the same implementation as in traditional languages, e.g. *C++*.

In our model there is no need to collocate the objects. A dislocation leads to a distributed inheritance mechanism, which allows the distribution of some parts of an object. This is possible because there is not only one object of an inheriting class, but many objects of many base or differential classes respectively.

3. We adopted these names from the *Smalltalk* terminology [13].

www.manaraa.com

3.6 Type Checking

Like classes types do not necessarily need a run-time representation. Types are introduced to get programs type checked, i.e. all method calls are legal and the well-known *Smalltalk* error message, "message not understood", will not occur.

Compilers can do most of the type checking at compile-time, but there are several circumstances which need a run-time type checking, e.g. for *type-case* and *type-of* statements and for binding references to a variable with larger type. In a distributed environment there may be objects with types which are not even known at compile-time. To initialize a distributed application the run-time system needs to type check all references to objects with these unknown types. For all these cases of run-time type checking a representation of types at run-time is necessary.

Therefor we introduce one more set of auxiliary objects for types. All objects have a reference to a type-object. This is constantly and initially bound and immutable. Thus, it is subject to the above-mentioned optimizations.

The type objects have a method for type comparisons as in *Emerald* [4]. These methods and the unique identification of type-objects are used for run-time type checking.

4 Conclusion

We have shown that in some circumstances the implementation of a uniformly distributed object model can be optimized. This optimization is as efficient as non-distributed language implementations when the application is not distributed, because the implementation places all objects into one cluster and collocates them. Internal pointers are avoided by memory-inlining when objects are initially and constantly bound. Immutable objects allow the inlining of object references in the method code. This corresponds to compiled programs in a non-distributed language.

A distributed application is obviously not as efficient as a non-distributed application when we only look at the addressing of objects and memory usage. The above-mentioned optimizations are a step towards an optimal implementation.

To validate the practical use of the optimizations the construction of a prototypical compiler is planned. This compiler will output, with specific distribution descriptions, the possible implementation optimizations of specific example applications.

References

1. B. Liskov, "The Argus language and system"; In: *Distributed systems, methods, and tools for specification*; M. Paul, H. J. Siegert [Eds.], Lecture Notes in Comp. Sci. 190; Springer, Berlin; 1985 – pp. 343-430
2. P. Dasgupta, R. Anathanarayanan, S. Menon, A. Mhindra, R. Chen: *Distributed programming with objects and threads in the Clouds system*; Tech. Report GIT-CC 91/26, Georgia Inst. of Techn., Atlanta GA; 1991

www.manaraa.com

3. L. Gunaseelan, Richard J. Jr. LeBlanc: *Distributed Eiffel: a language for programming multi-granular distributed objects on the Clouds operating system*; Georgia Inst. of Techn., Tech. Rep. GIT-CC-91/50; Atlanta GA, 1991
4. N.C. Hutchinson, R.K. Raj, A.P. Black, H.M. Levy, E. Jul: *The Emerald Programming Language*; Univ. of Washington, Seattle WA, Tech. Report 87-1υ-07; Oct. 1987, revised Aug. 1988
5. D. Ungar, R.B. Smith: "Self: The power of simplicity", In: *Proc. of the Conf. on Obj.-Oriented Progr. Sys., Lang., and Appl.*; N. Meyrowitz [Ed.], (Orlando FL, Oct. 4-8, 1987); SIGPLAN Notices 22(12); ACM, New York NY; Dec. 1987 – pp. 227-242
6. M.A. Ellis, B. Stroustrup: *The annotated C++ reference manual - ANSI base document*; Addison-Wesley, Reading MA, USA; 1990
7. B. Meyer: *Eiffel: the language*; Prentice Hall, New York NY; 1992
8. N. Oxhøj, J. Palsberg, M.I. Schwartzbach: "Making type inference practical"; In: *Proc. of the 6th Eur. Conf. on Obj.-Oriented Progr.*; O. Lehrmann Madsen [Ed.], (Utrecht, June 29-July 3, 1992); Lecture Notes in Comp. Sci. 615; Springer, Berlin; June 1992 – pp. 329-349
9. J. Palsberg, M.I. Schwartzbach: "A note on multiple inheritance and multiple subtyping", In: *Multiple inheritance and multiple subtyping* - Pos. Papers of the ECOOP '92 Workshop W1; M. Sakkinen [Ed.], (Utrecht, June 29,1992); Tech. Rep., Dep. of Comp. Sci. and Inf. Sys., Univ. of Jyväskylä, Finland; May 1992 – pp. 3-5
10. M. Fäustle: *Beschreibung der Verteilung in objektorientierten Systemen*; Diss., IMMD, Univ. Erlangen-Nürnberg; 1992
11. M. Fäustle: "An orthogonal optimization language for uniform object-oriented systems"; *Parallel Comp. Architectures: Theory, Hardware, Software, and Appl.* – SFB Colloquium SFB 182 and SFB 342; A. Bode, H. Wedekind [Eds.], (Munich, Oct. 8-9, 1992); Lecture Notes in Comp. Sci.; Springer, Berlin; to appear 1992
12. B. Liskov, J. Guttag: *Abstraction and Specification in Program Development*; MIT Press, Cambridge MA; 1986
13. A. Goldberg, D. Robson: *Smalltalk-80: the language and its implementation*; Addison-Wesley, Reading MA, USA; 1983

www.manaraa.com

FOCUS: A Formal Design Method for Distributed Systems

Frank Dederichs, Claus Dendorfer, Rainer Weber
Institut für Informatik der Technischen Universität München
Postfach 20 24 20, D-8000 München 2, Germany

E-Mail: dedericf,dendorfe,weberr@informatik.tu-muenchen.de

Abstract

FOCUS is a formal method for the development of distributed systems. It covers all the way of formal system development in three abstraction levels: requirements specification, design specification, and implementation. This paper gives a short, informal account of the method FOCUS and the rationale behind some of its features. The application of FOCUS is illustrated by a simple example.

1. Introduction

The formal method FOCUS aims at the development of distributed systems. It covers the whole development process by providing formalisms and methods for the phases of

- requirements specification,
- design specification, and
- implementation.

In order to span such a wide range of abstraction levels, a number of distinct formalisms are used. To specify the initial system requirements we use the formalism of *trace specifications*. Such a specification describes the allowed runs of a distributed system from a global perspective. A trace specification is non-constructive, i.e. it describes only the allowed system behaviour and not how it may be achieved.

In a design specification, we use the modular and compositional formalism of *stream-processing functions*. The initial design specification is derived from the requirements specification and is usually not executable. Then step-by-step design decisions like modularization by functional decomposition are taken until finally a functional program is obtained.

www.manaraa.com

The functional program can be seen as a particular functional specification which has the property of being executable; it is an *implementation* of the distributed system. For reasons of efficiency and availability on distributed hardware the functional program may be transformed into an imperative program. In FOCUS, there are two experimental languages for representing programs: AL and PL. The first one is an applicative language, while the second one is procedural. Both are tailored for the methodological and technical framework of FOCUS.

Currently, numerous formalisms for the description of distributed systems are proposed, for example, temporal logics [11], Petri nets [16], CCS [14], CSP [8], I/O-automata [9], Statecharts [7] and so on. However, the integration of these formalisms into a methodological framework that supports their goal-oriented use has only recently attracted more attention. Among the approaches that emphasise methodological aspects are Lamport's transition axiom method [12], Unity from Chandy and Misra [4] as well as the Esprit Projects Procos [15] based on process algebras and Demon [17] based on Petri nets. Furthermore there are a number of object oriented approaches [5], [13], which strive for the same aim but are less formal.

All these methods have their advantages and disadvantages. Their methodological merits can only be assessed when they are applied. A notable comparison of different design approaches including (a predecessor of) FOCUS can be found in [18].

With respect to the formal basis FOCUS is distinguished form the methods mentioned above in that it places the emphasis on functional techniques. The formalisms of trace specification, functional specification, functional and imperative programs are chosen to reflect the abstraction levels of the development phases. For example, initially it is appropriate to have a global view of a system instead of thinking about it in terms of its processes. For the subsequent development, the functional setting provides a more modular description technique. These formalisms are well-integrated, and guidelines and transformation rules are provided for the transition between consecutive levels. It is necessary to make these transitions as straightforward as possible. The main work of program development is done within the individual abstraction levels. Here the creativity of the system designer is needed. However, some heuristic design principles can be provided for the refinement within each abstraction level.

Each of the sections 2 to 4 is devoted to one of the three abstraction levels of requirements specification, design specification, and implementation. Within each section, we first give an informal overview of the level, then we motivate why we consider the suggested formalism and technique appropriate for the particular phase, and finally we give an illustrative example. The examples are simplified "snapshots" of a protocol development for a communication system, where a transmitter and a receiver component are designed to guarantee reliable communication over an unreliable medium. The complete development can be found in [6]. A much more detailed description of FOCUS is in [2]; for a summary of case studies see [3].

المنارة للاستشارات

www.manaraa.com

2. Requirements Specification

2.1 Overview

Trace specifications describe the behaviour of distributed systems in a very abstract and property-oriented way. Thus they are well-suited for formalising requirements.

Technically, a trace specification describes the allowed runs of a distributed system by giving a set of finite and infinite sequences of actions. *Actions* are atomic entities that represent the basic activities in a system, like "pressing a button" or "sending a message". A *trace* can be seen as the observation of a complete system run from a global perspective. For example, the behaviour of a communication system is specified by the valid sequences of send and receive actions. In a trace specification the allowed system runs are characterised by stating which *properties* have to be fulfilled. For this purpose predicate logic formulae are used.

In many cases a reactive system can only be described properly together with the environment in which it is embedded. Then also the environment is modelled in the requirements specification. Technically, there is no difference in the description of the system components and the environment. Methodologically, however, there is: the system has to be implemented during the development process whereas the environment is supplied by the customer. Hence we will talk about two kinds of components in our model: *system components* to be implemented and *environment components* to be supplied by the customer.

Often it is a fundamental requirement of the customer that the overall system is structured into several components. An example for this is a communication system, where it is a necessity that the transmitter and receiver are separate components (see Fig. 2).

Summing up, a requirements specification includes:

- *Global requirements*
  These are requirements on the whole system expressing properties of its global behaviour.
- *Description of the components' syntactic interface*
  Here it is fixed which components the system should consist of, which components are to be implemented and which are environment components. Furthermore it is stated what the input and output actions of each component are.
- *Environment assumptions*
  These are properties of the environment components. The system designer may assume these properties to hold.

In this form, a requirements specification describes the overall task of system development in a concise way: develop components with appropriate input and output actions, such that the composition of these components with the environment components satisfies the global requirements on the whole system, provided the environment components stick to their assumptions. On the trace level, the task of the system designer is to find a so-called *local*

www.manaraa.com

specification for each component to be implemented. A local specification describes the behaviour of a single component. These local specifications will then be refined into parallel programs in the subsequent development phases.

2.2 Rationale

We made the observation that during the initial phase of a system development it is often easier to state some "global objectives" of how the system should behave rather than stating particular explicit requirements for each system component. Moreover, in this phase it is easier to state requirements for the system runs rather than to give an already operational, executable specification. The task of stating global objectives in an abstract, non-operational way can be accomplished with trace specifications. Moreover, due to the use of predicate logic, trace specifications have a great expressive power, which is an essential property for a requirements specification formalism.

In contrast to related formalisms, in particular the trace formalism in [8], we use both finite and infinite traces. Infinite traces are needed to model arbitrary liveness properties of possibly nonterminating systems. For example, a liveness property of a communication system is that every message sent will eventually be delivered.

2.3 Notation and Example

In a trace specification we specify the allowed traces using predicate logic formulae. A trace specification has the form $P_1(t) \wedge ... \wedge P_n(t)$, where t is a trace variable and each $P_i(t)$ is an (auxiliary) predicate on traces. Auxiliary predicates are introduced to structure a specification into several less complex requirements. When defining trace predicates, the usual logical connectives are used. Each trace predicate represents a *property* of the system behaviour. We distinguish between *safety* and *liveness* properties. Informally, safety properties exclude undesired effects, liveness properties guarantee desired effects. More precisely, a safety property is a property whose violation can always be detected after a finite amount of time, i.e. by considering a sufficiently large finite prefix of a trace. A liveness property is a property whose violation can only be detected after a complete, possibly infinite observation, i.e. by considering a complete, possibly infinite trace. See [12] for further aspects of safety and liveness.

To illustrate our approach we use the example of a communication system (see Fig. 1).

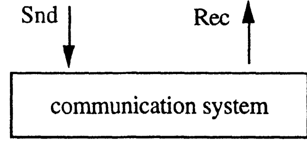


Fig. 1: Global view of the communication system

www.manaraa.com

Here, Snd is the set of send actions, Rec is the set of receive actions. The action snd(d) models that the datum d is sent from the environment to the communication system, while rec(d) models that it is received by the communication system from the environment. Formally:

$$\text{Snd} \equiv \{\text{snd}(d) \mid d \in D\} \qquad \text{Rec} \equiv \{\text{rec}(d) \mid d \in D\}$$

are the action sets considered here. D is a (predefined) set of "user data".

It is the overall purpose of the communication system to establish a reliable communication line. This is expressed by the properties US and UL, which can be seen as the *global requirements* of this specification.

US(t) denotes that t is an allowed trace iff for all prefixes of t the sequence of received data is a prefix of the sequence of sent data. In other words: only those data are received that have been sent, and this happens in the correct order. US is a safety property.

$$\text{US}(t) \equiv \forall s \sqsubseteq t : \text{dt}(\text{Rec} \copyright s) \sqsubseteq \text{dt}(\text{Snd} \copyright s)$$

The symbols $\sqsubseteq$, © and dt denote predefined operations. $s \sqsubseteq t$ denotes that s is a prefix of t. © is a filter operation. Rec © t filters the actions in the set Rec from t. For example, $\text{Rec} \copyright \langle \text{rec}(d_1), \text{snd}(d_2), \text{rec}(d_3)\rangle = \langle \text{rec}(d_1), \text{rec}(d_3)\rangle$. The function dt projects the data component, e.g. $\text{dt}(\langle \text{rec}(d_1), \text{snd}(d_2), \text{rec}(d_3)\rangle) = \langle d_1, d_2, d_3\rangle$.

UL(t) denotes that finally as many receive actions as send actions occur. UL is a liveness property.

$$\text{UL}(t) \equiv \#(\text{Rec} \copyright t) = \#(\text{Snd} \copyright t)$$

#t denotes the length of t (which may also be infinite: ∞). So, $\langle \text{snd}(d_2), \text{snd}(d_3), \text{rec}(d_2), \text{snd}(d_4), \text{rec}(d_3), \text{rec}(d_4)\rangle$ is a correct trace. The trace $\langle \text{rec}(d_2), \text{snd}(d_2)\rangle$ is not allowed (because it violates US), and neither is $\langle \text{snd}(d_2), \text{rec}(d_2), \text{snd}(d_3)\rangle$ (because it violates UL).

It seems as if these requirements could be achieved quite easily: design an agent that realizes the identity function. However, there is the additional constraint that the transmitter and the receiver must not communicate directly, but only via a given medium. Thus, the communication system must be structured as in Fig. 2:

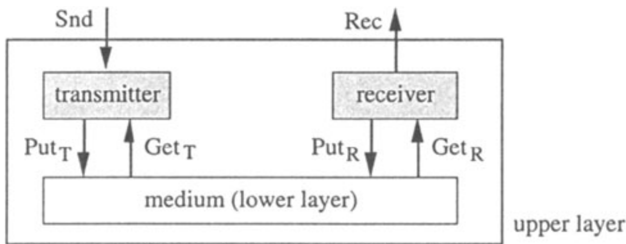


Fig. 2: The inner structure of the communication system

www.manaraa.com

This situation is typical for protocol design: the medium represents a lower layer of a protocol hierarchy. It provides unreliable data transfer. It is our task to develop a transmitter and a receiver component that turn the unreliable transfer into a reliable one.

Thus the communication system consists of three components: the transmitter and the receiver, which are to be implemented, and the medium, which is an environment component. We will now describe the components' syntactic interfaces. Four additional action sets are considered for the interaction of the transmitter and the receiver with the medium:

$$Put_T \equiv \{put_T(x) \mid x \in \mathbb{N} \times D\} \qquad Put_R \equiv \{put_R(y) \mid y \in \mathbb{N}\}$$

$$Get_T \equiv \{get_T(y) \mid y \in \mathbb{N}\} \qquad Get_R \equiv \{get_R(x) \mid x \in \mathbb{N} \times D\}$$

The reason why x is from $\mathbb{N} \times D$ and y is from $\mathbb{N}$ is explained below.

The assumptions for the medium, the *environment assumptions,* are the properties LS and LL. LS(t) denotes that anything the transmitter gets from the medium must have been put to the medium by the receiver and vice versa. LL(t) denotes that if the transmitter puts infinitely many data items to the medium, then infinitely many data items are delivered to the receiver and vice versa.

$$LS(t) \equiv \forall s \sqsubseteq t : (get_T(x)\ \mathit{in}\ s \Rightarrow put_R(x)\ \mathit{in}\ s) \wedge (get_R(x)\ \mathit{in}\ s \Rightarrow put_T(x)\ \mathit{in}\ s)$$

$$LL(t) \equiv (\#(put_T(x) © t) = \infty \Rightarrow \#(get_R(x) © t) = \infty) \wedge$$
$$(\#(put_R(x) © t) = \infty \Rightarrow \#(get_T(x) © t) = \infty)$$

Hence, the medium is unreliable in the sense that it may interchange the position of sent data and it also may lose data. However, if something is sent often enough it finally gets through. This is ensured by LL.

A first step in the development of the transmitter and the receiver is to develop local specifications for these components. Technically, such a specification is a trace formula that only talks about input and output actions of one component.

After several refinement steps (see [6]), we arrive at local specifications T for the transmitter and R for the receiver, such that

$$T(t) \wedge R(t) \wedge LS(t) \wedge LL(t) \Rightarrow US(t) \wedge UL(t)$$

i.e. the requirements T and R together with the assumptions LS and LL about the medium imply (satisfy) the global requirements US and UL. In our setting, refinement simply means implication of trace predicates.

The implementation we are going to develop in the sequel is based on an idea from [19]. It makes use of so-called sequence numbers. The transmitter attaches a unique sequence number to every message obtained from the environment. This is the reason why in $put_T(x)$ the x is from $\mathbb{N} \times D$. On the other hand, when the receiver gets a message $get_R(\langle k,d \rangle)$ from the medium, it gives the data part d to the environment and puts the sequencs number k back to the medium to acknowledge the receipt of the message. The medium transfers the

www.manaraa.com

sequence number back to the transmitter. This is reason why the y in $get_T(y)$ is from $\mathbb{N}$. The transmitter can be sure that the message with sequence number k has been transmitted if it receives k as acknowledgement from the receiver. Of course also acknowledgements may get lost.

The specification of the transmitter reads as follows: $T \equiv TS \wedge TL$ where TS and TL are defined below: TS(t) denotes that the datum d put to the medium by the transmitter with the sequence number k is actually the k-th datum sent by the environment. TL(t) expresses that if at least k send actions occurred and the k-th message has not received its acknowledgement yet, then some message must be sent infinitely often.

$$TS(t) \equiv \forall s \sqsubseteq t : put_T(\langle k,d \rangle) \; \mathit{in} \; s \Rightarrow (Snd © s)[k] = snd(d)$$

$$TL(t) \equiv \#(Snd©t) > k \wedge \neg get_T(k) \; \mathit{in} \; t \Rightarrow \exists j, d : \neg get_T(j) \; \mathit{in} \; t \wedge \#(put_T(\langle j,d \rangle)©t) = \infty$$

Here s[k] denotes the k-th element of the stream s.

This local specification, which consists only of local requirements for the transmitter, is the basis for the subsequent design. Since Focus is a modular and compositional formalism the development of the transmitter and the receiver can be done independently.

3. Design Specification

3.1 Overview

Once the overall structure of the system components is fixed and the (local) requirements for each component have been determined, we switch to a component-oriented specification technique such that modular refinement becomes possible. The system is seen as a collection of components called *agents*, which are connected by unidirectional *communication channels*. As an example for an agent, consider the transmitter of our communication system (see Fig. 3).

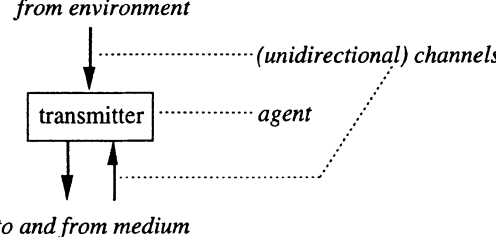


Fig. 3: The transmitter as an agent on the functional level

It is an important feature of Focus that each of these agents can be developed independently, based on the local requirements specification. Such a decomposition process can be applied

www.manaraa.com

repeatedly, i.e., not only the system is split into a collection of agents, but also each agent can be split into several subagents in order to add further structure.

In such a modular system description, every agent is an independent entity. Because the only connection with its environment is via the communication channels, the inner structure of the other agents need not be taken into account.

Agents communicate in an *asynchronous* way. Operationally, one may imagine that the channels are first-in-first-out buffers of infinite capacity, such that an agent may always put some message into a channel without being blocked or delayed. On a more formal level, the communication history of each channel is represented by a *stream*, i.e. a finite or infinite sequence of messages. Agents are represented by *stream-processing functions*, and the whole network is defined by the functional composition of its components (feedback loops are modelled using fixpoint-techniques). See [1] for a detailed description.

The specification of a system contains the specification of the individual agents and that of the network structure. An agent is specified by a predicate on stream processing functions. The network structure is specified either as a set of (mutually) recursive stream equations or by special composition operators.

The connection between a trace specification and the functional specification is that each message put onto a communication channel constitutes an action. Each trace that is generated by one of the specified functions for some input must be allowed by the trace specification.

On the functional level, the development process starts with a (usually very implicit) specification of the agent, which is derived from the local trace specification.

During the development process specifications are refined, the predicates are transformed and design decisions are incorporated. In every such design step, the specification is strengthened or simply rewritten. The aim is to obtain an explicit description of the stepwise computation process, which can then be transformed into an implementation.

3.2 Rationale

The design specification level has three main ingredients:

- a model for distributed, communicating systems,
- a specification formalism for such systems, and
- a refinement method for the transition from implicit to implementable specifications.

The system model must represent distributed, communicating systems. It should be carefully balanced such that it has both a sound mathematical basis and is close to a practical implementation, especially on distributed hardware. We think that this is true for the FOCUS system model of agents that communicate asynchronously via point-to-point channels. It has been shown in [10] that networks of such agents can be seen as stream processing functions, which yields a compositional denotational semantics. The connection to an implementation can either be made using functional programming languages or the denotational semantics of an imperative programming language (see Section 4).

www.manaraa.com

The specification of systems by predicates on stream processing functions has the advantage that the well-known formalism of predicate logic can be used. Stepwise refinement and the incorporation of design decisions is straightforward by strengthening predicates, e.g. by adding conjuncts. As on the trace layer the refinement relation is simply the implication of predicates.

Refining specifications in order to obtain more constructive (implementable) formulae can be seen as *vertical refinement.* In addition, it is possible to use functional decomposition not only to express spatial or logical distribution, but also as a means of *horizontal refinement*. Decomposing an agent into several subagents leads to more modular and structured system descriptions.

3.3 Notation and Example

The network in Fig. 2 can be specified by the following set of stream equations. In these mutually recursive formulae, the input and output channels are called in and out, respectively, and x, y, z, v are local channels between the system components:

$$x = \text{transmitter}(in, y)$$

$$(y, z) = \text{medium}(x, v)$$

$$(out, v) = \text{receiver}(z)$$

In a certain sense the behaviour of the transmitter is time dependent. For instance, it sometimes has to repeat the sending of a message $\langle k,d \rangle$ until it finally receives k as acknowledgement. To specify a behaviour like this from now on we assume that all input streams of the transmitter carry timing information. This is modelled by a special message $\surd$ (pronounced "tick"). Therefore the transmitter has the following functionality:

$$\text{transmitter: } (Snd \cup \{\surd\})^{\infty} \times (Get_T \cup \{\surd\})^{\infty} \rightarrow Put_T{}^{\omega}$$

A $\surd$ in an input stream represents the situation that a time unit has passed but no "real" message occurs. Since time cannot stop, it is consistent with this modelling that all streams that contain ticks are infinite (often they contain an infinite number of ticks). One can assume that the environment and the medium are required to include $\surd$ in their output streams.

The transmitter is specified as follows: The predicate TS' expresses that the transmitter may only output an message $\langle k,d \rangle$ if the k-th message of the environment has been d. TS' looks quite similar to last section's corresponding predicate TS on traces.

$$TS'(f) \equiv \forall x \in (Snd \cup \{\surd\})^{\infty} : \forall y \in (Get_T \cup \{\surd\})^{\infty} :$$
$$\langle k,d \rangle \textit{ in } f(x,y) \Rightarrow (Snd \copyright x)[k] = snd(d)$$

The predicate below states the following: suppose the transmitter receives an input stream that contains more than k messages from the environment, but not the acknowledgement to the k-th message. Then there must be a pair $\langle j,d \rangle$ that has not been acknowledged so far and that is sent infinitely often.

www.manaraa.com

$$TL'(f) \equiv \forall x \in (Snd \cup \{\surd\})^{\infty} : \forall y \in (Get_T \cup \{\surd\})^{\infty} : \\ \#(Snd © x) > k \land \neg get_T(k) \; in \; y \Rightarrow \\ \exists j, d : \neg get_T(j) \; in \; y \land \#(put_T(\langle j,d \rangle) © f(x,y)) = \infty$$

In fact, the predicates TS' and TL' can be obtained from TS and TL by a schematic (syntactic) transformation. Note that due to our assumption above we only talk about infinite input streams. The sets Snd, Get_T, Put_T are defined in the previous section.

The developer's task on the functional level is to strengthen the predicates with the goal of eventually arriving at a formula that fulfils certain syntactic constraints so that it can be considered executable. This is a process that requires the developer's creative imagination. It generally includes different steps, such as:

- functional decomposition, i.e. splitting an agent into a network of subagents in order to modularize the system,
- making design decisions by adding conjuncts to the specification (for example, in which order will the transmitter send the non-acknowledged pending messages),
- rewriting the specification such that the reaction of an agent to some newly arriving input is made explicit, i.e. giving a specification of the form $f(z \circ i) = f.z \circ h(z \circ i)$ for some suitable auxiliary function h,
- constructing an explicit state space, which is easy to access and update (for example, a state space that contains the next valid sequence number and a list of pending messages).

The first step mentioned above is horizontal refinement, while the other ones are vertical refinement steps. Note that the second step still describes the intended design in a very abstract way, while the third and fourth step aim towards an reasonably efficient implementation.

After all these design steps are made, we arrive at an executable functional specification, which can be seen as a functional program. At the end of this stage, the transmitter may, in a typical functional programming style, look as follows:

$$transmitter = trans(1, \langle\rangle)$$

where trans is defined by

$$\begin{aligned}
trans(n, \langle\rangle)(\surd \circ x, \surd \circ y) &= trans(n, \langle\rangle)(x, y) \\
trans(n, \langle\rangle)(snd(d) \circ x, \surd \circ y) &= trans(n+1, \langle n,d \rangle)(x, y) \\
trans(n, \langle\rangle)(\surd \circ x, get_T(k) \circ y) &= trans(n, \langle\rangle)(x, y) \\
trans(n, \langle\rangle)(snd(d) \circ x, get_T(k) \circ y) &= trans(n+1, \langle n,d \rangle)(x, y) \\
trans(n, \langle k,d \rangle \circ q)(\surd \circ x, \surd \circ y) &= \langle k,d \rangle \circ trans(n, \langle k,d \rangle \circ q)(x, y) \\
trans(n, \langle k,d \rangle \circ q)(snd(e) \circ x, \surd \circ y) &= \langle k,d \rangle \circ trans(n+1, \langle k,d \rangle \circ q \circ \langle n,e \rangle)(x, y) \\
trans(n, \langle k,d \rangle \circ q).(\surd \circ x, get_T(j) \circ y) &= \langle k,d \rangle \circ trans(n, remove(j, \langle k,d \rangle \circ q))(x, y) \\
trans(n, \langle k,d \rangle \circ q)(snd(e) \circ x, get_T(j) \circ y) &= \langle k,d \rangle \circ trans(n+1, remove(j, \langle k,d \rangle \circ q) \circ \langle n,e \rangle)(x,y)
\end{aligned}$$

www.manaraa.com

Here the first parameter n represents the first unused sequence number and the second parameter q represents the queue of message which have already been sent but not yet acknowledged. The auxiliary function remove is used to delete messages from q when acknowledgements arrive. It is defined as follows:

$$\text{remove}(j, \langle\rangle) = \langle\rangle$$
$$\text{remove}(j, \langle k,d\rangle \circ x) = \mathit{if}\ j = k\ \mathit{then}\ x\ \mathit{else}\ \langle k,d\rangle \circ \text{remove}(j,x)$$

4. Implementation

4.1 Overwiev

FOCUS offers two implementation languages, called AL and PL, respectively. AL is an applicative (functional) programming language, while PL is an imperative (procedural) one. Nondeterminism can be expressed in both languages. An AL program is an executable functional specification in a special syntax. There is a set of transformation rules which allow AL programs to be translated into PL programs.

4.2 Rationale

The reason for transforming functional programs into imperative ones is clearly the increase in efficiency, especially on distributed hardware. AL and PL are designed to share the same semantic basis (sets of stream processing functions), which makes it easy to give transformation rules and prove them correct.

4.3 Notation and Example

The transmitter written as an AL program looks similar to the functional implementation given in the previous section. Thus we do not give an AL version here. By applying some transformation rules, the following PL program for the transmitter can be derived. The correctness of this program follows by construction.

```
agent transmitter ≡ chan snd x, chan get y → chan put o :
   var nat count := 1; var seq info queue := ⟨⟩;
   var snd inputx; var get inputy;
   loop x?inputx; y?inputy;
         if   queue ≠ ⟨⟩  then  o!ft(queue)  fi;
         if   inputy = √  then  skip
                          else queue := remove(inputy,queue) fi
```

www.manaraa.com

```
        if    inputx = √   then skip
                           else queue := queue ∘ ‹count,inputx›; count := count + 1 fi;
    endloop
  endagent
```

Here **snd**, **get**, **put**, **info** are syntactic (sort) identifiers for the respective sets introduced before.

5. Conclusion

We have sketched the design method FOCUS in its present state. On the basis of case studies, we are always consolidating and expanding the method. Whereas initially we concentrated on control-intensive software systems (the communication system being a typical example), we now extend our method to areas such as hardware modelling.

Up to now, FOCUS has been a paper-and-pencil-method. However, first experience has been gained concerning tool-support. We are going to continue these efforts, mainly in the area of verification assistance.

References

[1] M. Broy: Towards a Design Methodology for Distributed Systems. In: M. Broy (ed.): Constructive Methods in Computing Science, NATO ASI Series F: Computer and System Sciences, Vol. 55, Springer 1989, pp. 311-364

[2] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. Gritzner, R. Weber: The Design of Distributed Systems — An Introduction to FOCUS. SFB-Bericht Nr. 342/2/92 A, Technische Universität München 1992

[3] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. Gritzner, R. Weber: Summary of case studies in FOCUS — A Design Method for Distributed Systems. SFB-Bericht Nr. 342/3/92 A, Technische Universität München 1992

[4] K. M. Chandy, J. Misra: *Parallel Program Design, A Foundation.* Addison-Wesley 1988

[5] P. Coad, E. Yourdan: *Object-Oriented Analysis,* Second Edition. Prentice Hall 1991

[6] C. Dendorfer, R. Weber: From Service Specification to Protocol Entity Implementation — An Exercise in Formal Protocol Development. In: R. J. Linn, M. Ü. Uyar (Eds.): Proc. 12th Symposium on Protocol Specification, Testing, and Verification. North-Holland 1992, pp. 163-178

[7] D. Harel: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8, 1987, pp. 231-274

www.manaraa.com

[8] C. A. R. Hoare: Communicating Sequential Processes. Prentice-Hall 1985

[9] B. Jonsson: A Fully Abstract Trace Model for Dataflow Networks. In: Proc. 16th Annual ACM Symposium on Principles of Programming Languages, 1989, pp. 155-165

[10] Gilles Kahn: The Semantics of a Simple Language for Parallel Programming. In: J. L. Rosenfeld (Ed.): Information Processing 74, Elsevier Publishers 1974, pp. 471-475

[11] F. Kröger: *Temporal Logic of Programs.* EATCS Monograph 8, Springer 1987

[12] L. Lamport: A simple Approach to Specifying Concurrent Systems. Communications of the ACM 32(1), 1989, pp. 32-45

[13] M. E. S. Loomis, A. V. Shah, J. E. Rumbaugh: An Object Modelling Technique for Conceptual Design. In: J. Bézivin et al. (Eds.): ECOOP '87, European Conference on Object-Oriented Programming, LNCS 276, Springer 1987, pp. 192-202

[14] R. Milner: *A Calculus of Communicating Systems,* LNCS 92, Springer 1980

[15] E.-R. Olderog: Towards a Design Calculus for Communicating Programs. In: J. C. M. Baeten, J. F. Groote (eds.): CONCUR '91, 2nd International Conference on Concurrency Theory, LNCS 527, Springer 1991, pp. 61-77

[16] W. Reisig: *Petri Nets. An Introduction.* EATCS Monograph 4, Springer 1985

[17] G. Rozenberg, G. Goos, J. Hartmanis (Eds.): Advances in Petri Nets. LNCS 524, Springer 1992

[18] Special Issue on Specification of Concurrent Systems. Distributed Computing 6(1), 1992

[19] V. Stenning: A Data Transfer Protocol. Computer Networks 1, 1976, pp. 98-110

www.manaraa.com

Parallelism in a Semantic Network for Image Understanding

V. Fischer and H. Niemann

Lehrstuhl für Informatik 5 (Mustererkennung)
Friedrich-Alexander-Universität Erlangen-Nürnberg
Martensstr. 3
D-8520 Erlangen

Abstract. The use of multiprocessor systems in the field of image analysis and understanding is motivated by the huge amount of data and the need for extremly fast processing that is necessary to achieve practical computer vision systems. The development of parallel algorithms for the knowledge-based interpretation of complex scences (patterns) must consider aspects of knowledge representation as well as it's efficient use. It is complicated by special problems of symbolic image processing, like unreliable segmentation results, variable data dependencies, or the need for integration of different levels of representation.
This paper introduces a parallel control algorithm for an image understanding system that uses semantic networks for the representation of task-specific knowledge. The algorithm is based on an explicit representation of all necessary inferences in a single data flowchart. In a bottom-up instantiation the sensor data are used for the computation of competing instances for each element of the knowledge base. A top-down optimization is used for the iterative improvement and selection of hypotheses.

1 Motivation

The processing of a huge amount of data under very restricted time conditions is the main ability that is assigned to today's computer systems by the analysis of complex sensor data, and especially by knowledge-based image understanding. The analysis of image sequences taken at video rate requires the interpretation of complex scenes every 40 milliseconds. A typical image size of 512×512 pixels in each of the three spectral channels therefore requires the execution of about 20 million instructions per second to perform a single operation keeping up with the sensory input [8]. It is generally assumed that the interpretation of an image needs much more operations per pixel ($10^3 - 10^4$), so that a computational power of about 150 GIPS must be available for real-time computer vision.

Considering the limitaions of today's conventional, single-CPU computers, the use of multiprocessor systems seems the only way to achieve the desirable speed. Furthermore, the massively parallel processing in the human visual system is a strong indication that the use of parallel systems can support the development of new, powerful algorithms (and intersting applications) in the field of machine vision.

www.manaraa.com

Image interpretation is a mapping of sensor data into a set of symbols (the knowledge base), that requires a (large) variety of operations and different representations of intermediate results (and intermediate representations). Usually three different levels of processing are distinguished [5, 1, 18]:

- Iconic processing *(low-level-processing)* deals with problem-independent algorithms and typically uses pixel-oriented methods.
- Symbolic processing *(high-level-processing)* is characterized by the representation and use of task-dependent knowledge. The goal of symbolic processing is the computation of an appropriate symbolic description which
  - optimally fits to the result of iconic processing,
  - is maximally compatible with a priori knowledge,
  - contains the information relevant for the current application.

  Hence, symbolic processing is an optimization problem.
- An *intermediate level* serves as an interface between iconic and symbolic processing and may include an initial symbolic description to represent all data available from the iconic level [11].

There is quite a large number of architectures and algorithms dealing with problems of low-level-processing [26, 2, 20]. In contrast, for several reasons parallelization of algorithms for symbolic processing is not so far developed:

- Different from many low-level-algorithms there are no fixed, predetermined and/or regular data dependencies or communication patterns.
- The performance of high-level methods is highly dependent on results from the other processing stages (e.g. initial segmentation), and some problems of symbolic processing (e.g. integration of different knowledge sources or different judgments (judgment schemes) are not well understood on monoprocessor systems.

The present contribution describes first results in the development of a parallel algorithm for the knowledge-based interpretation of complex patterns. Section 2 gives a brief account on the knowledge base and it's use in image analysis. In section 3 we introduce the parallel control algorithm, that is based on an explicit representation of all inferences in the knowledge base. The sections 4 and 5 report first results and experiences with that algorithm and give an outline of further work.

2 The Semantic Network System ERNEST

The automatic interpretation of complex patterns like color images, image sequences or continuous speech requires the representation and use of problem-dependent knowledge.

The ERlangen semantic NEtwork SysTem ERNEST ([16]) provides a framework for knowledge based pattern understanding that allows both representing declarative and procedural knowledge in a semantic network and formulating problem-independent control algorithms. The system is used in fields of image analysis [14, 12, 13, 23] as well as in speech understanding [15].

www.manaraa.com

2.1 Declarative Knowledge

In general, a semantic network is a directed and labeled graph, whose nodes represent ideas or objects and whose links represent certain relations between those ideas (nodes).

The particular knowledge representation language of the ERNEST-system distinguishes three types of nodes and five different links.

The three nodes are the *concept*, the *instance*, and the *modified concept*. A *concept* is a complex data strucuture for the definition of an idea, an object, or an arbitrary situation. A concept consists of several substructures for further intensional descriptions and of slots for the use of knowledge during analysis. Furthermore, there are slots used for automatic knowledge acquisition and for explanation of results.

Regarding the representation and use of knowledge during analysis the conceptually most important slots (substructures) are:

- the *link description* for the detailed description of relations between concepts,
- the *attribute description* for the definition of a concept's properties,
- the *relation description* for the description and test of relations between attributes.

While concepts represent events or objects intensionally, *instances* associate intervals of the sensor data with concepts in the knowledge base and are therefore an extensional description.

The third type of node is the *modified concept*. It is used to constrain or modify the uninstantiated concepts during analysis and to reduce the complexity of instantiation. The data strucures of all nodes are identical, but in instances and modified concepts the description of attributes, relations, and links are replaced by results or restrictions.

The five types of links defined in the representation language are the *specialization link*, the *part link*, the *concrete link*, the *model link*, and the *instance link*. The *specialization link* points from a general concept to a more special one and is used for building up an inheritance hierarchie (a taxonomy). All substrucures of a concept (attributes, relations, links) are inherited from the general concept to a more special, unless inheritance is explicitly prohibited in the structure.

Decomposition of a complex concept C into simple parts or components P_i is expressed by introducing a *part link* from C to each P_i. As a special feature of the representation language, part links may be *context-dependent*, because often a certain concept P_i can only be recognized with respect to the whole object C.

In complex pattern understanding applications there are usually different levels of abstraction that must be processed (e.g pixels, 3D-Objects, real-world objects). The *concrete link* establishes relationships between concepts of different conceptional levels and provides some formal restrictions for the use of part and specialization links (see also [16]).

Finally, the *model link* is used during automatic knowledge acquisition to connect a learned concept to the corresponding concept in a model-scheme and the *instance link* associates instances with concepts according to classification.

www.manaraa.com

Both links are not part of the knowledge base and therefore not considered furthermore.

For concepts and link descriptions the representation language provides slots for structuring the problem-dependent knowledge. In order to support the efficient realization of large knowledge bases and their use during analysis, parts and concretes of a concept can be marked as *obligatory* or *optional* and can be grouped in the so-called *modality sets*. Modality sets are defined in a concept and enable the representation of multiple definitions of a certain object in a single concept. Part and concrete relationships between concepts can be expressed efficiently by assigning a minimal and maximal number of occurence (a *dimension*) to a link and by specifying a XOR-list of goal nodes in a link description.

In order to illustrate the terms introduced above, Figure 1 shows a simple graphical representation of a truck and it's corresponding network model.

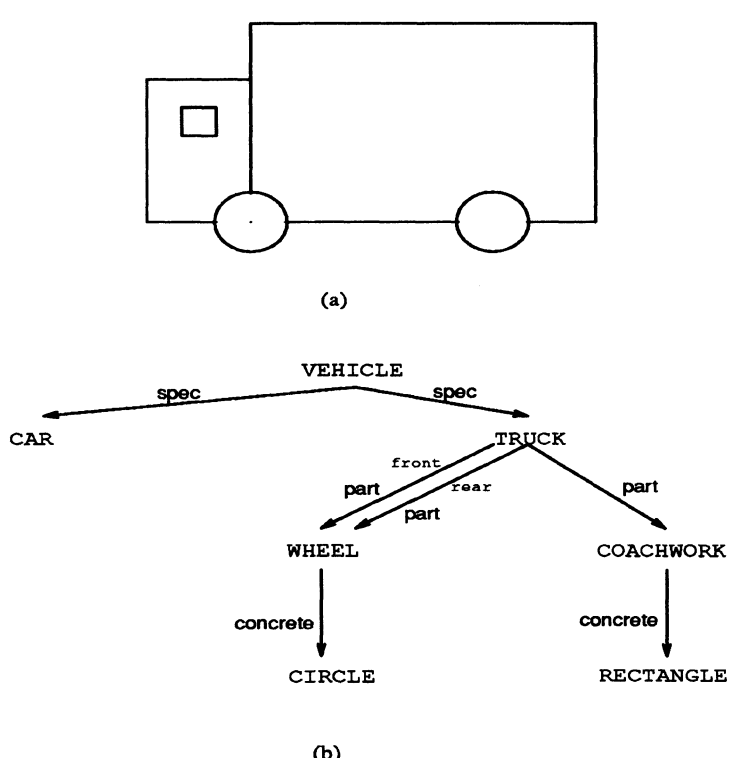


Fig. 1. Simple graphical representation (a) and semantic network (b) of a truck (from [16]).

www.manaraa.com

2.2 Procedural knowledge

Generation and verification of hypotheses or partial interpretations during analysis requires the incorporation of procedural knowledge into the network. In the syntax of the network, the substructure *function description* supports procedural attachment. Procedures can be attached to different substructures or slots of a concept:

- In an attribute description the functions are *computation of value* and *judgment*. The first is used to calculate the (quantitative) properties of a concept from the sensor data or from already known attribute values. The judgment compares the computed values with the restriction or the expected values from the knowledge base.
- In a relation the *judgment* points to a function that measures the degree of fulfillment of the relation.
- The *judgment* in a link dscription allows the valuation of a part or concrete with respect to the whole object.
- The slot *judgment* of a concept contains a pointer to a function that compares an instance to its concept in terms of quality, certainty, and/or priority.

With the exception of the judgment in an attribute description and in a concept, for each procedure an inverse function can be defined for the model-driven propagation of restrictions.

In the network formalism, every substructure is identified by a unique *role name*. By definition, the syntax of the function description restricts the potential arguments of a procedure. In section 3.1 these restrictions are used for the explicit representation off all data dependencies needed during analysis.

2.3 Problem-independent Control

Different from other semantic network formalisms (e.g. PSN, *procedural semantic network* [7, 10]) in our system the pragmatics of the network (that means the utilization of a network for image or speech understanding) is definded globally for the whole network, without respect to the field of application. This allows the development of problem-independent control algorithms for the utilization of the stored knowledge during analysis, because only the syntax of the network, but not the meaning of concepts has to be considered.

The main task of the control algorithm is the computation of an instance for a concept that defines a goal of analysis (goal concept) with respect to the sensory input. This requires the consideration of the representation language (e.g. resolution of modality sets) as well as the management of competing instances or modified concepts that causes competing states of analysis.

In the ERNEST-system control is based on six rules that define the instantiation process. Competing instances are handled by a state space search based on the A^*-algorithm [17]. The rules define the computation and completion of (competing) instances, their extension with optional parts or concretes, and the generation of modified concepts for constraint propagation. The latter can be

www.manaraa.com

done data-driven (bottom-up) or model-driven (top-down) and is a powerful method to reduce the complexity of search. A sixth rule is introduced to incorporate results of initial segmentation for the computation of possible goal concepts.

The rules are described in detail in [6, 22, 16] and can only be characterized here. Each rule has a premise and an action. In the premise the presence of parts, concretes and contexts is tested. If a premise holds true, the operations defined in an action are activated. They comprise the computation of attribute values and the judgment of attributes, relations, links, and the instance or modified concept itself.

3 Massively Parallel Control

As an important prerequisite for the development of algorithms for very fast ("reflexive") inferences, in [25] it is mentioned, that the syntactic structure of an effective representation should directly mirror the inferential structure used during analysis. Therefore, the first step in the design of a fast control algorithm is an encoding of the knowledge base into a graph, whose nodes contain appropriate elements of the network language, and whose links explicitly represent the inferential dependency between nodes.

Different from (classifier-)systems that draw conclusions from (certain) facts, a control algorithm for a knowledge-based image analysis system must be capable to work with uncertain and competing hypotheses. Those result from noisy input data and/or from segmentation errors and cause a combinatorial explosion of possible matches between models and data. Therefore a control algorithm must incorporate techniques for the suppression of possible intermediate results and the selection of competing interpretations.

For a given semantic network, the inferences needed during analysis are independent of the semantics of the concepts and the current state of interpretation. Hence they can be computed in advance from the syntax of the network to prepare an analysis process (section 3.1). In contrast, elemination of less promising results caused by segmentation errors is dependent on the input data and must be done by the control algorithm itself (section 3.2).

3.1 Encoding of the Knowledge Base

The goal of network encoding is the creation of a data flowchart that represents all neccessary inferences and dependencies used for matching the input data against the network. Criteria for consistency of a knowledge base (see [21]) are tested, and inheritance of attributes, relations, parts and concretes is established along the specialization hierarchy [6].

The first step for the computation of the data flowchart is the selection of appropriate chunks of knowledge that should be represented by the nodes. For semantic networks a natural choice is to map each concept to a single node. This seems to be promising, if concepts are far less complex than in the frame-based

www.manaraa.com

language of the ERNEST-system (see e.g. [24]). The links of the data flowchart would directly express the relations between concepts, in our case the part and concrete link. Beginning with the so-called *minimal concepts*, that are concepts without (inherited) parts and concretes, concepts in independent nodes can be instantiated in parallel.

The choice of concepts as elementary units for the nodes of the data flowchart preserves the benefits of semantic networks for the development of large knowledge bases (e.g. node-centered representation, lucidity, ergonomic adequacy). In our case, where concepts are rather complex structures, there are several disadvantages:

- The parallel computation of concepts does not allow data- and/or model-driven restrictions of the knowledge base by means of modified concepts. To clarify this, let us assume a concept C having two parts P_1 and P_2. If an instance for P_1 is computed first, the area for instances of P_2 is restricted to the complement in the image. If computed in parallel, the whole image must be considered for both concepts.
- Communication between nodes must handle complete instances, even if only single values from subinstances are needed for the instantiation of superior concepts.
- Parallelism is limited, because independent computations in superior concepts cannot be executed before or during instantiation of subconcepts.

Considering the last two items, it seems reasonable to look for finer parallelism in a network. Our suggestion is to handle all computations necessary for the instantiation of a goal concept independent from their concept membership. Dependencies between computations are stored in a graph, whose nodes represent a single attribute, a relation or the judgment of a link or a concept. During creation of the graph, all links are assumed obligatory and having maximal dimension (cf. section 2.1). This allows the proper treatment of multiple occurences of objects and their parallel instantiation during analysis.

Because the function description provides an uniform interface for the attachment of procedural knowledge (task-dependent procedures), the graph can be generated automatically from a given knowledge base. Thereby the advantages of the representation language are preserved.

Dependent on the structure, the following syntactical definitions of arguments are possible:

- The *computation of a value* in an *attribute description* may have attributes from the same concept, from it's parts, concretes, or a superior context.
- The argument of the *judgment* of an attribute is the attribute itself. No explicit entries are allowed.
- Arguments of the *judgment* in a *relation description* are attributes. Possible entries are the same as for the computatiopn of a value.
- The arguments of the *judgment* in a *link description* are the judgments of the instances or modified concepts that are given in the link's list of goal nodes. No explicit entries are allowed.

www.manaraa.com

- The *judgment* in a concept may be computed from judgments of attributes, relations, and/or links defined in or inherited by the concept.

Exploiting the syntactical description of arguments allows us to assign each node to a level l of the data flowchart. Let $\varphi_a(r_1, ..., r_n)$ denote the function description with arguments r_1 to r_n, that is bound to node a. The level l of node a is determined by the level of the function φ_a:

$$l_{\varphi_a} = \begin{cases} \max\{l_{\varphi_{r_1}}, \ldots, l_{\varphi_{r_n}}\} + 1 \text{ if } & n > 0 \\ 0 & \text{else} \end{cases}$$

The lowest level ($l = 0$) is built by those attributes, that directly process the results from initial segmentation; the highest level usually contains nodes for the judgment of goal concepts. In Figure 2 some of the necessary computations from

```
TRUCK                                      WHEEL
  Part: front_wheel                          Concrete: circle
    Goalnode: WHEEL                            Goalnode: CIRCLE
    Judgment: restr_radius                     Judgment: restr_radius

  Part: rear_wheel                           Attribute: w_radius
    Goalnode: WHEEL                            Comp_of_value: com_radius
    Judgment: restr_radius                     Arguments:circle.c_radius

                                           Judgment: wheel_judge
  Part: coachwork                            Arguments: w_radius
    Goalnode: COACHWORK
    Judgment: restr_height

  Attribute: truck_height                  CIRCLE
    Comp_of_value: com_heigth                Attribute: c_radius
    Arguments: rear_wheel.w_radius,            Comp_of_value: com_length
               front_wheel.w_radius,           Arguments:
               coachwork.height

  Judgment: truck_judge                      Judgment: circle_judge
    Arguments: rear_wheel,front_wheel,         Arguments: c_radius
               coachwork, truck_height
```

Fig. 2. Some computations defined in the knowledge base shown in Figure 1.

the model in Figure 1 are presented; Figure 3 shows a part of the data flowchart. Boxes on the same horizontal line contain the functions which can be computed in parallel.

www.manaraa.com

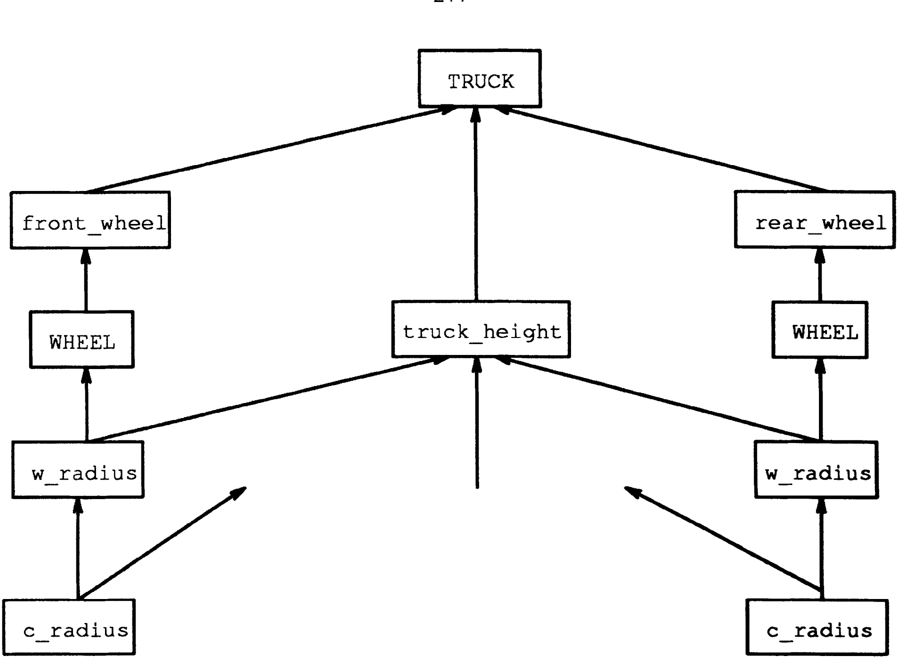


Fig. 3. Part of the inference graph generated from the model shown in Figure 2.

3.2 Analysis

The control algorithm sketched in section 2.3 computes an interpretation by the dynamic creation of instances and modified concepts during analysis. With respect to the special features of the representation language (e.g. sets of modalities) the hierarchy of parts and concretes must be dissolved recursively. Control, that means computation of an interpretation which is compatible to the sensory input, is reduced to heuristic graph search.

In [4] we showed, that the length of a solution path grows exponentially with the depth of the knowledge base, and in [19] it is shown, that very good heuristics are needed to avoid exponential complexity when searching for an optimal path. Furthermore, state space search demands the collection of intermediate results into states of analysis (nodes of the search graph) and therefore requires a large amount of communication between processors and a dynamically growing state space.

Therefore it seems reasonable to look for methods that promise less complexity than graph-based control and that are well suited for massively parallel processing.

The control algorithm described below treats the mapping of sensor data into the knowledge base as an optimization problem where an error criterion (judgment error) has to be reduced. The algorithm consists of two alternating phases, a data-driven *bottom-up instantiation* and a model-driven *top-down optimization* that uses techniques from neural networks and linear optimization for the iterative improvement of an interpretation.

www.manaraa.com

Bottom-up Instantiation. During bottom-up instantiation on each level of the data flowchart the nodes are computed in parallel. Therefore the user-defined function (e.q. the computation of an attribute value) is executed. Starting with the attributes on the lowest level that directly access results from the initial symbolic description or from initial segmentation, instantiation proceeds stepwise for all levels. Because results from iconic or iconic-symbolic-processing are usually erroneous and ambiguous, in general a node passes competing values to it's successors. Therefore the nodes on higher levels must be activated for every possible combination of inputs. To avoid the combinatorial explosion of competing hypotheses and also to restrict the communication between processors, it is useful to limit the number of hypotheses created by each node. [1]

Considering the judgment of a hypothesis it is possible to develop a local pruning method for each node. In our current applications it is possible to use an identical pruning criterion for each node. Actually the best n results, or all results judged better than a given threshold θ are possible candidates for further processing.

Top-down Optimization. During bottom-up instantiation for each node of the data flowchart competing judgments are computed. Each judgment consists of a measure for the quality of the mapping between sensor data and the piece of knowledge, that is represented by the node. The highest level of the graph usually contains judgments for different goal concepts, that represent competing interpretations of a scene or a signal.

The purpose of top-down optimization is to select the interpretation that matches best, and requires also the determination of all components of the interpretation and the elimination of wrong results.

To determine an optimal interpretation, we minimize the judgment error

$$E = f(\underline{g}_d - \underline{g}_o) \qquad (1)$$

The desired judgments are stored in the vector $\underline{g}_d$, and $\underline{g}_o$ is the vector of judgments computed during bottom-up instantiation. The determination of $\underline{g}_d$ requires an analyzed scene, that can be computed using the sequential control algorithm of the ERNEST-system (see section 2.3).

To minimize E, the links of the data flowchart are provided with weights, and each judgment g is written as a function of it's weighted predecessors:

$$g = \varphi_a(\omega_1 r_1, \ldots, \omega_n r_n) \qquad (2)$$

In each step of iteration the weights $\underline{\omega} = (\omega_1, \ldots, \omega_n)$ are adjusted for each node

[1] Other possibilities would be the application of additional tests for plausibility and, of course, an improvement of low-level-processing, which is both out of interest in this paper.

www.manaraa.com

according to (Eq. 3). Alternating with bottom-up instantiation, this proceeds until an appropriate criterion, e.g. $E \leq \epsilon$, becomes true.

$$\underline{\omega}^{t+1} = \underline{\omega}^{t} + \Delta^{t} \cdot \underline{r}^{t} \tag{3}$$

An investigation of the user-defined judgment is necessary to determine an appropriate optimization method. If a derivative exists, a gradient descent can be done ($\underline{r}^t = \delta E / \delta \omega_i$), else a coordinate descent would be appropriate.

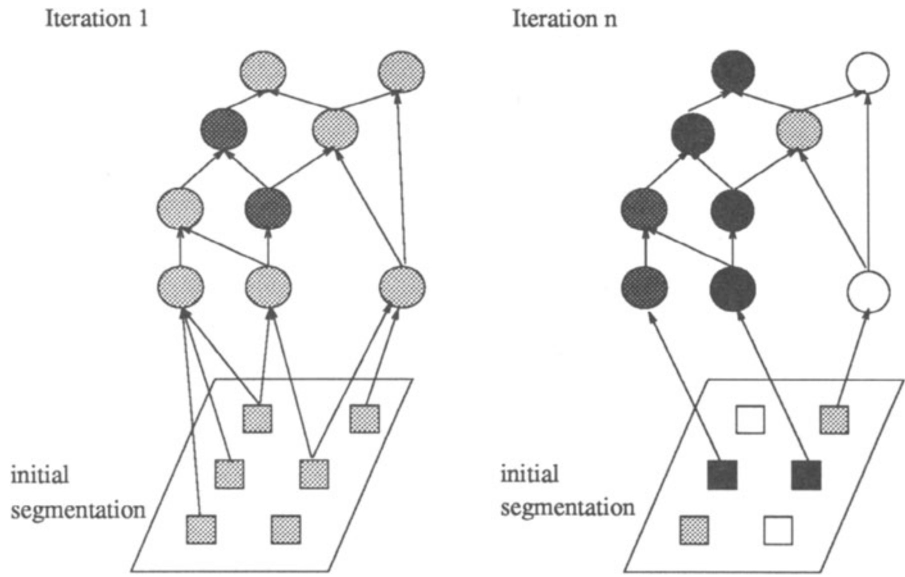


Fig. 4. Effects of the two-stage control algorithm. The judgment of a node is improved stepwise (indicated by dark circles) and a unique correspondence is established.

In Figure 4 the effect of the two-stage algorithm is shown schematically. After bottom-up instantiation (left side), results from initial segmentation (squares) are assigned to the elements of the knowledge base (circles) and competing interpretations are equally shaded. Performing n iterations results in a unique assignment of segmentation objects and a refined and precise interpretation (black circles), whereas wrong results are suppressed (white circles).

4 Results

Most of the work done so far investigates the phase of bottom-up instantiation of the algorithm described above. Figure 5 shows part of a knowledge base for

www.manaraa.com

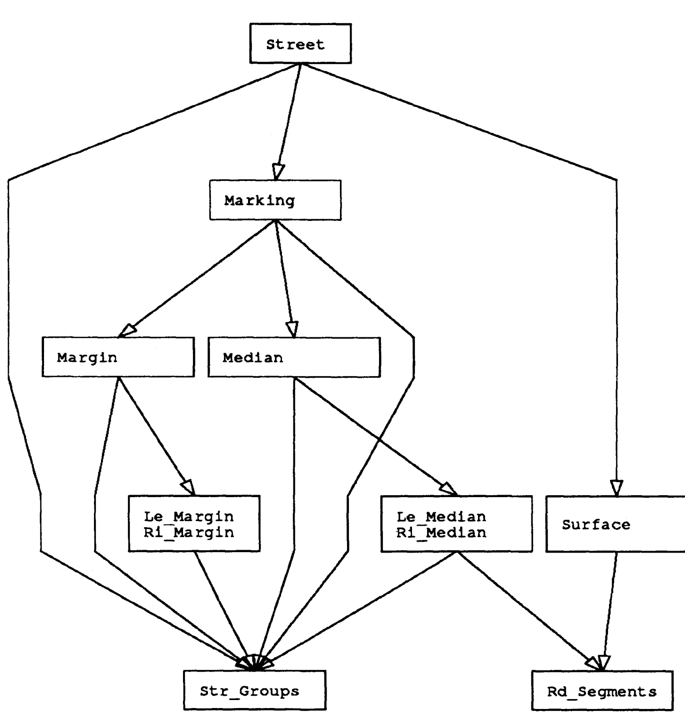


Fig. 5. Knowlege base for the interpretation of traffic scenes (excerpt).

the interpretation of traffic scenes [2] [9]. During the preparation of analysis for the shown network a inference graph is constructed, that contains 48 nodes on 10 levels. Figure 6 shows an excerpt of the graph, and verifies the unequal filling of the different levels (max. 19 nodes (level 1), min. 1 node) as well as the irregularity of connections.

Table 1 reports CPU-times for the bottom-up instantiation of some concepts. Columns 2 - 4 show results for processing of the $n = 3, 2, 1$ best ranked hypotheses generated by a node. Because of the fuzzy-functions used in this example, the best interpretation is always found, if we only process the best alternative ($n = 1$). For comparison, the last column reports results for the graph-based sequential algorithm [6]. For $n = 1$ Table 2 reports the speed-up s and efficiency e of a multiprocessor simulation, where largest possible parallelism (column labeled p) was assumed. Results were averaged over a sequence consisting of 30

[2] The authors are very grateful to Mrs. S. Steuer (FORWISS Munich) for making her knowledge base available to us.

www.manaraa.com

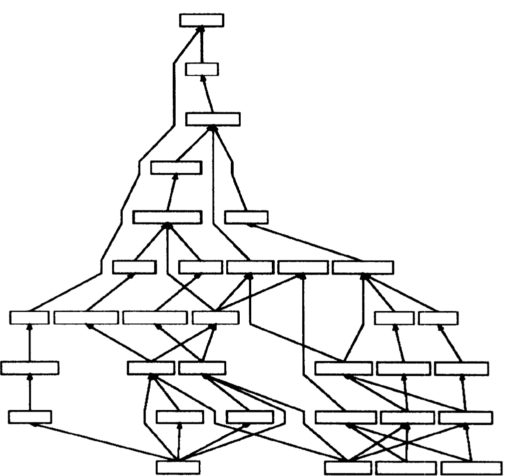


Fig. 6. Structure of the data flowchart.

goal concept	T[s]			T_s[s]
	$n=3$	$n=2$	$n=1$	
Street	0.18	0.15	0.15	0.41
Marking	0.18	0.15	0.14	0.32
Median	0.13	0.11	0.11	0.16
Margin	0.05	0.05	0.05	0.10

Table 1. CPU-time for bottom-up instantiation.

images and obtained by using the PEPP-system described in [3]. The columns marked *coarse* are the results we obtain, if we assign a concept to each processor. Our approach, that is assigning a single computation to each processor, is marked *medium*. As a result it is obtained, that we can achieve better speed-ups by means of a finer parallelization. In addition to the unequal filling of the different levels, the different tasks on each level provide the main reason for quite small speed-ups: more than 50 percent of the total time T_p (respectively T_s) was spent for the computations on level 0, the interface to signal processing. Considering this observations it seems promising to use several processors for the computation of these nodes. First investigations in this direction show, that this will result in a significant improvement of speed-up without losses in efficiency. Also small modifications of the knowledge base (e.g. splitting up complex attributes into simple components) seem to be promising.

Because of the large variety and number of parameters (see section 3.2), the top-down optimization has not been tested sufficiently up to now. First results indicate, that the weighted links bear a large amount of information about a scene that is won during analysis.

www.manaraa.com

goal concept	coarse			medium		
	p	s_p	e_p	p	s_p	e_p
Street	4	1.597	0.399	14	2.17	0.155
Marking	3	1.809	0.603	12	2.510	0.209
Median	2	1.689	0.844	5	2.020	0.404
Margin	2	0.960	0.480	4	1.633	0.408

Table 2. Speed-up and efficiency for bottom-up instantiation.

5 Conclusions and Outlook

In the preceding sections a massively parallel control algorithm for knowledge based pattern analysis was introduced. To preserve the advantages of our semantic network formalism for knowledge representation and it's use in pattern understanding, we decided to explicitly represent all inferences in a graph by running some algorithms for the preparation of an analysis. By use of a two-stage method the control algorithm generates an interpretation that is compatible with the sensor data. Parallel bottom-up instantiation of the nodes computes competing matches for all elements of the knowledge base. It is alternating with a top-down optimization, that causes a stepwise improvement and elemination of alternative interpretations. In a simple example, fairly small speed-ups are observed as a result obtained from a multiprocessor simulation of the bottom-up instantiation. The main reasons are the irregular data dependencies and the different tasks on each level of abstraction.

In contrast, there seems to be some "learning capability" during top-down optimization. The iterative top-down optimization is a means for variable depth analysis. If much time is available, many iterations can be carried out to obtain refined and precise results. If a quick respose is necessary, only few iterations are performed at the price of less reliable results. They are the object of further investigations, that are beyond our original goal, which was parallel instantiation. In addition to aspcets of representation and use of knowledge in "connectionist semantic networks", we think of the efficient analysis of image sequences.

References

1. D. Ballard and C. Brown. *Computer Vision.* Prentice Hall, Englewood Cliffs, New Jersey, 1982.
2. V. Chaudhary and J.K. Aggarwal. Parallelism in computer vision: A review. In V. Kumar, P.S. Gopalakrishnan, and L.N. Kumar, editors, *Parallel Algorithms for Machine Intelligence and Vision*, pages 271–309. Springer-Verlag, New York, Berlin, Heidelberg, 1990.

www.manaraa.com

3. P. Dauphin, F. Hartleb, M. Kienow, V. Mertsiotakis, and A. Quick. Pepp: Performance evaluation of parallel programs. users's guide – version 3.1. Technical Report 5/92, Lehrstuhl für Informatik 7, Friedrich-Alexander-Univerität Erlangen-Nürnberg, September 1992.
4. V. Fischer. Parallelisierung von Instantiierungsoperationen in semantischen Netzen. In *Arbeits- und Ergebnisbericht des Sonderforschungsbereichs 182 "Multiprozessor- und Netzwerkkonfigurationen"*. Teilprojekt D1, chapter 5.2, pages 109–136. Erlangen, 1992.
5. A. Hanson and E. Riseman. Processing cones: A computational structure for scene analysis. In S. Tanimoto and A. Klinger, editors, *Structured Computer Vision*. Academic Press, New York, 1980.
6. F. Kummert. *Flexible Steuerung eines sprachverstehenden Systems mit homogener Wissensbasis*, volume 12 of *Dissertationen zur Künstlichen Intelligenz*. Infix, Sankt Augustin, 1992.
7. H. Levesque and J. Mylopoulos. A procedural semantics for semantic networks. In N. Findler, editor, *Associative Networks*, pages 93–121. Academic Press, Inc., New York, 1979.
8. S. Levitan, C. Weems, A. Hanson, and E. Riseman. The UMASS image understanding architecture. In L. Uhr, editor, *Parallel Computer Vision*, pages 215–248. Academic Press, Inc., Boston, 1987.
9. T. Messer, S. Steuer, C. Weighardt, D. Wetzel, A. Winklhofer, and A. Zins. Movie-Projektbeschreibung: Schritthaltende Bildfolgeninterpretation von Fahrszenen. Technical report, Bayerisches Forschungszentrum für Wissensbasierte Systeme (FORWISS) und Bayerische Motorenwerke AG (BMW AG), 1990.
10. J. Mylopoulos and H. Levesque. An overview of knowledge representation. In M. Brodie, J. Mylopoulos, and J. Schmidt, editors, *On Conceptual Modelling: Perspectives from Artificial Intelligence*, pages 3–17. Springer-Verlag, Berlin, Heidelberg, New York, 1983.
11. H. Niemann. *Pattern Analysis and Understanding*. Springer-Verlag, Berlin, Heidelberg, New York, 1990.
12. H. Niemann, H. Brünig, R. Salzbrunn, and S. Schröder. Interpretation of industrial scenes by semantic networks. In *Proc. of the IAPR Workshop on Machine Vision Applications*, pages 39–42, Tokyo, 1990.
13. H. Niemann, H. Brünig, R. Salzbrunn, and S. Schröder. A knowledge-based vision system for industrial applications. In *Machine Vision and Applications*, volume 3, pages 201–229. Springer-Verlag, New York, 1990.
14. H. Niemann, H. Bunke, I. Hofmann, G. Sagerer, F. Wolf, and H. Feistel. A knowledge based system for analysis of gated blood pool studies. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 7(3):246 – 259, 1985.
15. H. Niemann, G. Sagerer, U. Ehrlich, E.G. Schukat-Talamazzini, and F. Kummert. The interaction of word recognition and linguistic processing in speech understanding. In R. De Mori and P. Laface, editors, *Recent Advances in Speech and Language Modelling*, NATO ASI Series F. Springer-Verlag, Berlin, 1992.
16. H. Niemann, G. Sagerer, S. Schröder, and F. Kummert. **ERNEST**: A semantic network system for pattern understanding. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 12(9):883–905, 1990.
17. J. Nilsson. *Principles of Artificial Intelligence*. Springer-Verlag, Berlin, Heidelberg, New York, 1982.
18. D. Paulus. *Objektorientierte Bildverarbeitung*. Dissertation, Technische Fakultät der Universität Erlangen-Nürnberg, 1991.

www.manaraa.com

19. J. Pearl. *Heuristics. Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.
20. V. Prassana Kumar. *Parallel Architectures and Algorithms for Image Understanding.* Academic Press, Inc., Boston, 1991.
21. G. Sagerer. *Darstellung und Nutzung von Expertenwissen für ein Bildanalysesystem*, volume 104 of *Informatik-Fachberichte.* Springer-Verlag, Berlin, Heidelberg, New York, 1985.
22. G. Sagerer. *Automatisches Verstehen gesprochener Sprache*, volume 74 of *Reihe Informatik.* B.I. Wissenschaftsverlag, Mannheim, 1990.
23. G. Sagerer, R. Prechtel, and H.-J. Blickle. Ein System zur automatischen Analyse von Sequenzszintigrammen des Herzens. *Der Nuklearmediziner*, 3:137–154, 1990.
24. L. Shastri. *Semantic Networks: An Evidential Formalization and its Connectionist Realization.* Pitman and Morgan Kaufmann Publishers, Inc., London and San Mateo, California, 1988.
25. L. Shastri. Why semantic networks? In J.F. Sowa, editor, *Principles of Semantic Networks. Explorations in the Representation of Knowledge*, chapter 3, pages 109–136. Morgan Kaufmann Publishers, Inc., San Mateo, Ca., 1991.
26. L. Uhr. *Parallel Computer Vision.* Academic Press, Inc., Boston, 1987.

www.manaraa.com

Architectures for Parallel Slicing Enumeration in VLSI Layout

Henning Spruth and Frank M. Johannes

Institute of Electronic Design Automation
Technical University of Munich, D-8000 Munich 2, Germany

Abstract. This paper presents parallel algorithms for solving the final placement problem of rectangular cells with predefined neighborhood relations. Optimum solutions for small cell subsets are obtained by enumerating all arrangements, i.e. slicing structures. These solutions are combined in a global construction step such that they fit well into the global arrangement.
An increased size of the enumerated local subproblems leads to placements that are closer to a global optimum. However, this requires significantly larger computing resources. Parallel computers provide huge amounts of computing power and memory that can be used to meet these high demands.
In this paper, we present new algorithms to solve this problem on several parallel architectures. By adjusting the granularity of the algorithm to the properties of the specific target architecture, we achieve significant speed-ups.

1 Introduction

The placement of rectangular cells of fixed or variable size is a combinatorial optimization problem that is very hard to solve. Many solution methods have been proposed based on simulated annealing, e.g. [1], partitioning by min-cut or clustering, e.g. [2, 3, 4], and analytical methods like [5, 6].

Partitioning and analytical algorithms separate the placement task in two subproblems:

- the global (or point) placement phase which determines the neighborhood relations of the cells, optimizing total net length
- the final placement phase which generates a design rule correct cell arrangement, using the neighborhood relations as constraints and optimizing chip area.

The advantage of these methods is a significantly shorter computation time than usually consumed by annealing methods. However, the placement results can be acceptable (i.e. close to a global optimum solution) only if both steps are closely interlocked.

In this paper, we present parallel algorithms for the final placement. Based on the principles of slicing [7] and shape functions [8, 9], a local enumeration

www.manaraa.com

algorithm [10] yields optimum results for the final placement problem of small cell sets. These optimum results are then combined to a global solution whose quality strongly depends on the size of the optimally solved local problems. Therefore, it is crucial to enlarge the size of the local subproblems which becomes feasible by supplying more computing power and memory. Since these resources are available on parallel computers, we have developed concepts for parallel enumeration algorithms [11]. In this paper, we show how these concepts are successfully applied both to distributed- and shared-memory architectures.

Section 2 briefly introduces shape functions and the construction step, where local solutions are combined in a global arrangement. In Section 3, a sequential recursive enumeration algorithm [10] is discussed and compared with a nonrecursive algorithm that produces the same result, but is parallelizable. Parallel algorithms for the enumeration step are presented in Section 4. A coarsegrained parallel algorithm is used to enumerate different subsets in parallel, while a finegrained parallel algorithm speeds up the enumeration process for one individual cell subset. By selecting the best combination of algorithms for different target architectures, we achieve high speed-ups and good layout quality that are presented in Section 5.

2 The construction phase

The goal of final placement is to minimize the wasted chip area A_w, which is defined using the quotient of the area sum of all cells (with dimensions w_μ and h_μ) and the area of the computed cell arrangement (with dimensions w_0 and h_0):

$$A_w = 1 - \frac{\sum_{\mu \in \mathcal{M}_0}(w_\mu \cdot h_\mu)}{w_0 \cdot h_0}$$

This is achieved in the following way:

As many other placement methods, e.g. [2, 3, 12], the global placement procedure GORDIAN [6, 13] generates a partitioning tree by recursively partitioning the set of cells to be placed. By additionally dissecting the available placement area with horizontal or vertical cuts, the partitioning tree becomes a binary slicing tree (see Fig. 1). Each node of the slicing tree represents a rectangular region $\rho = (x_\rho, y_\rho, \mathcal{M}_\rho, \sigma_\rho, p_\rho, \mathcal{F}_\rho)$ of the placement area with the coordinates of the lower left corner x_ρ and y_ρ, a subset of cells $\mathcal{M}_\rho$, and a label $\sigma_\rho \in \{c, e, h, v\}$ indicating whether a region is a single cell (c), undissected (i.e. to be enumerated) (e) or dissected by a horizontal (h) or vertical (v) cut. In the finegrained parallel algorithm which is executed on a set $\mathcal{P}$ of processors (see Section 4), $p_\rho \in \mathcal{P}$ is the processor that enumerates and stores all shapes of region ρ.

$\mathcal{F}_\rho = \{(w_i, h_i) | w_i < w_{i+1} \wedge h_i > h_{i+1},\ 1 \leq i \leq n-1\}$ is the discrete shape function [8, 9] containing the widths and heights of all area minimal arrangements of the cells in region ρ. This discrete function can be extended to a

www.manaraa.com

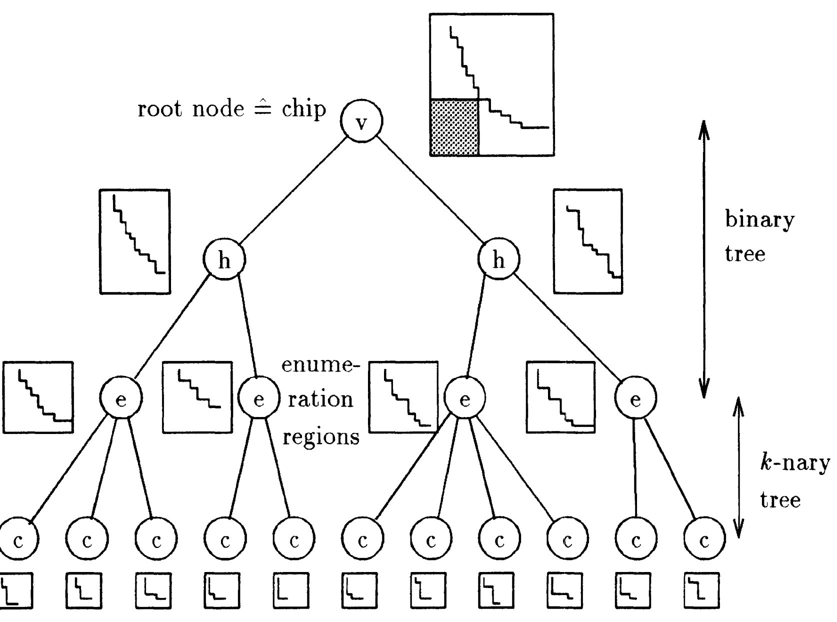


Fig. 1.: Slicing tree

continuous integer staircase shape function f_ρ (see Fig. 2)

$$f_\rho(w) = \begin{cases} \infty & w < w_1 \\ h_i & w_i \leq w < w_{i+1} \quad i = 1, \dots, n-1 \\ h_n & w_n \leq w \end{cases}$$

or to an inverse integer staircase shape function f_ρ^{-1}:

$$f_\rho^{-1}(h) = \begin{cases} \infty & h < h_n \\ w_{i+1} & h_{i+1} \leq h < h_i \quad i = n-1, \dots, 1 \\ w_1 & h_1 \leq h \end{cases}$$

The shape functions of all nonleaf regions in the slicing tree can be computed by adding the shape functions of their sons. Depending on the cut direction σ_ρ, the shape function of a father region ρ is computed from its sons ρ' and ρ'' by the following addition operator:

$$f_\rho = f_{\rho'} \textcircled{\scriptsize σ_ρ} f_{\rho''} \Leftrightarrow \begin{cases} f_\rho \;\; = f_{\rho'} + f_{\rho''} & \text{if } \sigma_\rho = h \\ f_\rho^{-1} = f_{\rho'}^{-1} + f_{\rho''}^{-1} & \text{if } \sigma_\rho = v \end{cases}$$

Starting with the enumeration nodes in the slicing tree, a bottom-up traversal yields a shape function for the root region $\rho = 0$ of the slicing tree, which corresponds to the whole chip. After selecting an appropriate shape (w_0, h_0) for the chip, the coordinates of the lower left corners (x_ρ, y_ρ) can be computed for all regions and cells in a top-down traversal of the slicing tree.

www.manaraa.com

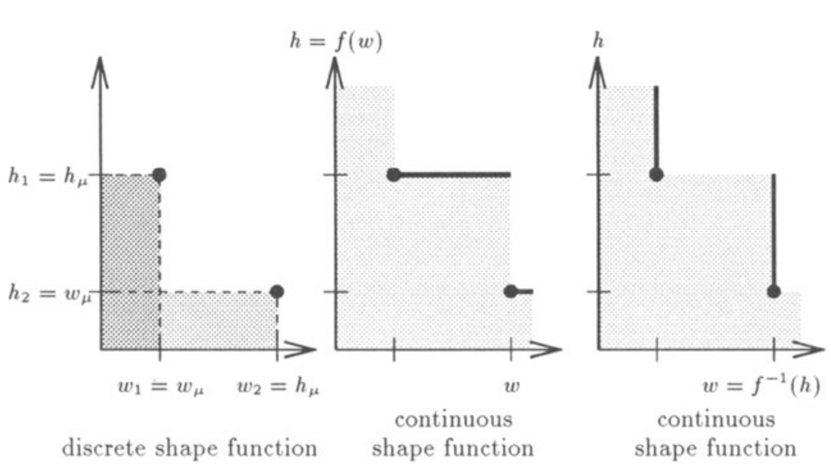


Fig. 2.: Shape function of a rotatable cell

3 Slicing enumeration

The construction step described above is applicable if the shape functions of all enumeration regions $\epsilon \in \mathcal{R}^e = \{\rho \mid \sigma_\rho = e\}$ are known, which is immediately true if each enumeration region contains only one cell. Unfortunately, large wasted area (see Fig. 3), which may occur if cells with different sizes are combined in one father region, decreases the area utilization of the resulting placement. A good area utilization can be achieved only if shape functions of enumeration regions consist of a large set of shapes. To generate such a set of shapes (i.e., a shape function), slicing structures are enumerated for regions that contain more than one cell but at most k_{max} cells.

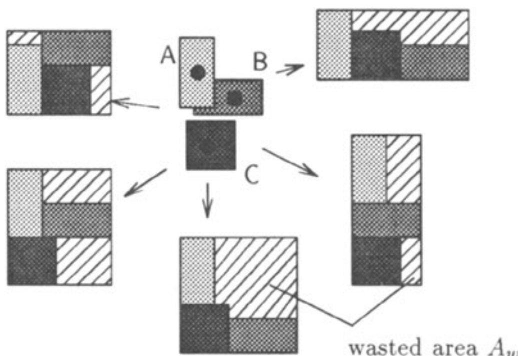


Fig. 3.: Enumeration for a region with three cells

An enumeration algorithm with polynomial complexity [10] is used that pre-

www.manaraa.com

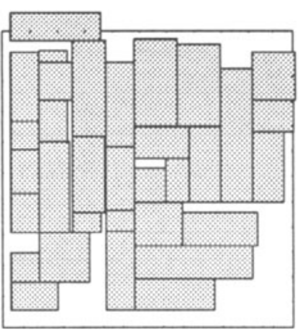
serves the neighborhood relation defined by the global placement coordinates (x_μ, y_μ) of the cells. These coordinates are computed in the global placement phase such that wire length is minimized with cells modeled as points [13]. To enable a definite ordering, cells with equal x- or y-coordinates are separated using a simple net cut model.

Figure 3 shows an example of the arrangements computed by the enumeration algorithm for a set of 3 cells $\{A, B, C\}$. Preserving the neighborhood relation means here that cell A, e.g., is always placed at the left or on top of the other cells.

The enumeration algorithm is based on the observation that shape functions of alternative arrangements of two son regions can be combined in one shape function by the minimizing union operation [12]. If, e.g., two son regions ρ' and ρ'' can be separated either by a horizontal or by a vertical cut, the resulting shape function of the father region is computed as

$$f_\rho = \min(f_{\rho'} \circledS f_{\rho''} \ , \ f_{\rho'} \circledcirc f_{\rho''})$$

For an enumeration region $\epsilon \in \mathcal{R}^e$ with $k_\epsilon = |\mathcal{M}_\epsilon| \leq k_{max}$, the time complexity of the enumeration algorithm is $O(k_\epsilon^6)$ and the memory usage increases with $O(k_\epsilon^5)$. However, less chip area is wasted when the number of cells in an enumeration region is increased [14] (see Fig 4).



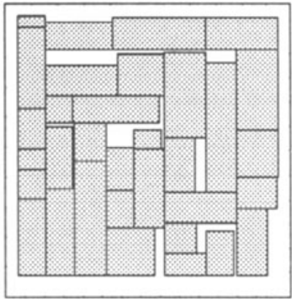


Fig. 4.: Final placements of circuit ami33 for $k_{max} = 7$ (left) and $k_{max} = 33$ (right). Note the much smaller wasted area in the right picture.

3.1 A recursive algorithm

The task of the enumeration algorithm shown in Fig. 5 is to calculate a shape function for each region $\epsilon \in \mathcal{R}^e$ which describes all area minimal cell arrangements that can be derived from slicing the point placement. This recursive procedure first dissects the region by horizontal and then by vertical cuts into son regions ρ' and ρ''. For the cell subsets $\mathcal{M}_{\rho'}$ and $\mathcal{M}_{\rho''}$ created by these dissections, the same procedure is performed recursively. If the shape functions of the two subsets are known, they are added either horizontally or vertically (operator ⓗ

المنارة للاستشارات

www.manaraa.com

```
For all ε ∈ R^e call: Enumerate(M_ε);

procedure Enumerate (M_ρ)
if (f_ρ = 0)
   for each σ ∈ {h, v}                    /* hor. and vert. cuts */
     M_ρ' := M_ρ; M_ρ'' := ∅;
     while (|M_ρ'| > 1)
       if (σ = v)  μ := λ_0 | x_λ0 =  min   (x_λ);
                                    λ∈M_ρ'
       else        μ := λ_0 | y_λ0 =  min   (y_λ);
                                    λ∈M_ρ'
       M_ρ' := M_ρ'\{μ}; M_ρ'' := M_ρ'' ∪ {μ};
       f_ρ' := Enumerate(M_ρ');
       f_ρ'' := Enumerate(M_ρ'');
       f_ρ := min(f_ρ; f_ρ' ⓢ f_ρ'');
     end while
   end for
end if
return (f_ρ);
end procedure
```

Fig. 5.: Recursive enumeration algorithm

or ⓥ). This new shape function is then merged with the already known shape function of region ρ.

Unfortunately, this recursive algorithm is not suited for parallel execution. We will, however, show that careful analysis by means of a dependency graph leads to independent subproblems that may be executed in parallel using a nonrecursive enumeration algorithm.

3.2 The dependency graph

The treatment of the regions during enumeration leads to a dependency graph that expresses the father-son relationships of all subregions used in the enumeration. The dependency graph is a directed acyclic graph consisting of a single root node representing an enumeration region $\epsilon \in \mathcal{R}^e$. The graph has k_ϵ leaf nodes representing the k_ϵ cells inside the enumeration region. An arc leads from node ρ to node ρ' if the shape function of region ρ' is required to compute the shape function of region ρ. The set of all successors ρ' of a node ρ is defined as follows using $\overline{\mathcal{M}}_{\rho'} = \mathcal{M}_\rho \setminus \mathcal{M}_{\rho'}$:

$$\text{suc}(\rho) = \left\{ \rho' \mid \mathcal{M}_{\rho'} \subset \mathcal{M}_\rho \;\wedge\; \left[\underset{\substack{\mu \in \mathcal{M}_{\rho'} \\ \lambda \in \overline{\mathcal{M}}_{\rho'}}}{\forall} x_\mu < x_\lambda \;\vee\; \underset{\substack{\mu \in \mathcal{M}_{\rho'} \\ \lambda \in \overline{\mathcal{M}}_{\rho'}}}{\forall} x_\mu > x_\lambda \;\vee\; \underset{\substack{\mu \in \mathcal{M}_{\rho'} \\ \lambda \in \overline{\mathcal{M}}_{\rho'}}}{\forall} y_\mu < y_\lambda \;\vee\; \underset{\substack{\mu \in \mathcal{M}_{\rho'} \\ \lambda \in \overline{\mathcal{M}}_{\rho'}}}{\forall} y_\mu > y_\lambda \right] \right\}$$

www.manaraa.com

Geometrically, $\mathsf{suc}(\rho)$ can be interpreted as the set of subregions ρ' whose cells $\mu \in \mathcal{M}_{\rho'}$ lie all at the left, right, top, or bottom side of slice lines that dissect the point placement of the cells $\mu \in \mathcal{M}_{\rho}$.

All subregions that are needed to compute the shape function of a region ρ are called descendants, building the set $\mathsf{des}(\rho)$:

$$\mathsf{des}(\rho) = \rho \cup \bigcup_{\rho' \in \mathsf{suc}(\rho)} \mathsf{des}(\rho')$$

Using these definitions, we obtain a dependency graph for an enumeration region ϵ as $\mathcal{G}_\epsilon = (\mathcal{V}_\epsilon, \mathcal{E}_\epsilon)$ with

$$\mathcal{V}_\epsilon = \mathsf{des}(\epsilon) \quad \text{and} \quad \mathcal{E}_\epsilon = \{(\rho, \rho') \mid \rho' \in \mathsf{suc}(\rho)\}$$

Figure 6 shows an example of a dependency graph. In this example, the shape function of the subregion with $\mathcal{M}_\rho = \{A, C\}$ can be computed only if the shape functions of the subregions with the cell subsets $\mathcal{M}_{\rho'} = \{A\}$ and $\mathcal{M}_{\rho''} = \{C\}$ are known.

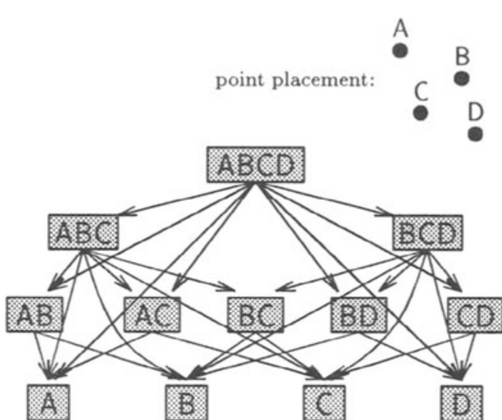


Fig. 6.: Dependency graph for an enumeration region with $\mathcal{M}_\epsilon = \{A, B, C, D\}$

3.3 A nonrecursive algorithm

If we want to parallelize the enumeration of a region, we have to remove the recursion from the algorithm. After calculating all required subregions $\mathcal{V}_\epsilon$ of the enumeration region ρ_ϵ during initialization, we are able to process the dependency graph bottom-up, starting with the subregions containing only one cell and ending at the enumeration region ϵ (see Fig. 7).

The nonrecursive algorithm requires some additional computational effort to determine the nodes of the dependency graph, $\mathcal{V}_\epsilon$. This overhead can be neglected, however, since it grows only with $O(k_\epsilon^4)$, which is two orders below the time complexity of the enumeration algorithm.

www.manaraa.com

```
For all ε ∈ R^e call: Enumerate_nonrecursive(M_ε);


procedure Enumerate_nonrecursive(M_ε)
V_ε = Determine_all_subsets(M_ε);
for i := 1,2...k_ε
   for each ρ ∈ V_ε with k_ρ = i
      for each σ ∈ {h,v}                          /* hor. and vert. cuts */
         M_ρ' := M_ρ; M_ρ'' := ∅; f_ρ := 0;
         while (|M_ρ'| > 1)
            if (σ = v) μ := λ0 | x_λ0 =   min   (x_λ);
                                        λ∈M_ρ'
            else       μ := λ0 | y_λ0 =   min   (y_λ);
                                        λ∈M_ρ'
            M_ρ' := M_ρ'\{μ}; M_ρ'' := M_ρ'' ∪ {μ};
            f_ρ := min(f_ρ; f_ρ'⊚f_ρ'');
         end while
      end for
   end for
end for
return (f_ρ);
end procedure
```

Fig. 7.: Nonrecursive enumeration algorithm

4 Parallel slicing enumeration

4.1 Coarsegrained parallelism

The most obvious way to speed up enumeration is to apply *coarsegrained* parallelism by processing different enumeration regions $\epsilon \in \mathcal{R}^e$ simultaneously.

We use a client-server model of communication with the client sending *enumeration jobs* to the servers where they are processed by one of the above described sequential algorithms. The assignment of enumeration regions to servers is done dynamically: as soon as a server sends a computed shape function back to the client it gets a new enumeration job. Due to the high complexity of the algorithm, the processing time of different enumeration regions varies largely. Therefore, regions that take a long time to enumerate are processed first in order to achieve a good load balancing.

The small communication overhead required by coarsegrained parallelism allows nearly linear speed-ups for sufficiently large k_{max}. A drawback of this scheme is that the maximum size of an enumeration region, i.e. the parameter k_{max}, is still bound by the size of a single processor's memory. On message-passing parallel computers that typically do not have virtual memory, storage constraints allow only for relatively small values of k_{max} and hence for less compact cell arrangements.

www.manaraa.com

4.2 Finegrained parallelism

In environments where memory is limited, it is sensible to parallelize the operations on a single enumeration region so that the memory of all processors can be used for shape function storage. This requires a parallel algorithm of finer granularity.

The parallelization of the nonrecursive slicing algorithm consists of two main parts: the determination of independent subproblems and the assignment of subproblems to processors.

Independent subproblems. If the shape function of each region is initialized to zero, we can define the set of regions whose shape functions are independent and can be calculated in parallel:

$$\mathsf{readyset} = \{\rho \mid f_\rho = 0 \wedge \bigvee_{\rho' \in \mathrm{suc}(\rho)} f_{\rho'} \neq 0\}$$

An efficient way of calculating successive readysets is to assign subregions to readysets according to their number of cells k_ρ, which is the same as the level of the region in the dependency graph (see Fig. 6). We increment the level from 1 to k_ϵ, so that regions with equal k_ρ are processed simultaneously. Using this scheme, there are k_ϵ successive readysets.

Partitioning the dependency graph. When implementing the nonrecursive algorithm on a parallel computer, the nodes of the dependency graph are distributed among the processors. Each processor enumerates the assigned regions and stores the calculated shape function in memory.

On shared-memory machines, the partitioning can be done dynamically, because the processors have access to a common list of nodes that are not enumerated yet. On distributed-memory machines, however, the partitioning has to be calculated in advance to keep communication overhead low. Therefore, each region ρ gets assigned a processor p_ρ during initialization.

This mapping p_ρ of regions to processors influences the algorithm in two ways. If the mapping is unbalanced there will be idle processors most of the time. Therefore, for each readyset, the mapping should try to minimize a *load balancing* criterion, i.e. the variance of the number of regions per processor:

$$C_b = \frac{1}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} \left(\frac{|\mathsf{readyset}|}{|\mathcal{P}|} - |\{\rho \in \mathsf{readyset} | p_\rho = p\}| \right)^2$$

On the other hand, there is a significant communication overhead when shapes have to be requested from other processors. This overhead has to be taken into account by minimizing a *locality* criterion which models the number of these non-local requests:

$$C_l = \sum_{\rho \in \mathcal{V}} \sum_{\rho' \in \mathsf{suc}(\rho)} l_{\rho,\rho'} \quad \text{with} \quad l_{\rho,\rho'} = \begin{cases} 1 & \text{if } p_{\rho'} \neq p_\rho \\ 0 & \text{if } p_{\rho'} = p_\rho \end{cases}$$

www.manaraa.com

An optimum partitioning of the dependency graph must minimize $C_b + \alpha C_l$, with α depending on the relation between communication cost and computing power of the parallel target architecture. Such an optimum partitioning is very hard to compute. We therefore use the following heuristic:

Starting with sets of regions $\mathcal{R} = \textsf{readyset}$ and processors $\mathcal{P}$, the mapping of regions to processors is calculated by recursively bipartitioning both sets breadth first. The $\mathcal{R}$ and $\mathcal{P}$ are bipartitioned into $\mathcal{R}_1, \mathcal{R}_2 \subset \mathcal{R}$ and $\mathcal{P}_1, \mathcal{P}_2 \subset \mathcal{P}$ such that the following equations hold:

$$\forall_{\substack{\rho_1 \in \mathcal{R}_1 \\ \rho_2 \in \mathcal{R}_2}} \sum_{\mu \in \mathcal{M}_{\rho_1}} x_\mu \leq \sum_{\mu \in \mathcal{M}_{\rho_2}} x_\mu \tag{1}$$

$$|\mathcal{P}_1| - |\mathcal{P}_2| \leq 1 \text{ and } \left| \frac{|\mathcal{R}_1|}{|\mathcal{P}_1|} - \frac{|\mathcal{R}_2|}{|\mathcal{P}_2|} \right| = \min \tag{2}$$

The sorting according to the center of gravity coordinates defined by equation 1 is a good heuristic to keep criterion C_l small. Equation 2 assures that criterion C_b is minimized. When bipartitioning the two subsets $\mathcal{R}_1$ and $\mathcal{R}_2$, the coordinate direction is alternated. The bipartitioning with alternating coordinate directions is continued recursively until the number of region subsets equals $|\mathcal{P}|$.

5 Implementations and results

We implemented the parallel slicing enumeration on three different target architectures and used it to generate placements of several macro cell benchmark circuits [15] as well as industrial sea-of-gates designs [6].

5.1 Workstation network

This "parallel computer" consists of up to 20 DECstations connected by ethernet and running under the Unix operating system. They communicate using the p4 parallel programming library [16] based on the Unix socket mechanism.

Every node of this distributed environment has a large virtual memory and a powerful processor. The throughput of the communication network, however, is rather limited. Therefore, we use the principle of coarsegrained slicing enumeration: a master process running on one workstation distributes the enumeration regions among the other workstations.

The speed-up of this approach, as defined by the ratio of the cpu time of the sequential program and the real time required by the parallel program, can be seen in Fig. 8 for a representative sea-of-gates circuit. The speed-up increases linearly up to about 10 workstations where it levels off, while the variation of the runtimes grows. The sublinear speed-up for large numbers of workstations can be accounted for by the limited bandwidth of the communication network, which leads to high latencies when many workstations are used. The increasing variation in runtime is due to the fact that the workstations have different computing power and workloads: when more workstations are used, there is an

المنارة للاستشارات
www.manaraa.com

increased possibility that a workstation with a high load or a slow processor has to be used.

Note that while high speed-ups can be achieved using this setup, the maximum useful value of k_{max} and hence the achievable result quality is still limited by the amount of memory in one single machine.

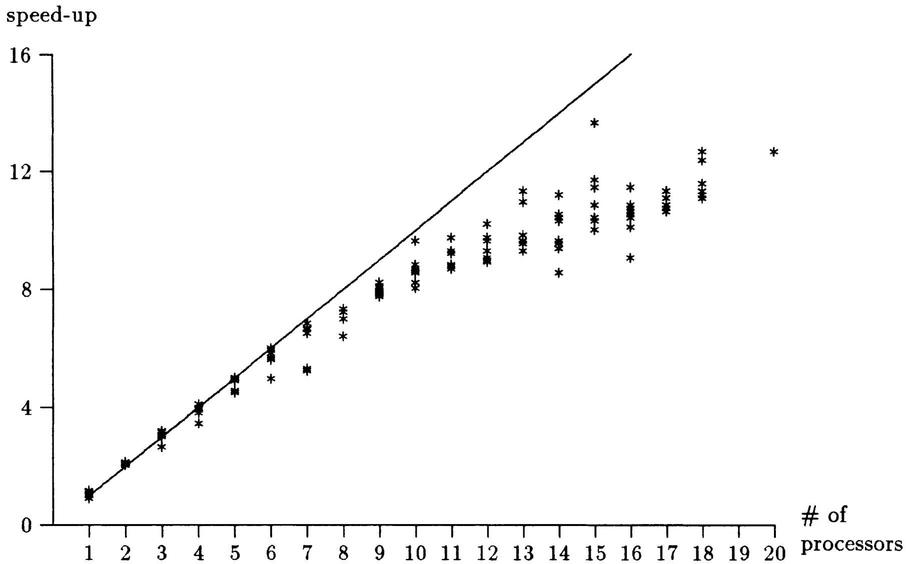


Fig. 8.: Speed-up of the workstation network for circuit sog3 and $k_{max} = 30$

5.2 Shared-memory machine

Our second architecture is a shared memory machine, a Sequent Symmetry with eight i386 processors and a total of 40 megabytes of RAM. The operating system is Dynix, an Unix variant with multiprocessor extensions. Besides the usual Unix interprocess communication mechanism, it provides special facilities for microtasking where multiple threads execute the same process [17].

The approach using coarsegrained parallelism is not practical on this machine. Because all processors use the same memory and swap-disks, evaluating n enumeration regions in parallel requires n times more memory, which leads to excessive swapping and therefore to low speed-ups for high values of k_{max}.

The memory is more efficiently utilized by applying finegrained parallelism. As described in the preceding section, the processors work together using the non-recursive slicing algorithm: the dependency graph is processed bottom-up, each processor computing the shape function of some of the nodes on the current level. After the completion of each level, the processors have to synchronize to

www.manaraa.com

make sure that all shape functions that are required on the next level have already been calculated.

The speed-up for the implementation on this machine using a macro-cell circuit can be seen in Fig. 9. The speed-up is sublinear because some parts of the slicing enumeration like initialization are not parallelized and because the shared-memory machine could not be used exclusively in our test runs.

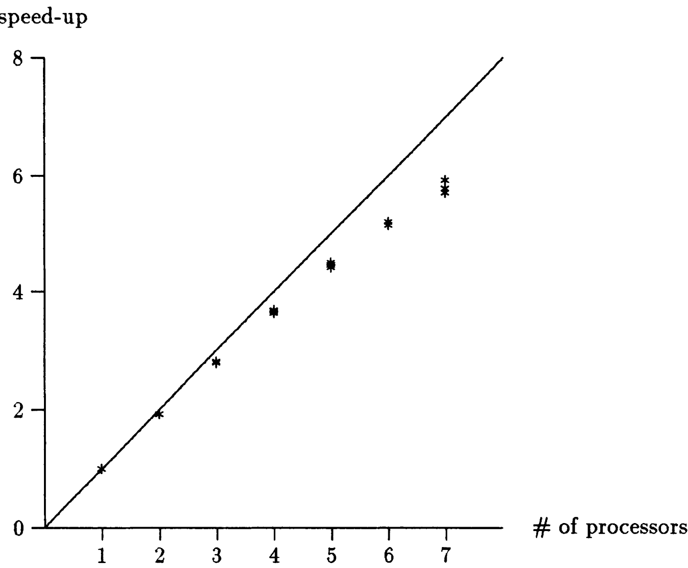


Fig. 9.: Speed-up of the shared-memory machine for circuit ami49 and $k_{max} = 25$

5.3 Distributed-memory machine

The third architecture is an iPSC/2 hypercube with 32 nodes, each containing an i386 processor and 4 MB of RAM, running under the MMK operating system [18]. This type of machine is characterized by a limited amount of memory per processor, the lack of virtual memory, and a fast communication network.

Because of the memory limitation, a single processor of the hypercube can only enumerate regions containing at most approximately 20 cells. Therefore, we have to use the finegrained approach, using the distributed memory of multiple processors for shape function storage.

The nodes of the dependency graph are mapped to the processors using the partitioning scheme presented in section 4.2. There are two lightweight tasks on every processor. The *compute task* works on the dependency graph bottom-up, calculating the shape functions and storing them in local memory. If during enumeration the shape function of a region that is kept on another processor is required, it is requested from that processor's *server task*.

www.manaraa.com

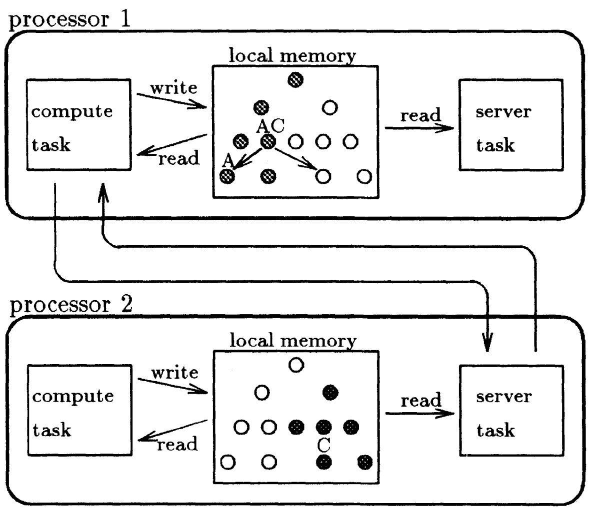


Fig. 10.: Processor tasks of finegrained parallel enumeration

In Fig. 10, two processors are working together on the dependency graph shown in Fig. 6. The nodes assigned to a specific processor are shown in gray. The compute task of processor 1 is currently computing the shape function of region AC. To do that, it requires shape C and thus has to send a request to the server task of processor 2, which fetches that shape from its local memory and sends a message back to processor 1.

Because of its inherent communication overhead, the distributed-memory finegrained parallelism usually yields a lower speed-up than the coarsegrained approach. Therefore, we use a combination of both, allowing good speed-ups as well as large k_{max} values.

The combined algorithm is similar to the coarsegrained approach, however a single processor is replaced by a *cluster* of processors. The processors within a cluster are working together on a single enumeration region using finegrained parallelism. For each region, the cluster size is dynamically adapted to the amount of memory required for the enumeration of that specific region. In this way, we combine both the high speed-up of coarsegrained parallelism and the high achievable quality of finegrained parallelism.

In order to calculate speed-ups for the hypercube, we needed execution times for the sequential algorithm. Because a single node of the hypercube cannot enumerate regions with $k_\epsilon > 20$, we cannot measure the sequential execution time on the hypercube and have to use the nonrecursive algorithm running on a sequential computer with the same processor and clock frequency as the hypercube as a reference.

The speed-up of the parallel algorithm for the industrial sea-of-gates circuit

www.manaraa.com

sog3 is shown in Fig. 11. For small values of k_{max}, the time to process an enumeration job is in the order of the delay involved in sending it from the host workstation to a processor cluster, resulting in poor speed-up. For $k_{max} > 25$, however, the speed-up is approximately half the number of processors used. Note that the minimum number of processors depends on the value of k_{max} due to the high memory requirements.

In contrast to the other two approaches presented, the speed-up does not level off for a high number of processors. This indicates that this algorithm has a good scalability for larger processor counts.

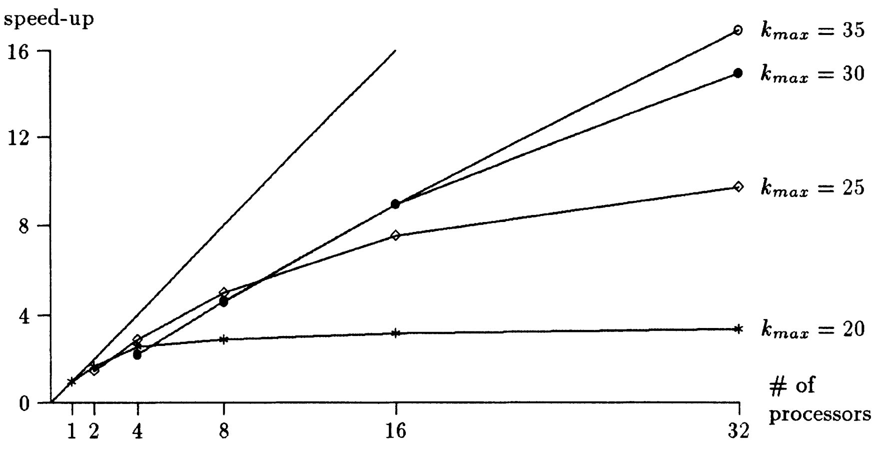


Fig. 11.: Speed-up of the hypercube for circuit sog3

6 Conclusions

As a result of our investigations, we can state that the problem of slicing enumeration is well suited for execution on parallel computers.

We are able to parallelize both the processing of different enumeration regions and the calculations within a single enumeration region. This allows for a good adaption of the procedure to different parallel architectures.

The reached speed-ups are, depending on the target architecture, between 0.5 and 1.0 times the number of processors used. The approach using finegrained parallelism together with distributed memory promises a good scalability together with the capability to enumerate larger cell subsets, yielding a higher layout quality than on sequential computers.

Thus, the parallelization of the final placement problem allows us to cope with the rapidly increasing circuit complexity of modern designs by achieving excellent results in short time.

www.manaraa.com

Acknowledgements

The authors would like to thank Prof. K. Antreich for his support and valuable suggestions.

References

1. C. Sechen, *VLSI Placement and Routing Using Simulated Annealing.* Kluwer Academic, 1988.
2. U. Lauther, "A min-cut placement algorithm for general cell assemblies based on a graph representation," *16th DAC*, pp. 1–10, 1979.
3. D. P. LaPotin and S. W. Director, "Mason: A global floorplanning approach for VLSI design," *IEEE Trans. on CAD*, vol. 5, no. 4, pp. 477–489, 1986.
4. W.-M. Dai and E. S. Kuh, "Simultaneous floor planning and global routing for hierarchical building-block layout," *IEEE Trans. on CAD*, vol. 6, no. 5, pp. 828–836, 1987.
5. L. Sha and R. W. Dutton, "An analytical algorithm for placement of arbitrarily sized rectangular blocks," *22nd DAC*, pp. 602–608, 1985.
6. J. M. Kleinhans, G. Sigl, and F. M. Johannes, "GORDIAN: A new global optimization / rectangle dissection method for cell placement," *ICCAD*, pp. 506–509, 1988.
7. A. A. Szepieniec and R. H. J. M. Otten, "The genealogical approach to the layout problem," *17th DAC*, pp. 164–170, 1980.
8. R. H. J. M. Otten, "Efficient floorplan optimization," *ICCD*, pp. 499–501, 1983.
9. L. Stockmeyer, "Optimal orientations of cells in slicing floorplan designs," *Information and Control*, vol. 57, pp. 91–101, 1983.
10. L. P. P. P. van Ginneken and R. H. J. M. Otten, "Optimal slicing of plane point placements," *EDAC*, pp. 322–326, 1990.
11. H. Spruth and G. Sigl, "Parallel algorithms for slicing based final placement," *Euro-DAC*, pp. 40–45, 1992.
12. G. Zimmermann, "A new area and shape function estimation technique for VLSI layouts," *25th DAC*, pp. 60–65, 1988.
13. J. M. Kleinhans, G. Sigl, F. M. Johannes, and K. J. Antreich, "GORDIAN: VLSI placement by quadratic programming and slicing optimization," *IEEE Transactions on Computer-Aided Design*, vol. 10, no. 3, pp. 356–365, 1991.
14. G. Sigl and U. Schlichtmann, "Goal oriented slicing enumeration through shape function clipping," *EDAC*, pp. 361–365, 1991.
15. MCNC International Workshop on Layout Synthesis, Microelectronics Center of North Carolina, Research Triangle Park, NC, 1990.
16. E. L. R. Butler, "User's guide to the p4 parallel programming system," 1990.
17. A. Osterhaug, *Guide to Parallel Programming on Sequent Computer Systems.* Sequent Computer Systems, 1987.
18. T. Bemmerl and T. Ludwig, "MMK - a distributed operating system kernel with integrated dynamic loadbalancing," *CONPAR 90 - VAPP IV Conference*, 1990.

www.manaraa.com

Application of Fault Parallelism to the Automatic Test Pattern Generation for Sequential Circuits

Peter A. KRAUSS and Kurt J. ANTREICH

Institute of Electronic Design Automation
Technical University of Munich, Germany

Abstract. The task of automatic test pattern generation for sequential circuits is to find test sequences which detect a difference between the faulty and the fault-free circuit. Since this task typically requires considerable computational resources, it provides a challenging application for parallel computer architectures.

The approach proposed here considers fault parallelism, supplying every processor with a certain number of target faults, depending on the job size. Every processor does the test pattern generation and subsequent fault simulation for its faults and then returns back the generated test sequence and all target faults detected by this test sequence to a controlling process.

The algorithm for partitioning the set of target faults among the number of available processors depends critically on the following three criteria: computation time, fault dependency, and job size.

We implemented our approach on various computer architectures, such as an iPSC/860 HyperCube and networks of workstations (DEC and HP). All implementations use the server/client communication model.

The parallel test pattern generation algorithm has been validated with the ISCAS'89 benchmark circuits, and we achieved a nearly linear speed-up.

1 Introduction

The circuit has to be tested after its manufacture. A conceptual approach to testing is shown in Fig. 1.

The circuit under test is a sequential circuit with memory elements and requires that test patterns are applied in a specific order. Using a set of test sequences, applied to both the manufactured circuit and the circuit model which is simulated, we compare the responses at the respective outputs. If both outputs are equal, the circuit is considered to be fault-free, otherwise the circuit is faulty [1].

We model the circuit at the gate level. The faults we consider are single stuck-at faults. Under this assumption there exists only one fault within the circuit, this fault being a constant 0 or a constant 1.

The test sequences should detect all modeled faults and report a fault coverage of 100% if feasible. Furthermore, the test sequences should be as short

www.manaraa.com

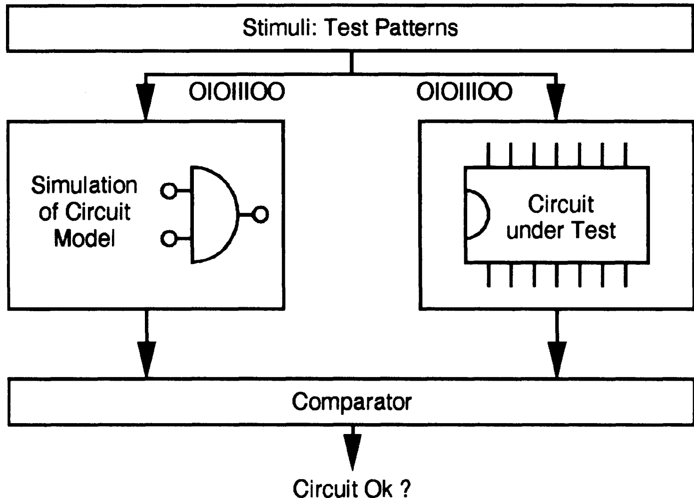


Fig. 1. Testing a Circuit

as possible to reduce the test application time, thereby lowering the production costs.

The generation of the test sequences can be done in several ways. For sequential circuits, a combination of the deterministic test generation and the fault simulation has proven to be most efficient.

However, it has been shown that the problem of test generation is NP-hard even for combinational circuits [2, 3]. Larger circuits have a computation time of several days, sometimes even weeks.

It will be shown, that the usage of parallel computers may help to speedup the automatic test pattern generation process.

This article is organized as follows. Section 2 introduces the definitions and preliminaries. The basic principles of the automatic test pattern generation are described in Section 3. Section 4 presents the application of fault parallelism and Section 5 discusses the achieved results. Conclusions are summarized in the last section.

2 The Test Problem

Integrated circuits provide no access to their internal signals. Therefore, the excitation of a fault and the observation of its effect has to be done by using only the external connections.

The problem of automatic test pattern generation may be formulated with help of a deterministic Finite State Machine (FSM) model, shown in Fig. 2.

www.manaraa.com

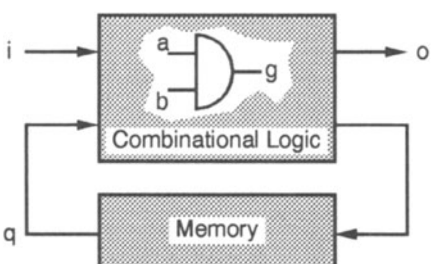


Fig. 2. Finite State Machine

The fault-free FSM A becomes the faulty FSM A^μ due to a fault μ:

$$A = (Q, \Sigma, \Delta, \delta, \lambda) \quad \overset{\text{fault}}{\Longrightarrow} \quad A^\mu = (Q, \Sigma, \Delta, \delta^\mu, \lambda^\mu) \tag{1}$$

$$\begin{array}{ll} Q = \{q_0, q_1, \ldots, q_r\} & \text{: set of states} \\ \Sigma = \{i_0, i_1, \ldots, i_n\} & \text{: input alphabet} \\ \Delta = \{o_0, o_1, \ldots, o_m\} & \text{: output alphabet} \\ \delta \;:\; Q \times \Sigma \rightarrow Q & \text{: transition function} \\ \lambda \;:\; Q \times \Sigma \rightarrow \Delta & \text{: output function} \end{array}$$

The task of automatic test pattern generation is to find an input sequence which excites the fault on the internal gate and assures the observation on at least one output:

$$\exists_{\substack{t \in \Sigma^* \\ z \in \Sigma}} \quad \forall_{\substack{q_0 \in Q \\ q_0^\mu \in Q}} \quad \lambda(\delta(q_0, t), z) \neq \lambda^\mu(\delta^\mu(q_0^\mu, t), z) \tag{2}$$

Σ^* denotes sequences constructed by concatenating any number of test patterns from Σ [4]:

$$\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i, \quad \Sigma^0 = \{\varepsilon\}, \quad \varepsilon : \text{empty string} \tag{3}$$

The condition (2) indicates that there exists at least one test sequence for which the output function λ of the fault-free FSM A produces a different result than the output function λ^μ of the faulty FSM A^μ after having read the last test pattern z, independent of the initial state of the fault-free FSM A and of the faulty FSM A^μ.

The complement of (2) gives the condition for untestable faults:

$$\forall_{\substack{t \in \Sigma^* \\ z \in \Sigma}} \quad \exists_{\substack{q_0 \in Q \\ q_0^\mu \in Q}} \quad \lambda(\delta(q_0, t), z) = \lambda^\mu(\delta^\mu(q_0^\mu, t), z) \tag{4}$$

Here, for all possible test sequences, there exists at least one initial state of the fault-free FSM A and one of the faulty FSM A^μ, where the output functions λ and λ^μ evaluate to the same value.

www.manaraa.com

3 The Automatic Test Pattern Generation

The automatic test pattern generation is performed in several phases [5, 6], as shown in Fig. 3.

First, we read the circuit netlist. In the preprocessing phase, we analyze the circuit and compile a list of the faults which are to be considered. These faults are designated as the *target faults* [7].

During the preprocessing phase some faults are recognized as untestable. These faults can not be tested by any test sequence, possibly because of some circuit redundancies. Therefore, these untestable faults can be removed immediately from the target fault list.

The automatic test pattern generation is performed for the remainder of the target faults. We consider one fault at a time, and attempt to generate a test sequence which detects this fault.

The test pattern generation is based on a search in a decision tree, using a heuristic to control the backtracking.

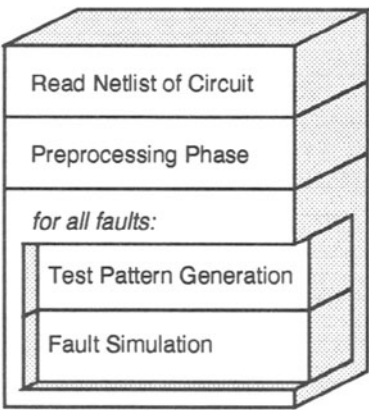


Fig. 3. Automatic Test Pattern Generation

Figure 4 shows the symbolic representation of a decision tree, which is divided into three areas: the *potential solution area* (white), the *known non-solution area* (dark), and the *unknown non-solution area* (hatched) [8].

The algorithm starts at the top of the triangle, trying to find a path to one of the solutions, which are located at the bottom of the triangle. When reaching the known non-solution area, a backtracking is done to return into the potential solution area. The unknown non-solution area is the region, where it is not yet known that the algorithm will not find a solution. It is the task of heuristics to keep the unknown non-solution area as small as possible, thereby reducing the number of further wrong decisions and subsequent backtracks. Every additional backtrack implies wasted computational resources.

www.manaraa.com

Fig. 4. Symbolic representation of the decision tree

If the search takes too much time or costs too much memory, it is aborted. A successful search returns either of the two results:

- a test sequence was generated, or
- the fault was detected to be untestable.

If a test sequence was generated, a fault simulator determines subsequently additional faults as also detectable by this test sequence. The computation time of a fault simulation is much less than the computation time of a test pattern generation, thereby accelerating the overall automatic test pattern generation process considerably.

All detected faults are removed from the target fault list. This process is repeated until the list is empty.

4 The Fault Parallelism

There are different possibilities to parallelize the automatic test pattern generation process. Fault parallelism for combinational circuits has been considered in [9]. The approach proposed here considers fault parallelism for sequential circuits.

The fault parallelism implies a distribution of the target fault list among the available processors. This can be done in two ways. The target fault list can be managed either by the processors themselves or by a central control processor.

4.1 The Distributed Self-Management

A communication model best suited for the self-management approach is one where all processors are fully meshed, as shown in Fig. 5.

In this approach, the target fault list is initially divided among the working processors, named *workers*, by using some criteria. A worker which finishes its assignment and becames idle, sends a request to the non-idle workers for a part of their unprocessed fault list.

As the run nears completion, the number of messages sent between the processors will increase considerably because of the shrinking size of the fault lists. This can be avoided by partitioning the lists only to a given granularity [10].

www.manaraa.com

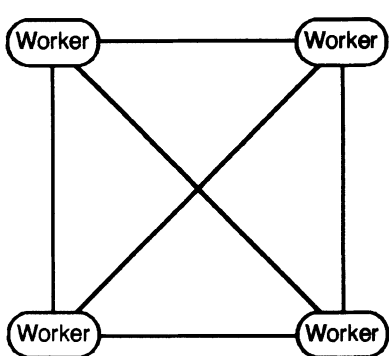


Fig. 5. Fully meshed working processors

4.2 The Central Control Management

The central control management is realized with a communication model based on a star topology, as shown in Fig. 6.

Here, a central control processor, named *controller*, keeps the whole target fault list. An idle worker requests one or more faults from the controller. Therefore, the controller has the possibility to select the faults which are to be processed next according to following criteria:

- fault dependency,
- selected job size, and
- estimated computation time.

We discuss the fault dependency in the next section.

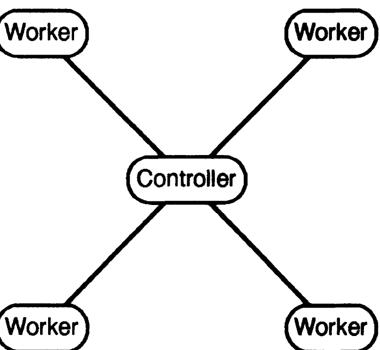


Fig. 6. Using a central control processor

www.manaraa.com

4.3 The Fault Dependency

The concept of fault dependency is illustrated in Fig. 7. Given one target fault φ, the test pattern generation *ATPG* produces a test sequence t. The fault simulation *FSIM* uses this sequence t to determine additional faults which are also detected by this sequence.

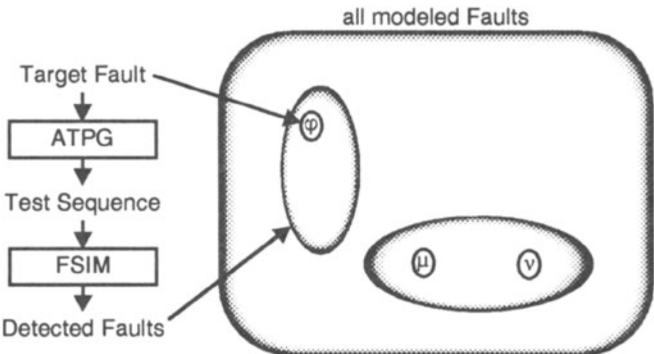


Fig. 7. Fault Dependency

Faults, which are detected by the same test sequence are called dependent on each other:

$$\exists_{\substack{t \in \Sigma^* \\ z \in \Sigma}} \exists_{\substack{q_0 \in Q \\ q_0^\mu \in Q \\ q_0^\nu \in Q}} [\lambda(\delta(q_0,t),z) \neq \lambda^\mu(\delta^\mu(q_0^\mu,t),z) \wedge \lambda(\delta(q_0,t),z) \neq \lambda^\nu(\delta^\nu(q_0^\nu,t),z)] \tag{5}$$

This dependency may be partially calculated during the preprocessing phase.

If one processor works on the fault μ, while another processor finds a test sequence for another fault ν which detects also μ, the computation time spent on the fault μ is wasted.

Therefore, dependent faults should not be distributed to different processors at the same time. Hence, the dependency relations obtained during the preprocessing phase have to be considered [9, 10, 11].

In addition, by supplying only one single fault to the requesting processor, we further reduce the chance of processing the same faults by multiple processors. Starting with moderately large circuits, the computation time for each fault is much higher than the communication time. This is an additional reason, why we can improve performance by distributing single faults only.

www.manaraa.com

4.4 Implementation Details

As mentioned before, an idle worker requests one fault for processing from the controller. The result sent back to the controller is either the eventually generated test sequence together with faults detected from this sequence, or this fault is reported untestable. A third possible reply is, that the search was aborted because of excessing some resource limits.

The controller removes all reported faults from its target fault list, including the aborted faults. Next, a new target fault is given to the worker that has just completed the previous task.

This method of requesting workers has several advantages:

1. As already mentioned, the controller has the possibility to select the job sequence by various criteria.
2. A dynamic load balancing is achieved automatically. The generation of the test sequences for different faults needs different computation time. It is not possible to calculate this time in advance. An estimation by some heuristics could be done [9]. However, the computation time may also vary because of some load from other processes on the same processor. Also, the usage of different hardware results in different computation times.
3. If a worker fails, it will not block the other workers and the controller. While the other worker will continue to work, the task given to the failed worker will be reassigned to another worker later on. Hence, nothing is lost by the failure of some workers. The controller terminates after having a result for each fault of its list.

5 Results

The parallel test pattern generation algorithm has been validated with the ISCAS'89 benchmark circuits [7].

5.1 Speedup on the HyperCube

Results achieved on an Intel iPSC/860 HyperCube are shown in Fig. 8. The diagram illustrates the relationship between the number of workers, the total sum of computation time for the test generation and subsequent fault simulation for different circuits, and the achieved speedup.

The speedup used here is the relation of the computation times of one worker to n workers. Every line running from left to right marks the computation times of the same circuit (e.g. s1423) for a growing number of workers. Every line running from the front to the back marks the computation times of different circuits for the same number of workers.

As can be seen in the diagram, the achieved speedup of the larger circuits is being kept when the number of workers grows. These circuits reach an almost linear speedup. With 15 workers we achieve a speedup of 14.5.

www.manaraa.com

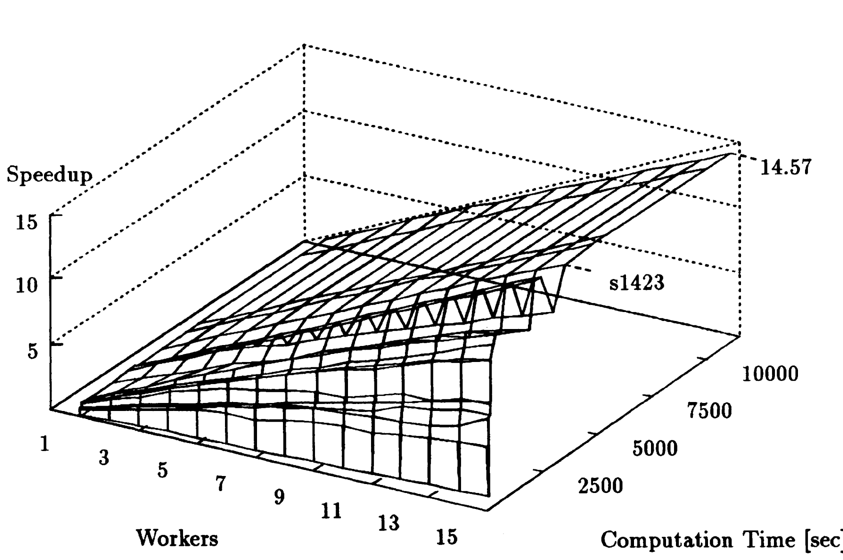


Fig. 8. Speedup in relation to the number of workers and computation time, achieved on the HyperCube

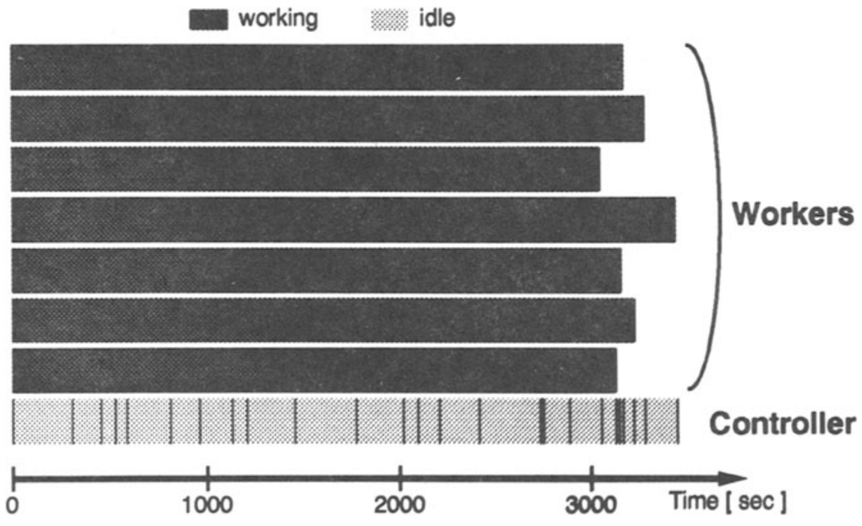


Fig. 9. Usage of the HyperCube

www.manaraa.com

Of interest is also the comparison of the computation time of different circuits with the achieved speedup at a constant number of workers. With growing computation time the almost linear speedup is achieved very quickly.

The observable break-ins will be discussed later on.

5.2 Usage of the HyperCube

Figure 9 shows a typical usage of all processors during test generation and fault simulation. All workers are running without any waiting times, while the controller works only at the few times which are marked by dark lines. Those messages show that the controller has only very little load from serving the workers.

We see the case where more than one worker is ready and has to wait until the controller can serve it only rarely. Also, the next fault is selected very fast, so that the interruption of the worker is minimal. Therefore, the controller is capable of handling a much larger number of workers.

5.3 Speedup on the Workstations

Results achieved on a network of workstations are shown in Fig. 10. The network consists of 110 Workstations HP 9000/720. Like Fig. 8, the diagram shows the relationship between the number of used workers, the total sum of computation time for the test generation and subsequent fault simulation for different circuits, and the achieved speedup. Using 100 workers, we achieve a maximal speedup of 88.

The already mentioned break-ins are clearly visible here. These circuits have a higher computation time per fault than the circuits which form the rising slope, but the total number of faults is too small to gain a dynamic load balancing on the number of used workers. The computation time of some faults dominate over the rest, they set the time needed for their circuit independent of the number of used workers. The speedup is saturated already at low number of workers.

6 Conclusion

We have demonstrated that the automatic test pattern generation is well suited for application of parallel methods. Our experiments were successful not only for special parallel computers, but also for networks of workstations.

The ongoing research is considering next the parallelism of the search tree, which belongs to the area of OR-parallelism. Previous works done for combinational circuits have reported good results [12].

Using fault parallelism, one processor handles one fault, where as the OR-parallelism takes all processors to work on the solution for one fault. This allows the search for the *hard-to-detect* faults, a case where the search based on earlier methods had to be aborted.

www.manaraa.com

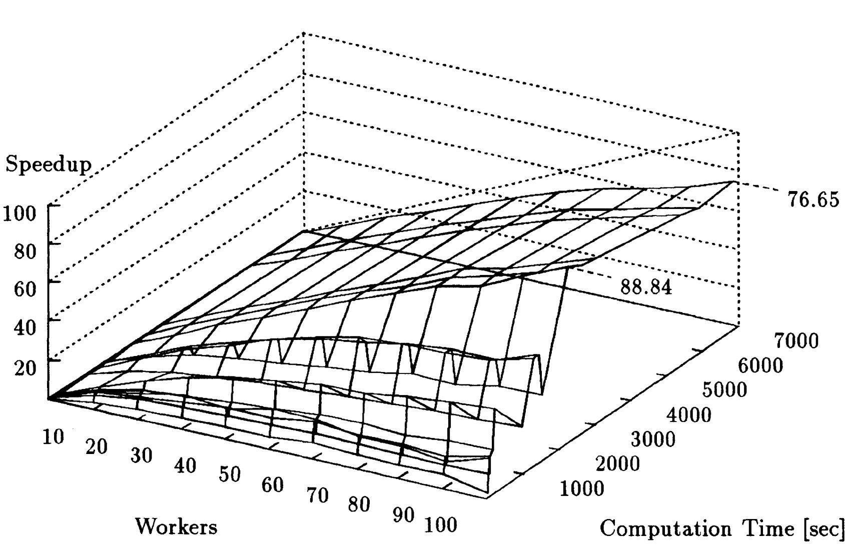


Fig. 10. Speedup in relation to the number of workers and computation time, achieved on the network of workstations

References

1. E. Hörbst, M. Nett, and H. Schwärtzel. *Venus Entwurf von VLSI-Schaltungen.* Springer-Verlag, Berlin, 1986.
2. O. H. Ibarra and S. K. Sahni. Polynomially complete fault detection problems. *IEEE Transactions on Computers*, pp. 242–249, 1975.
3. Hideo Fujiwara and S. Toida. The complexity of fault detection problems for combinational logic circuits. *IEEE Transactions on Computers*, pp. 555–560, 1982.
4. John Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addision-Wesley, Reading, 1979.
5. Michael H. Schulz and Elisabeth Auth. ESSENTIAL: An efficient self-learning test pattern generation algorithm for sequential circuits. *Proceedings International Test Conference*, pp. 28–37, 1989.
6. Elisabeth Auth and Michael H. Schulz. A test-pattern generation algorithm for sequential circuits. *IEEE Design & Test of Computers*, pp. 72–86, 1991.
7. Franc Brglez, David Bryan, and Krzysztof Kozminski. Combinational profiles of sequential benchmark circuits. *IEEE International Symposium on Circuits and Systems*, pp. 1929–1934, 1989.
8. Michael H. Schulz, E. Trischler, and T. M. Sarfert. SOCRATES: A highly efficient automatic test pattern generation system. *Proceedings International Test Conference*, pp. 1016–1025, 1987.
9. Hideo Fujiwara and Tomoo Inoue. Optimal granularity of test generation in a distributed system. *IEEE Transactions on Computer-Aided Design*, pp. 885–892, 1990.

www.manaraa.com

10. Srinivas Patil and Prithviraj Banerjee. Fault partitioning issues in an integrated parallel test generation / fault simulation environment. *IEEE Proceedings International Test Conference*, pp. 718–726, 1989.
11. Sheldon B. Akers and Balakrishnan Krishnamurthy. Test counting: A tool for VLSI testing. *IEEE Design & Test of Computers*, pp. 58–73, 1989.
12. Srinivas Patil and Prithviraj Banerjee. A parallel branch and bound algorithm for test generation. *IEEE Transactions on Computer-Aided Design*, pp. 313–322, 1990.

www.manaraa.com

Parallel Sorting of Large Data Volumes on Distributed Memory Multiprocessors

Markus Pawlowski, Rudolf Bayer

Institut für Informatik, Technische Universität München
Arcisstr. 21, D 8000 München 2, Germany
e-mail: {pawlowsk | bayer}@informatik.tu-muenchen.de

Abstract. The use of multiprocessor architectures requires the parallelization of sorting algorithms. A parallel sorting algorithm based on horizontal parallelization is presented. This algorithm is suited for large data volumes (external sorting) and does not suffer from processing skew in presence of data skew. The core of the parallel sorting algorithm is a new adaptive partitioning method. The effect of data skew is remedied by taking samples representing the distribution of the input data. The parallel algorithm has been implemented on top of a shared disk multiprocessor architecture. The performance evaluation of the algorithm shows that it has linear speedup. Furthermore, the optimal degree of CPU parallelism is derived if I/O limitations are taken into account.

1 Introduction

Data sorting plays an important role in computer science and has been studied extensively [23]. The problem of sorting is easily understood. In the sequential case, sorting of N tuples (i.e. data items) has at least a complexity of $O(N \cdot \log(N))$. Sorting is frequently the basis of more complex operations. The use of multiprocessor architectures renders the parallelization of sorting indispensable. First approaches were sorting networks introduced by Batcher [3]. Since then a lot of literature concerning parallel sorting has been published. A complete survey of this literature is beyond the scope of this paper. However, three fundamental approaches can be recognized in order to parallelize a sorting algorithm. In the following these approaches are discussed to obtain a general knowledge of them:

- One possibility to implement parallel sorting algorithms is to implement them in hardware. The progress in VLSI makes this approach promising. Although the architecture of parallel sorting chips differs in details, the paradigm used is *vertical parallelism* (cf. section 2). In [17] simple sort-merge-cells are placed in a binary tree topology in order to build a powerful parallel sorting hardware. The scheme of the rebound-sorter of [13] is used in [18] to design a parallel sorting chip. A rebound-sorter consists of 2 coupled arrays passed by the data streams. A rebound-sorter is comparable to systolic algorithms.
- The second approach has its origin in abstract parallel machine models like the PRAM (Parallel Random Access Machine). A parallel implementation of the merge-sort algorithm is proposed by Cole [14]. It sorts N tuples in time $O(\log(N))$ using N processors of a CREW-PRAM (Concurrent Read Exclusive Write PRAM). Other authors (e.g. Bilardi et al. [10]) modify the so-called bitonic sorting algorithm (cf. Batcher [3]) to achieve a parallel sorting algorithm. It sorts N tuples in time $O(N/p \cdot \log(N/p))$ using p processors. In contrast to the CREW-PRAM algorithm of Cole [14], here the degree of parallelism is not dependent on the number of input tuples. The algorithms of this class can be implemented on a shared memory multiprocessor.

www.manaraa.com

- Distributed memory multicomputers are the starting point of the third approach. Limitations of real systems are taken into consideration. Furthermore, the influence of a memory hierarchy (main memory vs. disk memory) is investigated. The literature contains lot of proposals (e.g. [4], [7], [16], [22] and [26]) on how to sort large volumes of data in parallel on top of a distributed memory multicomputer. The principle of the parallel sorting algorithms is similar and has its root in the well known external sorting algorithms as described by Knuth [23] or Aho et al. [1].

We designed a new parallel sorting algorithm belonging to the third class. The reasons are the following:

- Our prime premise is to use off-the-shelf hardware in order to explore possibilities of parallelism for database management systems. Therefore we do not consider the first approach. Another premise is to investigate the impact of massive parallelism on database management systems. Therefore shared memory multiprocessors (second approch) will not be used, as they do not scale well enough for our purposes.
- Sorting plays an important role during query evaluation [24]. The relational database system TransBase [33], which we use in our research project [25], implements duplicate elimination and a variant of join (sort merge join) by sorting the input operands. Since the database itself resides on secondary storage and query evaluation is done in main memory, the interaction between main memory and secondary memory has to be taken into account. This rules out the first and second approaches, too.
- A further reason is our cooperation with another research group providing the programming environment *MMK* [8] and *TOPSYS* [9] on top of a distributed memory multicomputer. In this way feedback can be obtained about the suitability of *MMK* and *TOPSYS* for developing parallel programs.

Our new parallel external sorting algorithm is based on data parallelism. Each processor sorts one data partition. Load imbalance is avoided by a new, efficient data partitioning algorithm. It does not sample the input data as in [16]. It samples the presorted initial runs. Hence it can limit the maximum load imbalance due to data skew which might occur. Therefore the partitioning algorithm is a component of our parallel sorting algorithm.

The remainder of the paper is organized as follows: Section 2 gives a brief introduction into the paradigms of parallel programming. In section 3 the horizontal paradigm is applied to the problem of parallel sorting. A new adaptive partitioning algorithm is introduced in section 4. Section 5 shows the implementation and performance results. The paper concludes in section 6 with a summary and future work. The appendix puts together the parameters used throughout this paper.

2 Paradigms of Parallel Programming

Horizontal parallelism is based on data-partitioning and works as follows: The input data is partitioned into independent data partitions (*splitting phase*). Each partition is subsequently processed by a single processor performing the usual sequential algorithm (parallel *processing phase*). Then the intermediate results computed by the parallel processing phase are collected into one final result (*integration phase*). Horizontal parallelism accellerates the sequential algorithm, if the overhead due to the additional splitting and integration phases is low and if load imbalance does not result from

www.manaraa.com

unequally sized data partitions.

Vertical parallelism exploits the fact that a complex task can be divided into several subtasks which have to be processed in a pipelined manner (i.e. temporally overlapped). By that means a parallel execution is obtained. The parallel processing time is reduced, if the subtasks have nearly the same processing time and if the communication costs are low w.r.t. the processing costs of one subtask.

Figure 1 compares horizontal parallelism with vertical parallelism.

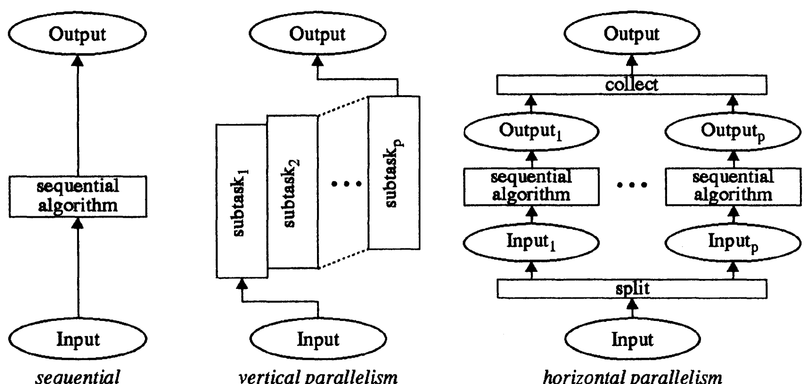


Fig. 1. Vertical and Horizontal Parallelism

3 Parallel External Sorting

The horizontal parallelization is applied to an algorithm performing external sorting. To have a common starting point, the usual sequential algorithm is described. After having presented the multiprocessor architecture of the target machine, first performance predictions of a parallel external sorting algorithm are presented. Finally the sequential sorting algorithm is parallelized horizontally. The paradigm of vertical parallelism is not applied, because it is not practicable.

3.1 Sequential External Sorting Algorithm

Sorting of tuples normally takes place in main memory. But if the amount of data to be sorted exceeds the capacity of the available main memory, the use of secondary storage is inevitable. The number of I/O accesses should be minimized. Most of the sorting process still takes place in main memory. For the sake of simplicity, we assume that the input tuples to be sorted are located in a file on secondary storage (input file) and that the sorted result tuples are stored in a file (result file), too. External sorting consists of two phases [23]. Refering to figure 2, the two phases are explained in detail.

During a so-called *presort phase* main memory is organized as a heap providing space for h tuples. All input tuples are pumped through main memory once, in the course of which they are presorted and stored in m so-called *initial runs* IR_i $(1 \le i \le m)$. The size of each initial run is as large as the heap in main memory at least. We can estimate m by the following formula:

$$m \le N/h \qquad (1)$$

www.manaraa.com

The CPU-time complexity of the presort phase is $O(N \cdot \log(h))$. This results from the fact that each tuple has to be inserted into and deleted from the heap exactly once. The I/O complexity of $O(N)$ during the presort phase is caused by the need of loading (storing) each tuple from (into) secondary storage exactly once.

During a so-called ***merge phase*** the m initial runs IR_i are merged to compute the sorted result file. We use m-way merging to minimize the I/O accesses. Details can be found in [5] and [6]. The I/O complexity of the merge phase is equal to the I/O complexity of the presort phase, since again each tuple has to be loaded (stored) from (into) secondary storage exactly once. The CPU-time complexity of the merge phase is $O(N \cdot \log(m))$. It is not linear in the problem size, since the number of initial runs depends linearly on the problem size.

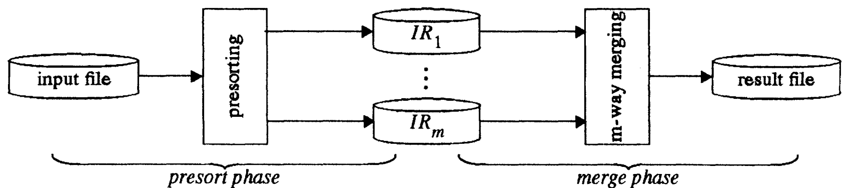


Fig. 2. External Sorting (Sequential Version)

3.2 Computer Architecture

Our target architecture is a shared disk multiprocessor. This architecture belongs to the family of distributed memory multiprocessors, which can be divided into shared disk [30] and shared nothing multiprocessors [31]. The structure of the two distributed memory multiprocessor architectures is shown in figure 3. Distributed memory multiprocessors consist of multiple nodes connected by a fast communication network. A processor and local main memory belong to each node. The processors can only access their local memory directly. They cannot access remote memory of other nodes. A global shared memory does not exist. Interprocessor communication is realized by message passing. Shared disk and shared nothing multiprocessors differ in the way how the secondary storage is connected to the processing elements.

Unlike main memory, secondary storage of shared disk multiprocessors is a subsystem of its own. Shared disk multiprocessors are characterized by the location and access transparency of the secondary storage [12]. The access to secondary storage is done by the fast communication network. This property makes it possible to think of the secondary storage as one large virtual device, even if it consists of many small devices. The access times to secondary storage are independent of the processors location. Our parallel external sorting algorithm exploits this transparency to a large extent.

The properties of access and location transparency cannot be found at shared nothing multiprocessors. This architecture is characterized by the fact that each node has its dedicated secondary storage. Each processor can only access this directly attached secondary storage. Prominent representatives of shared disk multiprocessors in the database area are systems like Bubba [11], Gamma [15], Prisma [2] and Teradata [32].

The way in which our sorting algorithm exploits the access and location transparency of secondary storage distinguishes it from the algorithms in [4], [7], [16] and [26], which are based on shared nothing architectures. Our algorithm discards the shared

www.manaraa.com

nothing paradigm, as it is not open to high performance new disk technologies like RAIDs (cf. [28] and [19]). RAIDs offer the location and access transparency needed by our algorithm.

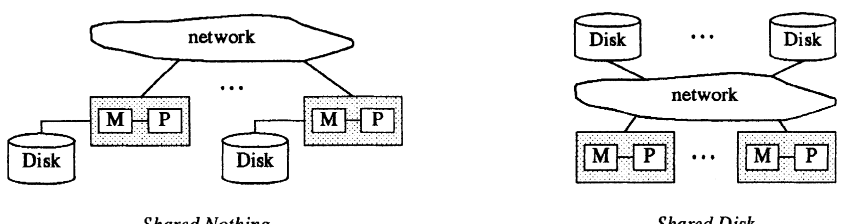


Fig. 3. Shared Disk vs. Shared Nothing

3.3 Optimal Parallel Run Time

In the sequential case, the run time of an external sorting algorithm has a lower bound which is determined by the transfer rate to secondary storage. Even in the parallel case, the run time of sorting is at least as high as the (parallel) transfer time needed to read and write all input tuples twice (once for the presort phase and once for the merge phase). An optimal parallel external sorting algorithm should have a run time near to that lower bound.

The parallel external sorting algorithm should exploit the I/O and CPU power of multiprocessors. But neither a significant I/O nor a significant CPU bottleneck can be accepted. In the following we assume that the transfer rate to secondary storage is a fixed system parameter whereas the degree of CPU parallelism can be chosen freely within the capability of the given system. This reflects the target architecture chosen for our research. In general, the I/O-subsystem of a shared disk multiprocessor has such a high performance that CPU parallelism is inevitable to yield a system without bottleneck. The CPU parallelism should be increased until a run time balance between CPU and I/O is obtained. This degree of CPU parallelism is called *balanced parallelism* (d in figure 4). The problem is now how to design a parallel external sorting algorithm which is effectively scalable. The term *effective scalability* expresses the property of a parallel external sorting algorithm having linear speedup characteristics up to the point of balanced parallelism. Figure 4 illustrates the above discussion.

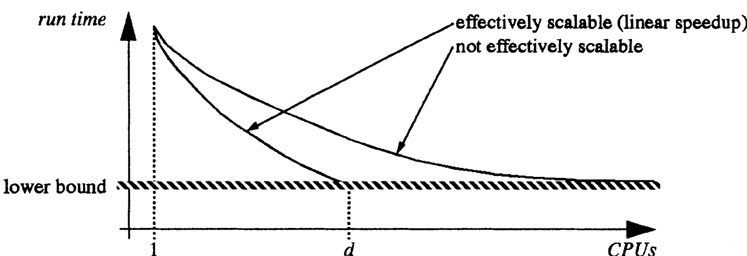


Fig. 4. Run Time of a Parallel External Sorting Algorithm

www.manaraa.com

3.4 Parallelization of External Sorting

In this paper we do not apply the paradigm of vertical parallelism to parallelize the external sorting algorithm. The reason is that in a distributed memory environment the costs of interprocessor communication have a high threshold value. In order to minimize the amount of communication it is necessary to build packets of tuples which are sent from one subtask of the pipeline to the next one, instead of sending the tuples alone. Each subtask has to unwrap the incoming packets in order to extract the tuples to be processed. Furthermore after processing the tuples, they have to be wrapped up by each subtask for the outgoing packets which are sent to the next subtask. The handling of the communication packets is very CPU intensive and cannot be parallelized, because each subtask has to deal with all tuples and their packaging. Measurements have shown that in case of external sorting the processing of the communication packets takes up 80% of total CPU time. The ratio becomes even worse if the degree of parallelism increases, as the processing time for the communication packets in each subtask is invariant with the degree of vertical parallelism.

Therefore we apply the paradigm of horizontal parallelism. The parallel processes can do their jobs without communication between each other, because their jobs are independent from each other. Thus the performance of the parallel algorithm does not depend on interprocessor communication costs in such a crucial way as it is the case with vertical parallelism.
However, a similar packaging mechanism as described in the previous paragraph must be implemented in order to access the tuples stored on secondary storage. In contrast to vertical parallelism this kind of packaging can be easily parallelized, because the processes of the horizontally parallelized external sorting algorithm just work on a data partition instead of on all tuples.

Because the presort phase and the merge phase of the external sorting algorithm cannot be overlapped[1)], the two phases of the parallel version are explained in distinct subsections. The paradigm of horizontal parallelism is applied to transform the sequential algorithm (cf. section 3.1) to a parallel one.

3.4.1 Presort Phase. The presort phase is performed by p processors P_j $(1 \le j \le p)$ in parallel. In accordance with figure 1, we describe the three phases (splitting, parallel processing and integration phase) of the horizontally parallelized presort phase in figure 5.

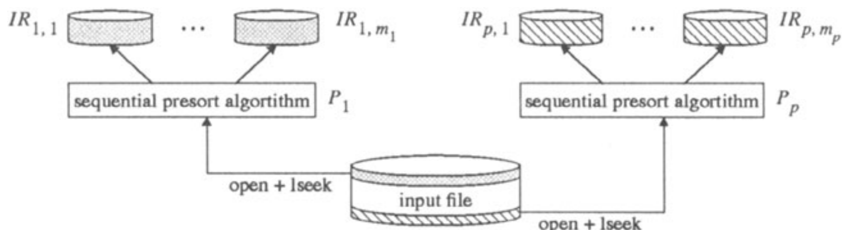


Fig. 5. Horizontally Parallelized Presort Phase

1) Recall: In order to merge the initial runs during the merge phase, the presort phase has to be completed.

www.manaraa.com

The input data must be split into p equally sized data partitions. The input file is stored on secondary storage having the property of access and location transparency, therefore each processor P_j can access the input file directly. The splitting of the input data can be done logically by simple commands of the operating system. The start and end positions of the data partitions are computed by simple arithmetic. It is not necessary to move the input data physically during the splitting phase. Furthermore this simple splitting phase guarantees that all data partitions have equal size.

Now the p data partitions of the input file can be processed in parallel and independently. Each processor P_j applies the usual sequential algorithm to construct its initial runs $IR_{j,i}$ $(1 \le i \le m_j)$. The initial runs are stored on secondary storage.

The final integration phase can be omitted, since the intermediate results are already in a form (set of initial runs) which can be processed by the following merge phase of the external sorting algorithm. The omission of the integration phase is due to the access and location transparency of the secondary storage.

Taking a closer look on the theoretical run time of the parallelized presort phase, we recognize a linear speedup. The parallel presort phase does not bear any additional costs because of the splitting and integration phases. The linear complexity of the presort phase deduced in section 3.1 together with the property that the data partitions are equally sized guarantees this linear speedup.

3.4.2 Merge Phase. The merge phase of the external sorting algorithm is performed by p processors P_j $(1 \le j \le p)$ in parallel. The m initial runs IR_i $(1 \le i \le m)$ form the input. In order to parallelize the merge phase horizontally, the initial runs IR_i must be split into p independent data partitions D_j. Two methods are possible to accomplish the data partitioning:

SETPART. This method is very simple and straightforward. It divides the set of the m initial runs IR_i into p disjunct subsets D_j. Each subset D_j contains m/p initial runs (cf. figure 6, left part).

KEYPART. This method is explained by figure 6 (right part), too. It splits each initial run IR_i into p regions $R_{i,j}$ $(1 \le i \le m, 1 \le j \le p)$. The borders of the regions are determined by the value of $(p+1)$ splitting keys k_j. They must obey the following condition:

$$k_0 \le k_1 \le k_2 \le \ldots \le k_p \qquad (2)$$

The value v of the sorting attribute of each tuple residing in region $R_{i,j}$ lies between k_{j-1} and k_j. Each region $R_{i,j}$ itself is a small initial run. Now define a data partition D_j to be the set of m small initial runs:

$$\forall (1 \le j \le p) : D_j = \{R_{1,j}, \ldots, R_{m,j}\} \qquad (3)$$

The splitting phase does not move any data neither by the *SETPART* nor by the *KEYPART* method. Because of the access and location transparency of the secondary storage the computation of the p data partitions D_j can be performed logically. Even the splitting of the initial runs into smaller ones can be done logically.

After having determined the independent sets D_j of initial runs, each of the p parallel merge processes P_j merges the partition D_j assigned to it. The result of each merge process P_j is the sorted file O_j (cf. again figure 6).

www.manaraa.com

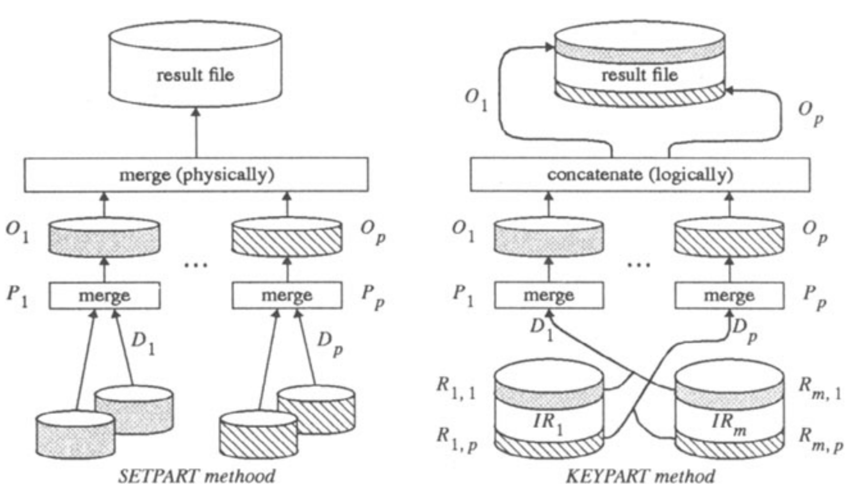


Fig. 6. Horizontally Parallelized Merge Phase

The final integration phase of the horizontally parallelized merge phase depends highly on the choice of the splitting method. If the p data partitions D_j are computed by the ***SETPART*** method, the integration of the p intermediate results O_j consists in merging them (cf. figure 6, left part). Although this merging can be temporally overlapped with the construction of the intermediate results O_j, it has nevertheless an unwanted effect of sequentialization. One single process has to merge all the files O_j. Because the processing time of merging depends linearly on the number of tuples to be merged, the total processing time of the parallelized merge phase is the same as in the sequential case. The result is, that no speedup at all is obtained. Therefore, the *SETPART* method should not be applied, although it is simple and straightforward.

If the p data partitions D_j are computed by the *KEYPART* method, the integration of the p intermediate results O_j consists in concatenating them (cf. figure 6, right part). This works, because equations (2) and (3) guarantee that the result of the concatenation is sorted. The concatenation can be done logically due to the access and location transparency of the secondary storage. There is no need at all to move the files O_j physically. Thus the processing time of the integration phase can be neglected. This is the reason why our new parallel external sorting algorithm uses a *KEYPART* method in order to split the m initial runs IR_i into the p data partitions D_j.

Before presenting the new splitting algorithm, which implements a *KEYPART* method, we will take a closer look at the run time of the horizontally parallelized merge phase. This is an interim calculation, which excludes the run time of the new splitting algorithm. Under the condition that the p data partitions D_j are equally sized (i.e. contain the same number of tuples), the speedup is linear like the speedup of the presort phase of the external sorting algorithm. This statement can easily be deduced from the observations made in section 3.1. However one crucial precondition of the linear speedup property is that the data partitions are equally sized and can be computed efficiently. This exactly is guaranteed by our data partitioning method even in the presence of arbitrary data skew.

www.manaraa.com

4 Data Partitioning

Data partitioning by means of splitting keys is one precondition that a horizontally parallelized external sorting algorithm will work in a reasonable manner. Therefore this section explores the possibilities of data partitioning using splitting keys. The discussion of data partitioning of this section is not done without reference to the parallel external sorting algorithm. Before describing and analyzing the new partitioning algorithm, first general requirements are postulated and related issues are discussed.

4.1 Requirements and Existing Approaches

A data partitioning algorithm should be efficient and robust against data skew in order to be useful for a *KEYPART* method. A trade-off exists between these two requirements. On the one hand, one can use a simple sampling algorithm to deduce the splitting keys. This sampling method is easy to implement and has good performance. But the quality of the splitting keys tends to become unacceptable in presence of data skew. The splitting keys may yield data partitions which differ a lot in their sizes. Thus one principle precondition is not fulfilled. Explorations by Quinn [29] confirm this statement.

On the other hand, one can be robust against data skew by paying an enormous computational overhead. An exact partitioning algorithm (like the percentile finding algorithm [21]) is based upon searching algorithms by bisection. The parallel external sorting algorithm proposed in [34] runs the percentile finding algorithm to compute independent data partitions needed for the parallel merge phase. But the I/O complexity, which depends linearly on the number of initial runs, and the random accesses to secondary storage show that the percentile finding algorithm is unsuited to implement a *KEYPART* method efficiently [27].

These observations are noticed by DeWitt et al. in [16], too. A compromise solution must be found in order to keep the trade-off between efficiency and robustness as low as possible. It is clear that the sizes of the data partitions do not need to be totally equal. A certain degree of imbalance can be tolerated. Therefore an exact splitting algorithm (like the percentile finding algorithm [21]) is superfluous. The compromise solution consists in an intelligent sampling method. The probabilistic splitting algorithm used in [16] samples the input file to be sorted. The maximum imbalance between the single data partitions can be estimated by means of probability theory. However, the drawback of the probabilistic splitting algorithm is that it never can guarantee the maximum imbalance, because a factor of uncertainty will always remain. To overcome this disadvantage, a new adaptive partitioning algorithm by sampling is developed in the next section.

4.2 Adaptive Partitioning by Sampling

The new adaptive partitioning algorithm of this section pays attention to the data distribution of the input file by sampling it during the presort phase. The distance between the samples determines the accuracy of the splitting keys (i.e. the maximum imbalance of the data partitions due to the splitting keys can be predicted). The possibility to vary the distance between the samples derived from the input file makes the algorithm adap-

www.manaraa.com

tive. Thus, every given limit of maximum imbalance can be reached. The following parallel merge phase does not suffer from processing skew in presence of data skew.

The main difference to the probabilistic splitting algorithm [16] is, that the adaptive partitioning algorithm can predict the quality of the splitting keys exactly instead of estimating it. Nevertheless the adaptive partitioning algorithm remains a sampling method and avoids the performance drawback of the percentile finding algorithm [21]. Furthermore, the adaptive partitioning algorithm exploits the access and location transparency of secondary storage and therefore avoids interprocessor communication during the redistribution of input tuples unlike the probabilistic splitting algorithm [16].

4.2.1 Description. The essence of the adaptive partitioning algorithm can be described in a few words: Each nth tuple of each initial run forms a sample. They mirror the data distribution of the input file. After having sorted these samples, the adaptive partitioning algorithm computes the splitting keys. They represent the data distribution as well. Therefore imbalance between the sizes of the data partitions determined by these splitting keys is limited.

In order to give a formal description of the adaptive partitioning algorithm, we need the following definitions:

Def. 1. The ***step distance*** n is a positive number. It is a parameter of the adaptive partitioning algorithm and determines its accuracy.

Def. 2. A ***sample area*** is a connected region within a file of tuples. Its size (in number of tuples) is equal to the step distance n. A sample area is a logical partition which is independent of the physical blocks of an I/O-device.

The adaptive partitioning algorithm consists of 6 basic steps described in the sequel. Assume, that the m initial runs IR_i $(1 \le i \le m)$ are given. The adaptive partitioning algorithm now computes p data partitions D_j $(1 \le j \le p)$ implementing a *KEYPART* method (cf. section 3.4.2).

(i) Each initial run IR_i $(1 \le i \le m)$ is divided into l_i sample areas $R_{i,l}$ $(1 \le l \le l_i)$ (i.e. $IR_i = R_{i,1} \ldots R_{i,l_i}$). The following equations hold:

$$\forall (1 \le i \le m) : l_i = \lceil |IR_i| / n \rceil \tag{4}$$

$$\forall (1 \le i \le m)\, \forall (1 \le l \le l_i - 1) : |R_{i,l}| = n \tag{5}$$

$$\forall (1 \le i \le m) : |R_{i,l_i}| = |IR_i| - n \cdot (l_i - 1) \tag{6}$$

Equation (4) determines the number of sample areas each initial run is divided into. Equation (5) says that all sample areas except the last one of each initial run consist of as many tuples as prescribed by the step distance n. The last sample area of each initial run possibly contains fewer tuples as indicated by equation (6).

(ii) Each sample area $R_{i,l}$ $(1 \le i \le m, 1 \le l \le l_i)$ obtains an area key $A_{i,l}$. The area key $A_{i,l}$ is the value of the sorting attribute of the first tuple of $R_{i,l}$. Now define $C(IR_i)$ to be the sequence of area keys of initial run IR_i.

$$\forall (1 \le i \le m) : C(IR_i) = (A_{i,1}, \ldots, A_{i,l_i}) \tag{7}$$

Each $C(IR_i)$ is sorted, because the initial runs are sorted. Furthermore, each $C(IR_i)$ mirrors the data distribution of IR_i. This observation is important, because it makes the computation of the maximum load imbalance during the

www.manaraa.com

merge phase possible. The situation after the first two basic steps of the adaptive partitioning algorithm is shown in figure 7. Initial run IR_i and $C(IR_i)$ are shown.

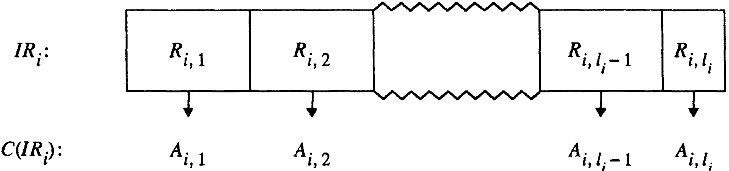


Fig. 7. Adaptive Partitioning Algorithm after the 2nd Step

(iii) The m sorted sequences of area keys $C(IR_i)$ are merged into the sorted sequence $C = (C_1, \ldots, C_L)$. It is $L = \sum_{i=1}^{m} l_i$. The merging can be realized by an usual m-way merging algorithm.

(iv) Now the sorted sequence C of area keys is divided into p equally sized partitions, thus yielding $(p-1)$ splitting keys k_j $(1 \le j \le p-1)$. The elements k_j are the splitting keys of the *KEYPART* method (cf. section 3.4.2), which divide the m initial runs IR_i into the p independent data partitions D_j. The value of k_j is defined as follows:

$$\forall (1 \le j \le p-1) : k_j = C_{j \cdot \lceil L/p \rceil} \quad (8)$$

The values k_0 and k_p can be defined arbitrarily, if they fulfill the following restriction:

$$k_0 < \min \{A_{i,1} | 1 \le i \le m\} \wedge k_p \ge \max \{A_{i,l_i} | 1 \le i \le m\} \quad (9)$$

(v) The splitting keys k_j are used to put the sample areas $R_{i,l}$ into the p groups G_j. The group G_j is defined as:

$$\forall (1 \le j \le p) : G_j = \{R_{i,l} | (1 \le i \le m) \wedge (1 \le l \le l_i) \wedge (k_{j-1} < A_{i,l} \le k_j)\} \quad (10)$$

Equation (10) says, that a group G_j contains exactly these sample areas $R_{i,l}$, whose area key lies in the interval bordered by the splitting keys k_{j-1} and k_j. Each group execpt G_p contains $\lceil L/p \rceil$ sample areas. The last group contains $L - ((p-1) \cdot \lceil L/p \rceil)$ sample areas.

Figure 8 shows the situation after the fifth step of the adaptive partitioning algorithm. The groups G_j approximate the searched data partitions D_j. The groups G_j are "nearly" disjunct. Only the borders of the groups G_j overlap. The last step has to compute the exact splitting positions within the initial runs.

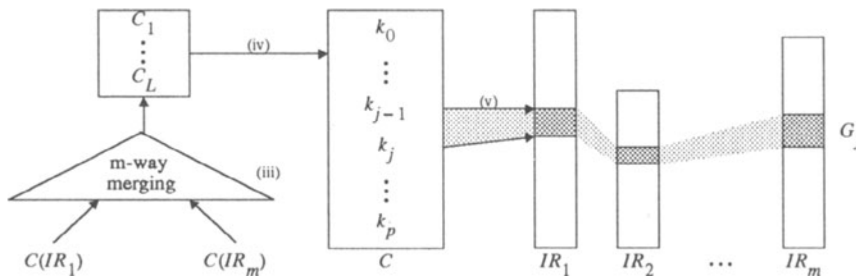


Fig. 8. Adaptive Partitioning Algorithm after the 5th Step

المنارة للاستشارات
www.manaraa.com

(vi) In order to compute the disjunct data partitions D_j from the groups G_j $(1 \le j \le p)$, the following observations are exploited:

· All tuples of the sample area $R_{i,l}$ have a sorting attribute, the value of which is not less than the area key $A_{i,l}$ belonging to $R_{i,l}$. This property can be deduced from the fact that initial runs are sorted.

· All tuples of the sample area $R_{i,l}$ have a sorting attribute, the value of which is not greater than the area key $A_{i,l+1}$ belonging to the sample area next to $R_{i,l}$. Again, this is true, because initial runs are sorted.

· The area keys $A_{i,l}$ of the sample areas $R_{i,l} \in G_j$ lie between the splitting keys determining the group G_j. More formally:

$$\forall (1 \le i \le m, 1 \le l \le l_i, 1 \le j \le p) : (R_{i,l} \in G_j \Rightarrow k_{j-1} \le A_{i,l} \le k_j) \tag{11}$$

Equation (11) limits the searching space for the exact splitting positions within the initial runs to compute the p searched data partitions D_j. The exact splitting positions are contained in the sampling areas bordering the groups G_j (cf. figure 9).

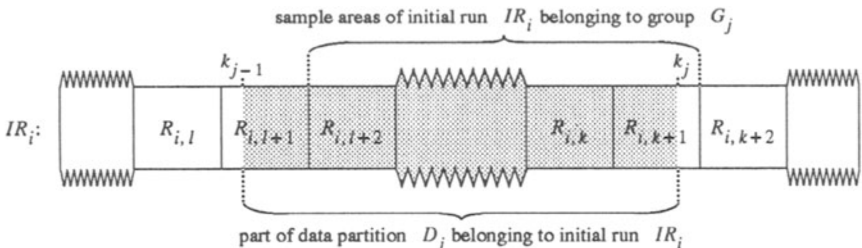


Fig. 9. Adaptive Partitioning Algorithm after the last Step

4.2.2 The Impact of Data Skew. Data skew might produce data partitions which have unequal size. We define the *imbalance* between two data partitions as the difference of the number of tuples in two data partitions:

$$Imbal(D_i, D_j) = \left| |D_i| - |D_j| \right| \tag{12}$$

$$MaxImbal = \max \{ Imbal(D_i, D_j) | \; 1 \le i, j \le p \} \tag{13}$$

A direct consequence of the description how to deduce the data partitions D_j from the groups G_j within step (vi) is the central proposition of this paper. It determines the maximum imbalance between the data partitions D_j.

Prop. 1. Assume that the p data partitions D_j are computed by our new adaptive partitioning algorithm from m initial runs using step distance n. Then the maximum imbalance $MaxImbal$ between two data partitions is limited by the following formula:

$$MaxImbal < 2 \cdot m \cdot n \tag{14}$$

It is an important observation that the maximum imbalance does neither depend on the degree of parallelism nor on the data skew. Using equation (1), one can deduce:

$$MaxImbal < 2 \cdot (N/h) \cdot n \tag{15}$$

Equation (15) is equivalent to:

$$MaxImbal/N < 2 \cdot n/h \tag{16}$$

The ratio $MaxImbal/N$ is called *normalized maximum imbalance*. It expresses the

www.manaraa.com

maximum imbalance which might be obtained by the adaptive partitioning algorithm, related to the total size of the input file. This is necessary to give a worst case estimation of the parallel merge phase. In contrast to the probabilistic splitting algorithm [16], we are able to guarantee an upper limit of the run time needed to sort a given input file in parallel. Using the expected length of $2 \cdot h$ for initial runs [23], the above worst case imbalance can be guaranteed, but the expected normalized maximum imbalance will be less than n/h.

The following example shows that just a small fraction of the input file must be sampled to guarantee a low degree of normalized maximum imbalance. We assume that the input file contains $2 \cdot 10^6$ tuples (each having a size of 50 bytes yields a total size of the input file of 100 MByte). Furthermore, we assume that the main memory is limited to 2 MByte thus having a heap size of $4 \cdot 10^4$ tuples during the presort phase. Then table 1 says that only 0.25% of the input data must be sampled in order to limit the normalized maximum imbalance to 2 %.

Normalized Maximum Imbalance	0.5%	1.0%	2.0%	5.0%
Percentage of Sampled Input Tuples	1.0%	0.5%	0.25%	0.1%
Step Distance n (in tuples)	100	200	400	1000

Table 1. The Accuracy of the Adaptive Partitioning Algorithm

5 Implementation and Performance

In this section we describe the implementation of the horizontally parallelized external sorting algorithm as developed in section 3.4. The new adaptive partitioning algorithm of section 4.2 is used to implement the *KEYPART* method. This section concludes with performance results done by measurements and theoretical analysis.

5.1 Hardware and Software Environment

The new parallel external sorting algorithm has been implemented on top of an iPSC/2 (intel Super Personal Computer) [20]. The iPSC/2 is a distributed memory multiprocessor. Its nodes are connected in a hypercube manner. The CFS (Concurrent File System) provides access and location transparency of the secondary storage. Thus, the iPSC/2 in combination with the CFS belongs to the class of shared disk multiprocessors (cf. section 3.2). The configuration we used for the implementation and the performance measurements have had 16 processing nodes and two I/O processors equipped with 2 disks each.

The operating system used for the implementation is *MMK* [8]. It offers the programmer three different types of objects in order to design and develop his program. All objects have global names, which are unique within the system. A parallel *MMK*-program consists of a set of these objects which are mapped onto real processors of the iPSC/2 during run time.

TASK. A task performs a sequential program written in any usual sequential programming language like C or FORTRAN. The tasks are the active components of a *MMK*-program. They perform the parallel algorithm in its literal sense.

MAILBOX. Mailboxes are used for the intertask communication. A task can send a message to any mailbox whose name is known to it. The messages are stored in the

www.manaraa.com

mailbox in a FIFO manner. Also, a task can receive a message from any known mailbox. The storing capacity of a mailbox is a start-up parameter which can be varied so that both synchronous and asynchronous intertask communication is possible.

SEMAPHORE. Semaphores deal with task synchronization.

5.2 Implementation

The set of tasks and mailboxes (semaphores are not used) of the parallel *MMK*-program implementing the new parallel sorting algorithm is shown in figure 10. The data flow in this figure is bottom up following the arcs.

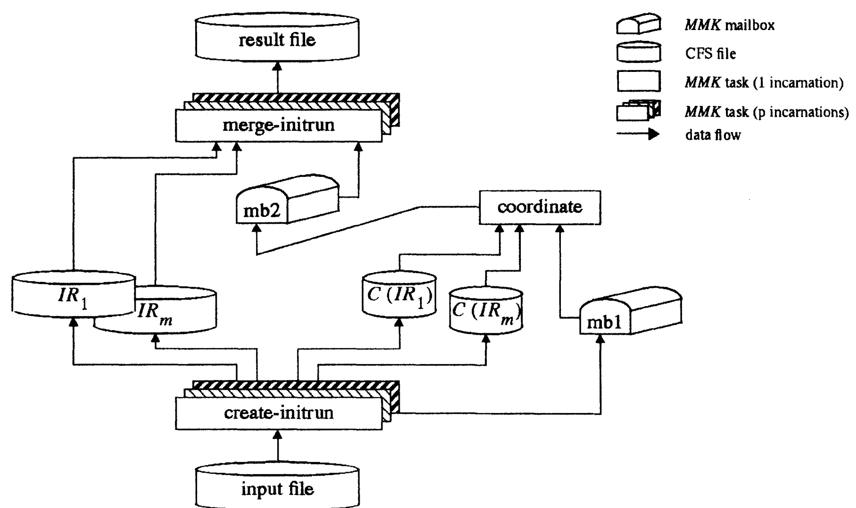


Fig. 10. The Parallel Sorting Algorithm on Top of *MMK*

The function of the task *create-initrun* is to generate initial runs from an unsorted input file which resides on the CFS (i.e. performing the presort phase). Furthermore it has to perform the basic steps (i) and (ii) of the adaptive partitioning algorithm. Therefore, its result is a set of initial runs IR_i and a set of area key sequences $C(IR_i)$ both stored in files of the CFS. After the completion of its job, the task *create-initrun* sends a message to the communication mailbox *mb1*. This message contains the filenames of the generated files. To process the task *create-initrun* in parallel, the paradigm of horizontal parallelism is applied and p incarnations of the task *create-initrun* are performed on p different nodes of the iPSC/2. The input file must be split logically as explained in section 3.4.1.

The function of the task *coordinate* is to perform the basic steps (iii) through (v) of the adaptive partitioning algorithm. It reads the files $C(IR_i)$. Their filenames are received from the mailbox *mb1*. The result of the task *coordinate* are the splitting keys k_j and the borders of the groups G_j. The result is sent to the mailbox *mb2*. This task cannot be performed in parallel, since it is the sequential part of the adaptive partitioning algorithm. Therefore, just one incarnation of it exists.

The function of the task *merge-initrun* is to compute the exact splitting positions within the initial runs IR_i according to the splitting keys k_j (i.e. performing the basic

www.manaraa.com

step (vi) of the adaptive partitioning algorithm) and to merge the initial runs (i.e. performing the merge phase). The task ***merge-initrun*** gets its start-up parameters (i.e. filenames, splitting keys, etc.) by reading the mailbox *mb2*. To perform the task ***merge-initrun*** in parallel, ***p*** incarnations of the task ***merge-initrun*** are processed on ***p*** different nodes of the iPSC/2 (i.e. horizontal parallelism). Attention must be paid that the number of splitting keys and the degree of parallelism during the merge phase are equal.

5.3 Performance

In order to demonstrate performance results, the following testbed has been chosen: The unsorted input file has a size of 100 MByte. Each tuple within the input file has a size of 50 bytes. The heap of the tasks ***create-initrun*** can hold 10^4 tuples each to generate the initial runs. The step distance ***n*** during the adaptive partitioning algorithm is set to 400. Comparing this parameter setting with table 1, a normalized maximum load imbalance of 2% can be guaranteed.

Figure 11 shows the run time during the presort and merge phases. It demonstrates linear speedup until the I/O-subsystem (i.e. the CFS of the iPSC/2) becomes the bottleneck. Referring to the discussion of section 3.3, our new parallel sorting algorithm is effectively scalable. Because the access pattern to the secondary storage is different during the presort and merge phases, the two phases have different values of balance parallelism. During the presort phase the access to the secondary storage is sequential, whereas it is random during the merge phase. The degree of CPU parallelism which can be exploited effectively seems to be very low. The reason is the performance imbalance between the CPU and the I/O-subsystem of the iPSC/2 which we used for our measurements. During the ***create-initrun*** phase 100 MByte are read from an written to the disk, i.e. a total of 200 MByte. This takes a total time of about 250 sec at the point of balance parallelism. Thus the performance of the I/O-subsystem of the iPSC/2 is about 800 KByte/sec. During the ***merge-initrun*** phase disk performance is about 500 KByte/sec.

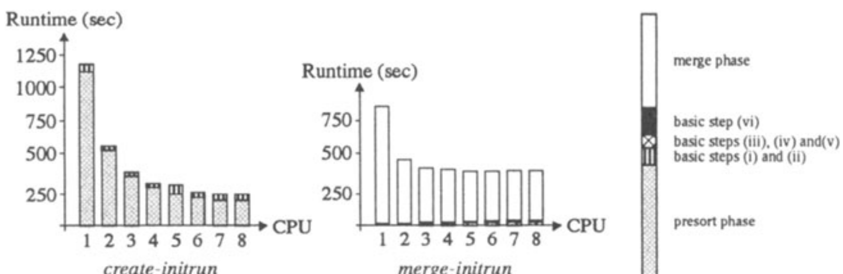


Fig. 11. Run Time of the Parallel Sorting Algorithm

Figure 11 expresses the additional run time due to the adaptive partitioning algorithm, too. It can be seen that the additional run time is very low w.r.t. the run time for sorting in its literal sense. In order to confirm this observation we give a theoretical analysis of the complexity of the adaptive partitioning algorithm.

The basic steps (i) and (ii) are performed during the presort phase by the task

www.manaraa.com

create-initrun. The computation of the files IR_i and $C(IR_i)$ is done simultaneously. Furthermore, the paradigm of horizontal parallelization is applicable. The additional CPU costs to compute the area keys $A_{i,j}$ can be neglected, because the sorting attributes of the sampled tuples have to be extracted anyway during the presort phase. The I/O complexity of the basic steps (i) and (ii) are $O((N/n) \cdot \kappa)$ where κ is the ration $KeySize/TupleSize$.

The basic steps (iii), (iv) and (v) are performed by the task ***coordinate***. This is the only sequential part of the adaptive partitioning algorithm. Therefore the additional costs must be explored very carefully. The additional I/O costs have again a complexity of $O((N/n) \cdot \kappa)$ like during the steps (i) and (ii). The additional CPU costs in order to merge the m sequences $C(IR_i)$ into the sequence $C = (C_1, \ldots, C_L)$ have a complexity of $O((N/n) \cdot \log(m))$. This may look expensive, but the measurements have shown that the sequential part of the adaptive partitioning algorithm can be performed quickly compared to the total run time needed to sort large volumes of data.

The final basic step (vi) of the adaptive partitioning algorithm can be performed in parallel again. Each process P_j which has to merge the data partition D_j assigned to it gets a prologue. During this prologue the exact splitting positions within the groups G_j are computed. The additional CPU and I/O both have a complexity of $O(n)$. The costs are due to the additional sample area, which must be loaded from secondary storage. Within this additional sample area the exact splitting position must be found according to the splitting keys. Table 2 summarizes the theoretical results about the additional costs due to the adaptive partitioning algorithm.

basic step	add. I/O costs	add. CPU costs	parallel processing
(i), (ii)	$O((N/n) \cdot \kappa)$	negligeable	yes
(iii), (iv), (v)	$O((N/n) \cdot \kappa)$	$O((N/n) \cdot \log(m))$	no
(vi)	O(n)	O(n)	yes

Table 2. Additional Costs of the Adaptive Partitioning Algorithm

6 Summary and Future Work

We have proposed a new parallel sorting algorithm for large data volumes. It applies the paradigm of horizontal parallelism. The two phases are treated separately. The presort phase turns out to have splitting and integration phases without any significant overhead. Furthermore, load imbalance cannot occur during the parallel processing phases. The subsequent merge phase gets a prologue in the course of which a *KEY-PART* method computes splitting keys. They are used to partition the initial runs horizontally. Parallel merging becomes possible. Both phases show linear speedup characteristics until the I/O-subsystem becomes the bottleneck. By means of CPU parallelism, we are able to tune the multicomputer.

Our new adaptive partitioning algorithm implementing a *KEYPART* method fulfills the necessary requirement of efficiency (low overhead) and robustness against data skew (load balance). Although it performs sampling in order to guarantee efficiency, it can give a worst case estimation of the maximum imbalance which may occur. The basic idea, which distinguishes our algorithm from existing sampling algorithms, is the way how the samples are taken. Instead of sampling the unsorted input data stream, we

www.manaraa.com

sample the initial runs. Thus, we have samples mirroring the data distribution. Therefore the splitting keys extracted from the samples are not sensitive to data skew. The accuracy of the splitting keys depends on the step distance (input parameter), which adapts the worst imbalance to given requirements. An other important requirement is that the accuracy of our adaptive partitioning algorithm does not depend on the degree of parallelism.

In the future, the following topics should be analyzed: The description of our adaptive partitioning algorithm provides various possibilities for tuning. Our current implementation sends the area keys $A_{i,j}$ via the CFS to the task *coordinate*. But the amount of data due to the area keys[2)] does not justify to store the area keys on secondary storage. Efficient main memory data structures and the intertask communication facility of *MMK* can be used to accelerate the algorithm.

Up to now, the performance measurements are done on a system which hardly provides parallelism within the I/O-subsystem. A highly parallel I/O-subsystem with fast access times and high transfer rates should be used to prove the ability of our new parallel sorting algorithm under this environment, too.

The most important challenge for the future is the integration of our new parallel sorting algorithm into the parallel query evaluation plans of our relational database system *TransBase* [33]. Its sequential query evaluation plans treat sorting as a monolithic operator. It will be necessary to divide this monolith into a presort operator and a merge operator. Thus a different degree of parallelism can be assigned to the operators ensuring the optimal degree of parallelism. Furthermore, this integration gives the processors more load due to other operators (e.g. selection) of the parallel query evaluation plans. This increments the potential of more parallelism compared to the case that only sorting has to be performed by the processors.

Appendix (List of used parameters)

- N: number of tuples to be sorted
- p: degree of parallelism (= number of processors)
- m: number of initial runs
- h: heap size used during presort phase
- n: step distance of our partitioning algorithm
- κ: ratio: $KeySize/TupleSize$

References

1. A. Aho, J. Hopcroft, J. Ullman; *Data Structures and Algorithms*; Addison Wesley Publ. Comp. Inc., 1983
2. P. Apers, M. Kersten, H. Oerlemans; *PRISMA Database Machine: A Distributed Main Memory Approach*; In: Proceedings of the 1st International Conference on Extending Database Technology, Venice, Mar 88
3. K. Batcher; *Sorting Networks and their Applications*; In: Proceedings of the 1968 Spring Joint Computer Conference, Vol. 32, 1968, pp. 307 - 314
4. B. Baugstø, J. Greipsland; *Parallel Sorting Methods for Large Data Volumes on a Hypercube Database Computer*; In: Proceedings of the 6th International Workshop on

2) If we assume that a key of a tuple has a length of 10 bytes, then the set of area keys extracted from the initial runs consumes 50 KByte of memory in the example chosen in section 5.3.

www.manaraa.com

Database Machines, Deauxville, Jun 89, LNCS No. 368, pp. 128 - 141
5. R. Bayer, T. Härder; *Preplaning of Disk Merges*; Computing, Vol. 21, No. 1, pp. 1 - 16, 1978
6. R. Bayer, T. Härder; *A Performance Model for Preplaned Disk Sorting*; Computing, Vol. 21, No. 1, pp. 17 - 36, 1978
7. M. Beck, D. Bitton, K. Wilkinson; *Sorting Large Files on a Backend Multiprocessor*; IEEE Transactions on Computers, Vol. 37, No. 7, Jul 88
8. T. Bemmerl, T. Ludwig; *MMK - A Distributed Operation System Kernel with Integrated Loadbalancing*; In: Proceedings of the CONPAR 90 - VAPP IV, Zürich, Sept 90
9. T. Bemmerl, A. Bode, P. Braun, O. Hansen, T. Treml, R. Wismüller; *The Design and Implementation of TOPSYS*; Technical report, Technische Universität München, No. 342/16/91 A, Jul 91
10. G.Bilardi, A. Nicolau; *Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared-Memory Machines*; SIAM J. Comput., Vol. 18, No. 2, Apr 89, pp. 216 - 228
11. H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, P. Valduriez; *Prototyping Bubba, A Highly Parallel Database System*; IEEE Transactions an Knowledge and Data Engineering, Vol. 2, No. 1, Mar 90
12. U. Borghoff; *Catalogue of Distributed File/Operating Systems*; Springer Verlag, Berlin Heidelberg, 1992
13. T. Chen, V. Lum, C. Tung; *The Rebound Sorter: An Efficient Sort Engine for Large Files*; In: Proceedings of the 4th International Conference on Very Large Data Bases, West Berlin, pp. 312 - 318
14. R. Cole; *Parallel Merge Sort*; SIAM J. Comput., Vol. 17, No. 4, Aug 88, pp. 770 - 785
15. D. DeWitt, S. Ghandeharizadeh, D. Scheider, A. Bricker, H. Hsiao, R. R. Rasmussen; *The Gamma Database Machine Project*; IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, Mar 90
16. D. DeWitt, J. Naughton, D. Schneider; *Parallel Sorting on a Shared Nothing Architecture using Probabilistic Splitting*; In: Proceedings of the 1st International Conference on Parallel and Distributed Information Systems, Miami Beach, Florida, Dec 91
17. Y. Dohi, A. Suzuki, N. Matsui; *Hardware Sorter and its Application to Database Machine*; In: Proceedings of the 9th Conference on Computer Architecture, Austin, Apr 82, pp. 218 - 225
18. B. Edem, R. Helliwell, T. Johnston, E. Lary, R. Lary; *Sort Accelerator*; Technical Report, Database Research Group, DEC, May 90, Do. No.: DBS-TR-3 DEC-TR-691
19. G. Gibson; *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*; Technical Report, No. UCB/CSB 91/613, Computer Science Decision, University of California, Berkeley, Dec 90
20. Intel Scientific Computers; *Concurrent Supercomputing - The Second Generation, A technical Summary of the iPSC/2 Concurrent Supercomputer*; Reprinted from the Proc. of the ACM Third Hypercube Conference
21. B. Iyer, G. Ricard, P. Varman; *Percentile Finding Algorithm for Multipe Sorted Runs*; In: Proceedings of the 15th International Conference on Very Large Databases, Amsterdam, 1989, pp. 135 - 144
22. B. Kandler, M. Pawlowski; *SAM: A Sorting Toolbox - User's Guide*; Technical Report, Technische Universität München, No. 342/2/91 B, Jun 91 (in german)
23. D. Knuth; *Sorting and Searching. The Art of Computer Programming*; Addison Wesley Publ. Comp. Inc., 1973, Vol. 3
24. K. Lehnert; *Regelbasierte Beschreibung von Optimierungsverfahren für relationale Datenbankabfragesprachen*; Ph.D. Thesis, Technische Universität München, Dec 88 (in german)
25. E. Loibl, H. Obermaier, M. Pawlowski; *Towards Parallelism in a Relational Database System*; Technical Report, Technische Universität München, No. 342/10/91 A, Jun 91
26. R. Lorie, H. Young; *A Low Communication Sort Algorithm For a Parallel Database Machine*; In: Proceedings of the 15th International Conference on Very Large Data Bases, Amsterdam, 1989, pp. 125 - 134

www.manaraa.com

27. D. Menzel; *Paralleles Externes Sortieren auf Multiprozessoranlagen*; Master Thesis, Technische Universität München, Nov 1991 (in german)
28. D. Patterson, G. Gibson, R. Katz; *A Case for Redundant Arrays of Inexpensive Disks (RAID)*; In. Proceedings of the SIGMOD International Conference on Managemant of Data, Editors: H. Boral, P. Larson, ACM Press, Chicago, Jun 88, pp. 109 - 116
29. M. Quinn; *Parallel Sorting Algorithms for Tightly Coupled Multiprocessors*; Parallel Computing, Vol. 6, 1988, pp. 349 - 357
30. A. Reuter; *Database Sharing*; Informatik Spektrum, Vol. 8, No. 4, Apr 85, pp. 225 - 226
31. M. Stonebraker; *The Case for Shared Nothing*; Database Engineering, Vol. 9, No. 1, 1986
32. Teradata Corp.; *DBC/1012 Database Computer System Manual*; Doc. No. C10-0001-02, Nov 1985
33. TransAction Software GmbH; *TransBase Relational Database System*; System Guide, München, 1988
34. P. Varman, B. Iyer, S. Scheufler; *A Multiprocessor Algorithm for Merging Multiple Sorted Lists*; In: Proceedings of the International Conference on Parallel Processing, 1990, pp. III-22 - III-26

www.manaraa.com

Quantum Mechanical Programs for Distributed Systems: Strategies and Results

H. Früchtl and P.Otto

Chair for Theoretical Chemistry
Friedrich-Alexander-University Erlangen-Nürnberg
Egerlandstr. 3
D-8520 Erlangen, Germany

Abstract. The knowledge about structure and functionality of chemical systems not only allows the verification of physical and chemical mechanisms but also gives the opportunity of proposing new materials with selected new properties. For investigations on molecular level quantum mechanical methods have to be used. Reliable calculations on molecules and macromolecules require the use of high performance computers. The development of massively parallel computer systems opens the possibility of doing calculations on larger and more realistic chemical structures.

1 Introduction

Analysis, modelling, prediction and optimization of molecular and macromolecular structures with computers is one of the "challenge classes" of scientific high performance computing.

The basis for understanding biological processes is the investigation of the mechanisms of biomolecules like DNA and proteins. Organic polymers are of increasing importance as materials for new technologies. Due to the variety of their physical and chemical properties there is still no end in sight for their application fields.

To make such investigations possible new efforts are necessary both at scaling, i.e. adjusting the problems to existing machines and at the development of new high performance computers for large scale calculations.

The application of quantum mechanical methods for investigations on molecular level can in principle serve two purposes:

- They can help to clear up mechanisms of physical and chemical phenomena or to check the correctness of theoretical interpretations of experimental results. Interesting and important effects like high-temperature superconductivity of ceramic materials or the fundamental steps of chemical carcinogenesis probably can only be explained by means of theoretical investigations on microscopic level.
- In the area of material sciences the knowledge of basic physical effects can be exploited to propose new substances with special selected properties using the theoretical methods based on quantum mechanics. Expensive and time-consuming experimental investigations can be avoided or at least reduced. Especially in the field of molecular biology the knowledge about the relation between biological effects and chemical structure in combination with quantum mechanical

www.manaraa.com

calculations can be used to design molecules with specific properties. This can be of great importance for the development of new enzymes and other pharmaceutical substances.

To obtain reliable results, these and many other problems have to be treated with ab initio methods. To carry out such calculations with sufficient accuracy on complex chemical systems large resources of computation time, memory and disk space are required, so that high performance computer systems have to be used.

In spite of the manifold different numerical algorithms involved in quantum mechanical calculations of the electronic structures of molecules and polymers a high degree of parallelism can be observed. This allows the realization of programs applicable to complex chemical systems on massively parallel computers.

A major part of our investigations deals with the calculation of band structures of organic polymers and biochemical macromolecules and the determination of their physical and chemical properties. Usually the computation consists of three time-consuming steps:

- The ab initio Hartree-Fock method [1-3] gives the energy band structure and electronic wave function of the quasi one-dimensional periodic system.
- Wave functions and energies are corrected with respect to electron correlation effects using perturbation theoretical methods [4].
- Finally the properties under consideration (mechanical, nonlinear optical, transport properties) are calculated from the correlated wave function [5].

As on one hand the different parts of the computation imply different requirements concerning computer architecture and on the other hand an exact knowledge of hardware and operating system is necessary to choose the best algorithm for a given problem, a close cooperation of development and application as it is realized in the MEMSY project is advantageous for both sides.

2 The Hartree-Fock Crystal Orbital (HF-CO) Method

Aim of the Hartree-Fock SCF (self-consistent field) approximation is the determination of the energy band structure and electronic wave function of periodic polymers, from which other properties (conducting properties, (hyper-)polarizabilities etc.) can be calculated. The SCF procedure consists in the iterative solution of the generalized hermitian eigenvalue problem

$$\boldsymbol{F}(k_i)\,\vec{c}_n(k_i) = \varepsilon_n(k_i)\,\boldsymbol{S}(k_i)\,\vec{c}_n(k_i)$$

$n = 1, \ldots, \mathrm{NBF}$, basis functions $\qquad i = 1, \ldots, \mathrm{NKP}$ points in reciprocal space

where the Fock matrix F is determined as a function of the eigenvectors $\vec{c}$ from the previous iteration. This is repeated until convergency is achieved.

www.manaraa.com

The Fock- and overlap-matrices, F(k) and S(k), are obtained as Fourier transforms of the corresponding matrices in direct space.

$$M(k) = \sum_{J=-NEIG}^{NEIG} e^{iR_J k} M^{0J} \qquad M = F, S$$

The matrices F^{0J} describe the interaction of the electrons in the reference cell 0 and those in cell J and S^{0J} occurs because of the non-orthogonality of the basis functions. NEIG is the number of cells interacting with the reference cell.

The first time-consuming step consists in the calculation of the so-called two-electron integrals, which are needed for the calculation of the Fock matrix. They describe the Coulomb and exchange interaction between two electrons and are of the general form:

$$\langle \chi_r^0 \chi_u^H | \frac{1}{r_{12}} | \chi_s^J \chi_v^L \rangle = \iint \chi_r^0(\vec{r}_1)\, \chi_u^H(\vec{r}_2)\, \frac{1}{r_{12}}\, \chi_s^J(\vec{r}_1)\, \chi_v^L(\vec{r}_2)\, d\vec{r}_1 d\vec{r}_2$$

r, s, u, v = 1,...,NBF
J, H, L = -NEIG,...,NEIG

The basis functions (atomic orbitals) are written as linear combinations of n_i so-called primitive Gaussian functions ($n_i \approx 2 - 10$). The superscripts denote one of Ngroup possible combinations of cells where the basis functions are localized. The value of Ngroup increases rapidly with increasing NEIG (NEIG = 1, 2, 3; Ngroup=5, 15, 35). The total number of integrals to be calculated then yields

$$Ngroup * NBF^4 * n^4$$

(without consideration of symmetries according to basis functions and negligible integrals smaller than an integral threshold).

In order to minimize the number of redundant arithmetic operations several quantities are calculated in advance and stored in tables for later use during the actual integral calculation. The different loops in the program were vectorized as far as this was possible [6,7]. The total number of arithmetic operations needed for the final calculation of an integral over primitive functions are:

- 22 additions
- 20 multiplications
- 2 divisions
- 2 square roots

For storing one integral two memory words of 8 bytes are needed (one for the value and one for the four indices).

www.manaraa.com

It is quite obvious that the calculation of the integrals has a high degree of parallelism, because every integral can be calculated independently of the others. The requirements for the communication system are low, the basic difficulty arises from the need for a balanced distribution of workload, because the number of integrals to be calculated in one group cannot be determined exactly in advance (neglect of integrals smaller than a threshold, symmetries according to basis functions).

3 Parallelization Strategy for the Integral Program

3.1 Program Structure

Fig. 3.1 shows the loop structure of the two-electron integral program.

igroup = 1 .. Ngroup

i = 1 .. NBF

j= 1 .. NBF

k= 1 .. NBF

l= 1 .. NBF

ic = 1 .. n_i

jc = 1 .. n_j

kc = 1 .. n_k

lc = 1 .. n_l

integral over primitive functions

Fig. 3.1 Loop structure of the two-electron integral program

In earlier attempts the distribution of tasks to the computing nodes was done in the loop over igroup [8], but a good load balancing could only be achieved for high values of Ngroup, although the work was dynamically distributed. In this investigation distribution over the loops i and j was used (NBF*NBF pairs) in order to obtain a sufficiently fine granularity. As the tasks are distributed using the farming concept (see Fig. 3.2) different task sizes have only small influence on the workload of the processors. A "master" processor calculates data needed for all integrals and divides the number of ij-pairs for every group in as many parts as there are "slave" proces-

www.manaraa.com

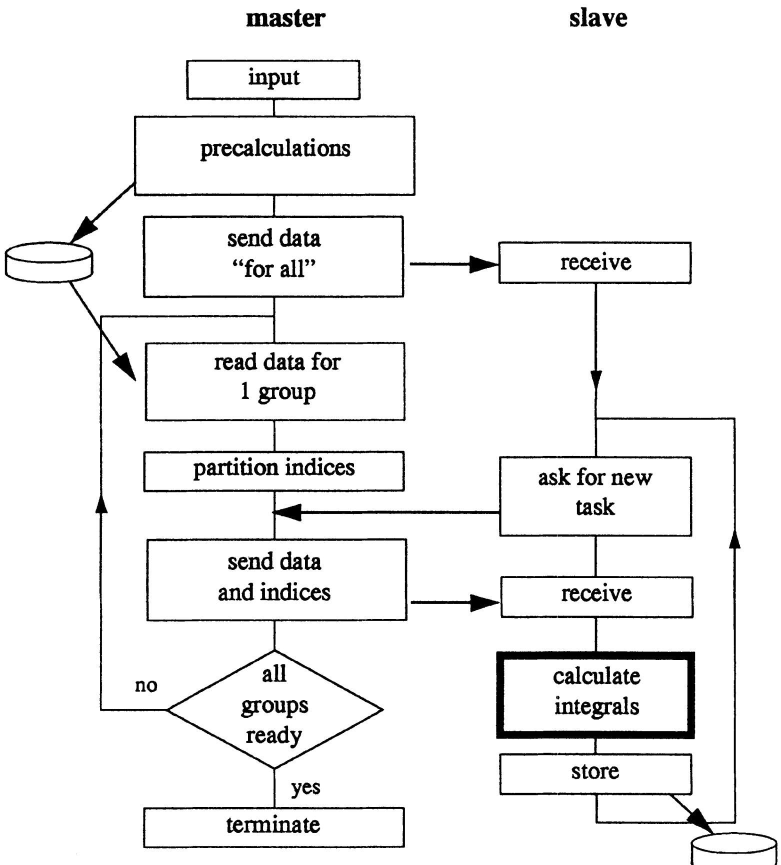


Fig. 3.2 Master-slave concept of the integral program

sors. Whenever a "slave" tells the "master" that he has nothing to do, the "master" sends him a new task, consisting of information about the range of ij-pairs to be computed and the precalculated data for the new group [9].

المنارة للاستشارات
www.manaraa.com

3.2 Implementation and Results

Measurements of speed-up and efficiency as well as analysis of communication overhead were done on two different parallel computers, both of them MIMD machines with distributed memory:

SUPRENUM	up to 256 nodes with vector units (up to 32 were used). 8 Mbytes memory and a peak performance of 20 MFlops per node.
Intel iPSC/860	32 nodes (hypercube architecture). 16 Mbytes memory, peak performance 60 MFlops per node.

In Fig. 3.3a and 3.3b the results for a model system (NBF=14, Ngroup=5) are shown in comparison. Although the actual calculation times are larger on the SUPRENUM than on the Intel by a factor of about 3.5 we obtain nearly identical curves for speed-up and efficiency. The maximum efficiency (in the case of the considered

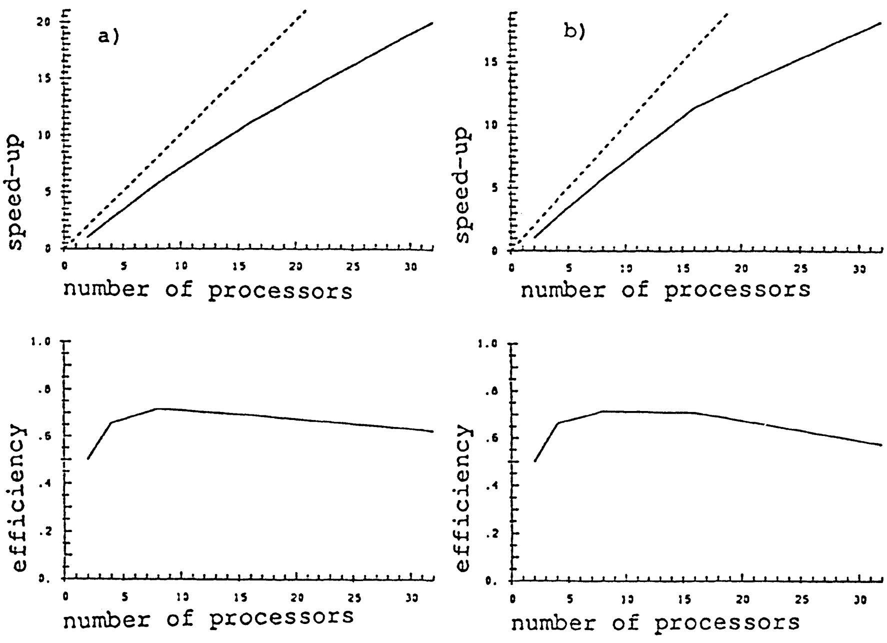


Fig. 3.3 Parallelization results on a) SUPRENUM and b) iPSC/860

polymer) is reached with ten processors. A further increase in the number of processors leads only to a slight decrease in efficiency under 0.8, i.e. 20% of the computation time is used for overhead arising from parallelization.

www.manaraa.com

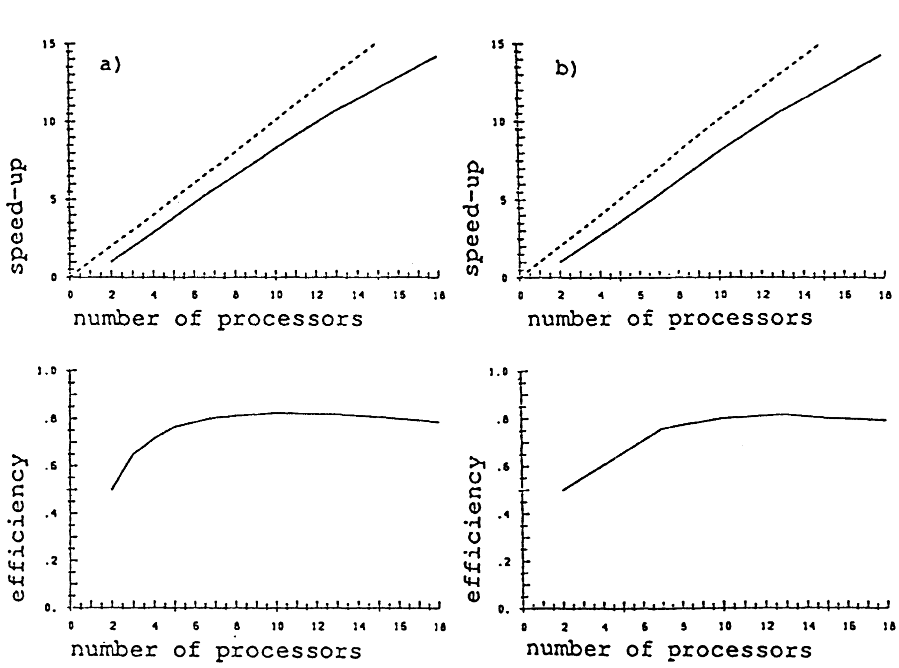


Fig. 3.4 Parallelization results for larger systems: a) polyethylene with two neighbors and b) polybutylene (double number of basis functions)

If the number of integral groups is increased further (Fig. 3.4a: Ngroup = 15) an efficiency above 0.8 can also be obtained for a larger number of processors. The same improvement can be observed, if the number of basis functions is increased, as Fig. 3.4b (NBF = 28) shows. This means that for calculations on systems of scientific interest, which are much larger than our model polymers, a sufficient speed-up can be achieved even with a much larger number of processors.

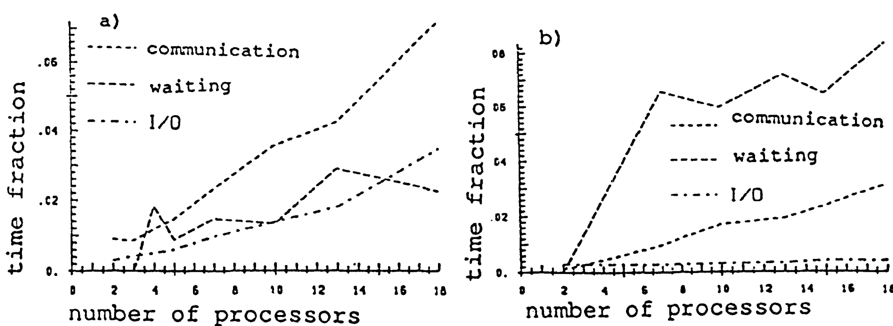


Fig. 3.5 Analysis of parallelization overhead for a) polyethylene with 2 neighboring cells and b) polybutylene with one neighboring cell

www.manaraa.com

An analysis of parallelization overhead (see Fig. 3.5a, 3.5b) shows, that in the case of many integral groups, i.e. a larger number of tasks to be distributed, communication dominates the overhead, whereas for a smaller number of groups and larger elementary cell waiting time is the major part, due to the coarser granularity of the distribution.

4 Outline of the Parallelization of the SCF Program

The generalized hermitian eigenvalue problem has to be solved iteratively, because the matrix elements are functions of the solution. In every iterative cycle NKP matrix diagonalizations have to be done.

A detailed analysis of the sequential algorithm shows, that the problem can be divided into two parts, for which different parallelization strategies on distributed computing systems have to be chosen. The sequence of the individual steps and their distribution to the single processor nodes is outlined in Fig. 4.1.

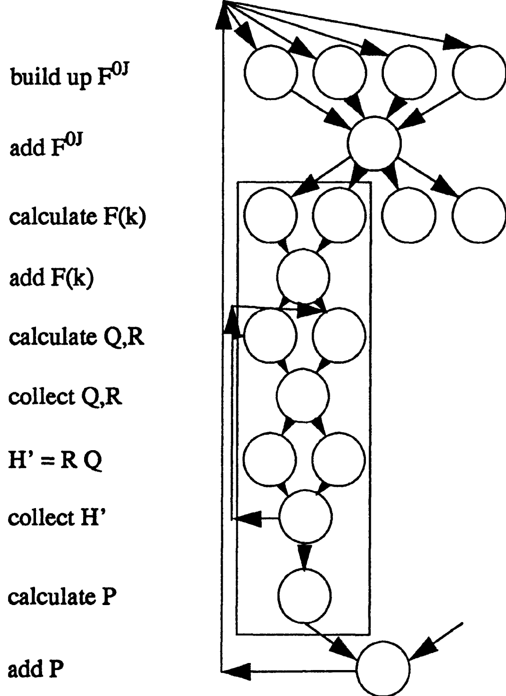


Fig. 4.1 Parallel structure of the SCF program

المنارة للاستشارات
www.manaraa.com

- The integrals calculated in the first step, multiplied with elements of the charge-density matrix **P** obtained from the eigenvector coefficients, are used to build up the matrices F^{0J} for all values of J. As each processor only accesses a part of the files where the integrals are stored (these are again distributed following the farming concept), in the memory of every processor there is only a partial sum of all matrix elements. These matrices are then sent to the "master" and added up there.

- The NKP diagonalizations per iteration cycle are distributed to the individual processors as follows. First the Nslav "slave" processors are grouped in NKP clusters with Nslav/NKP nodes each. In every cluster one processor is characterized as "cluster master" for coordination tasks in addition to the work of every other slave.

  - Within each cluster the columns of the matrices $\mathbf{F}^{0J}$ are distributed to the individual processors, where partial sums of the Fourier transforms $\mathbf{F}(k_i)$ for the particular values of k are built up. These are summed up by the "cluster master" afterwards. The amount of data to be sent is NBF^2*(NEIG+1) real numbers. The "cluster master" then does a Lowdin orthogonalization to transform the general to a special eigenvalue problem. This step, as well as the following tridiagonalization using a Householder or Block-Householder algorithm, can at least partially be done in parallel by all processors of the cluster.

  - The diagonalization of the tridiagonal matrix is now done using the QR algorithm. Fig. 4.2 illustrates how this method is realized within one cluster [10]. Two steps are necessary for this. First the matrices **Q** and **R** have to be calculated (factorization **H=QR**), then they have to be multiplied in opposite order (**H'=RQ**). The new elements of all columns of **Q** can be calculated without knowing the neighboring rows, so they can be calculated in parallel without any communication if **Q** is distributed by rows. The same holds for the rows of **R** in case of distribution by columns. Also the multiplication can be done in parallel, because for the calculation of elements of **H'** only the involved rows of **R** and columns of **Q** are needed. To avoid load balancing problems arising from the special structure of the matrices involved (**R** is a right triangular matrix, **Q** is of Hessenberg type) the distribution to the single processors happens according to the "Team Mapping" method, which means a distribution of columns (or rows) in alternatingly increasing and decreasing order.

  - Afterwards the new partial matrices are collected by the "cluster master" and distributed to the "slaves" for the next iteration.

www.manaraa.com

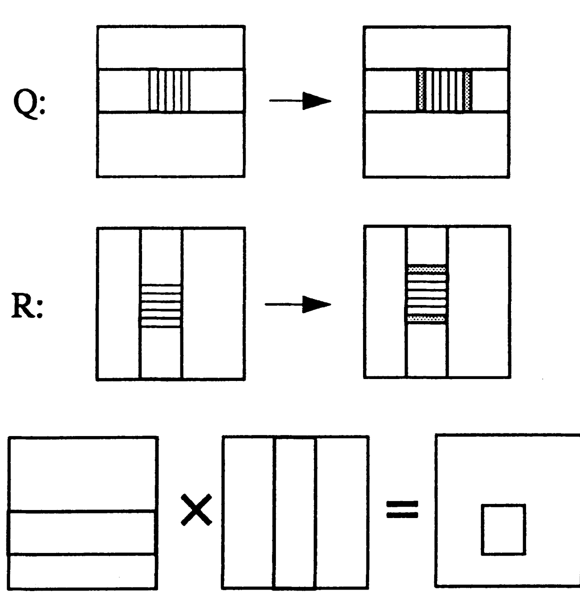


Fig. 4.2 Calculation of Q (distributed by rows) and R (distributed by columns) and matrix multiplication on one processor

- After the diagonalization is finished each "cluster master" calculates a part of the charge-density bond-order matrix **P** from the eigenvectors for his k-value. Then these matrices are sent to the global master, summed up there and sent back to the "slaves", because they are needed for building up the matrices F^{0J} for the next iteration cycle. The calculation terminates when the "master" recognizes that the solutions have converged.

In contrast to the calculation of the integrals in the SCF-calculation there is a much larger amount of communications and synchronizations. Many of the communications are of the form "one-to-all" (broadcast) or "all-to-one". The implementation of the single steps depends very much on computer architecture and communication system. Especially the MEMSY configuration with local communication memory can be used very efficiently for such problems. Here the step of summing up matrices (which occurs several times in different parts of the calculation) would be done successively while the matrices are "migrating" through the processor array.

www.manaraa.com

References

1. G. Del Re, J. Ladik and G. Biczo, Phys. Rev. **155**, 997 (1967).
2. J.-M. André, G., L. Gouverneur and G. Leroy, Int. J. Quant. Chem. **1**, 427, 451 (1967).
3. J. Ladik in *Quantum Theory of Polymers as Solids*, Plenum Press, New York, London (1988).
4. S. Suhai, Phys. Rev. B **27**, 3506 (1983).
5. J. Ladik, J. Mol. Struct. **206**, 39 (1991)
   P. Otto Phys. Rev. B **45**, 10276 (1992).
6. P. Otto and H. Reif, J. Comp. Phys. Comm. (accepted).
7. H. Früchtl and P. Otto, acm Trans. Math. Softw. (submitted).
8. S. Kindermann, E. Michel and P. Otto, J. Comp. Chem. **13** 414 (1992).
9. P. Otto and H. Früchtl, Comp. and Chem. (accepted).
10. T. Schreiber, F. Hofmann and P. Otto, Parallel Computing (accepted).

www.manaraa.com

On the Parallel Solution of 3D PDEs on a Network of Workstations and on Vector Computers

M. Griebel and W. Huber and T. Störtkuhl and C. Zenger

Institut für Informatik, Technische Universität München
Arcisstraße 21, D-8000 München 2, Germany
e-mail: griebel/huberw/stoertku/zenger@informatik.tu-muenchen.de

Abstract. In this paper we study the parallel solution of elliptic partial differential equations with the sparse grid combination technique. This algorithmic concept is based on the independent solution of many problems with reduced size and their linear combination. We describe the algorithm for three-dimensional problems and discuss its parallel implementation on a network of HP720 workstations and on vector computers.

1 Summary

We present a parallel method for the solution of elliptic partial differential equations. In this so called *combination method* the solution is obtained by a certain linear combination of discrete solutions on different meshes. Then, only $O(h_n^{-1}(\log(h_n^{-1}))^2)$ grid points are needed instead of $O(h_n^{-3})$ for three-dimensional problems, where $h_n = 2^{-n}$ denotes the mesh size. The accuracy of the combination solution is of the order $O(h_n^2(\log(h_n^{-1}))^2)$ provided that the solution is sufficiently smooth. This is only slightly worse than $O(h_n^2)$ obtained for the usual full grid solution.
The natural coarse grain parallelism of the combination method makes it prefectly suited for MIMD parallel computers and distributed processing on workstation networks. $O((\log(h_n^{-1}))^2)$ problems of the size $O(h_n^{-1})$ must be solved independently and can therefore be computed fully in parallel. On a parallel computer or a network with P processors we achieve a data distribution with an amount of data of only $O(h_n^{-1}(\log(h_n^{-1}))^2/P)$ on every processor. Furthermore, the number of communication steps is of order $O(\sqrt{P})$ and the amount of data to be exchanged in each step is $O(h_n^{-1}(\log(h_n^{-1}))^2/(P \cdot \sqrt{P}))$. Therefore, the communication cost is only of $O(h_n^{-1}(\log(h_n^{-1}))^2/P)$ and thus only dependent on the size of data assigned to one processor.
For the three-dimensional case, we report the results of numerical experiments on a network of 110 HP720-workstations and on a CRAY Y-MP4/464.

2 The combination method

We consider a partial differential equation

$$Lu = f$$

www.manaraa.com

in the unit cube $\bar{\Omega} = [0,1]^3 \subset \mathbb{R}^3$ with a linear, elliptic operator L of second order and appropriate boundary conditions.
The usual approach is to discretize the problem by a finite element or finite difference method on an equidistant grid $\Omega_{n,n,n}$ with grid size $h_n = 2^{-n}$ in x-, y- and z-direction and to solve the arising linear system of equations

$$L_{n,n,n}u_{n,n,n} = f_{n,n,n}.$$

Then, we get a solution $u_{n,n,n}$ with error $e_{n,n,n} = u - u_{n,n,n} = O(h_n^2)$, if u is sufficiently smooth. Here, we assume that $u_{n,n,n}$ represents an appropriate interpolant defined by the values of the discrete solutions on grid $\Omega_{n,n,n}$. For the solution of the discrete system, one of the most effective techniques is the multigrid method [8]. There, roughly speaking, the number of operations is of the order $O(h_n^{-2})$ and is therefore proportional to the number of grid points.
Extending this standard approach, we now study linear combinations of discrete solutions of the problem on different rectangular grids. Let $\Omega_{i,j,k}$ be the uniform grid on Ω with mesh sizes $h_i = 2^{-i}$, $h_j = 2^{-j}$ and $h_k = 2^{-k}$ in x-, y- and z-direction, respectively.
To this end, we consider the so-called *combination technique*

$$u^c_{n,n,n} = \sum_{i+j+k=n+2} u_{i,j,k} - 2 \cdot \sum_{i+j+k=n+1} u_{i,j,k} + \sum_{i+j+k=n} u_{i,j,k} \qquad (1)$$

that has been introduced in [6]. Here, i, j, k ranges from 1 to n. This method is illustrated in Fig. 1. Note that our approach can be interpreted as a special case of multivariate extrapolation [2], and is a generalization of the technique in [3] and [9].

Thus, we have to solve $(n+1)\cdot n/2$ different problems $L_{i,j,k}u_{i,j,k} = f_{i,j,k}$, $i+j+k = n+2$, each with about 2^n unknowns, $n\cdot(n-1)/2$ different problems $L_{i,j,k}u_{i,j,k} = f_{i,j,k}$, $i+j+k = n+1$, each with about 2^{n-1} unknowns and $(n-1)\cdot(n-2)/2$ different problems $L_{i,j,k}u_{i,j,k} = f_{i,j,k}$, $i+j+k = n$, each with about 2^{n-2} unknowns and combine their tri-linearly interpolated solutions. This gives a solution defined on the so-called sparse grid $\Omega^s_{n,n,n}$, see Fig. 2. The sparse grid $\Omega^s_{n,n,n}$ is a subset of the associated full grid $\Omega_{n,n,n}$. For further details on sparse grids, see [13].
Altogether, the combination method involves $O(h_n^{-1}\log(h_n^{-1})^2)$ unknowns in contrast to $O(h_n^{-3})$ unknowns for the conventional full grid approach. Additionally, the combination solution $u^c_{n,n,n}$ is nearly as accurate as the standard solution $u_{n,n,n}$. It can be proved (see [6]) that the error satisfies

$$e^c_{n,n,n} = u - u^c_{n,n,n} = O(h_n^2\log(h_n^{-1})^2),$$

(pointwise and with respect to the L_2- and L_∞-norm). This is only slightly worse than for the associated full grid where the error is of the order $O(h_n^2)$. For the proof we assume that u is sufficiently smooth, so that for $u_{i,j,k}$ (interpolated from grid $\Omega_{i,j,k}$ to the domain Ω) an error splitting of the form

$$\begin{aligned} u_{i,j,k} \;&= u + C_1(h_i)h_i^2 + C_2(h_j)h_j^2 + C_3(h_k)h_k^2 + \\ &D_1(h_i,h_j)h_i^2h_j^2 + D_2(h_i,h_k)h_i^2h_k^2 + D_3(h_j,h_k)h_j^2h_k^2 + \\ &E(h_i,h_j,h_k)h_i^2h_j^2h_k^2 \end{aligned} \qquad (2)$$

www.manaraa.com

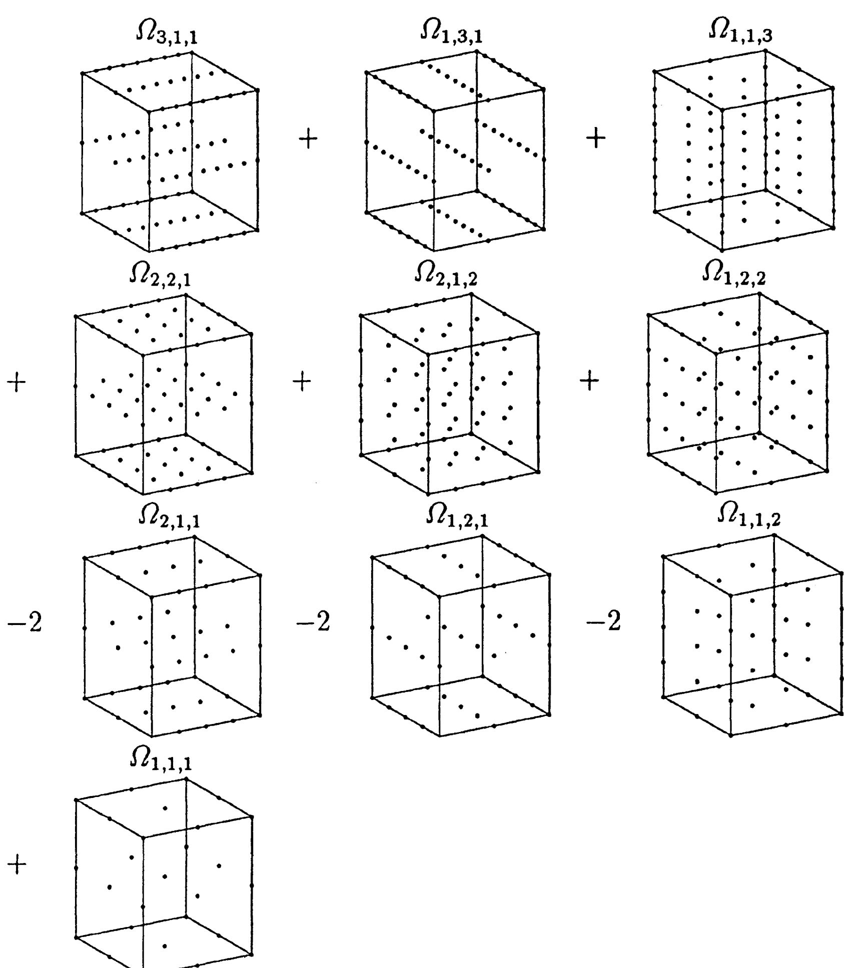


Fig. 1. The linear combination of grids $\Omega_{i,j,k}$ with i+j+k=n+2, i+j+k=n+1 and i+j+k=n, n=3.

holds for any fixed point $(x, y, z) \in \Omega$ with functions $C_1(h_i)$, $C_2(h_j)$, $C_3(h_j)$, $D_1(h_i, h_j)$, $D_2(h_i, h_k)$, $D_3(h_j, h_k)$, and $E(h_i, h_j, h_k)$ bounded by a constant C for all h_i, h_j, h_k. This is a requirement different (and weaker) than what is used for usual Richardson extrapolation. If we insert the error splitting formula (2) into (1), we see that the leading error terms cancel. We get the estimation

$$|u - u^c_{n,n,n}| \leq C \cdot \; h_n^2 \cdot (1 + 65/32 \log(h_n^{-1}) + 25/32 \log(h_n^{-1})^2) = O(h_n^2 \log(h_n^{-1})^2).$$

The combination technique is not restricted to the unit cube. In the two-dimensional

www.manaraa.com

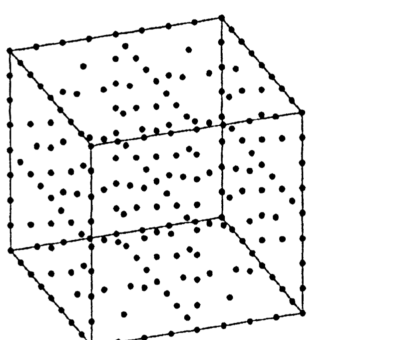
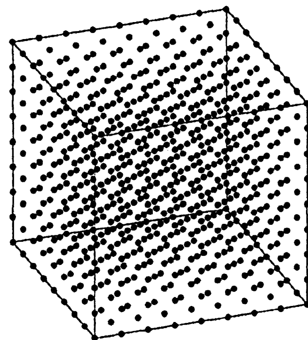



Fig. 2. The sparse grid $\Omega^s_{3,3,3}$ and the associated full grid $\Omega_{3,3,3}$.

case we have successfully treated problems on distorted quadrilaterals, triangles, and more general domains with polygonal boundaries. Additionally, problems with a non-linear operator and PDE-systems like the Stokes and Navier-Stokes equations have been solved (see [6] and [7]).
To some extent, the combination method even works in the case of non-smooth solutions ([6]). However, for problems with severe singularities the appropriate combination of adaptively refined grids is recommended. For further results on the combination method as sparse grid problem solver (see [3], [4], [5], and [6]).
We remark that all the different discrete problems whose solutions have to be combined are totally independent of each other and can be solved fully in parallel. This will be exploited in the following section.

3 Parallelization aspects of the combination method on distributed memory systems

Now, we study the parallelization properties of the combination technique for distributed memory computers and networks of workstations. On these types of parallel computing systems, any efficient parallel implementation of an algorithm needs a reasonable load distribution and balancing strategy and a sophisticated communication procedure that minimizes the data exchange.

3.1 Parallel execution and a simple load balancing strategy

The combination method provides a straightforward way for parallelization. For our three-dimensional problem, $n \cdot (n+1)/2$ problems with about 2^n unknowns, $(n-1) \cdot n/2$ problems with about 2^{n-1} unknowns, and $(n-2) \cdot (n-1)/2$ problems with about 2^{n-2} unknowns can be solved fully in parallel (see also (1)). This parallelization potential of the combination method can be gained already on a relatively coarse grain level and makes it perfectly suited for distributed memory computers and networks of workstations.

www.manaraa.com

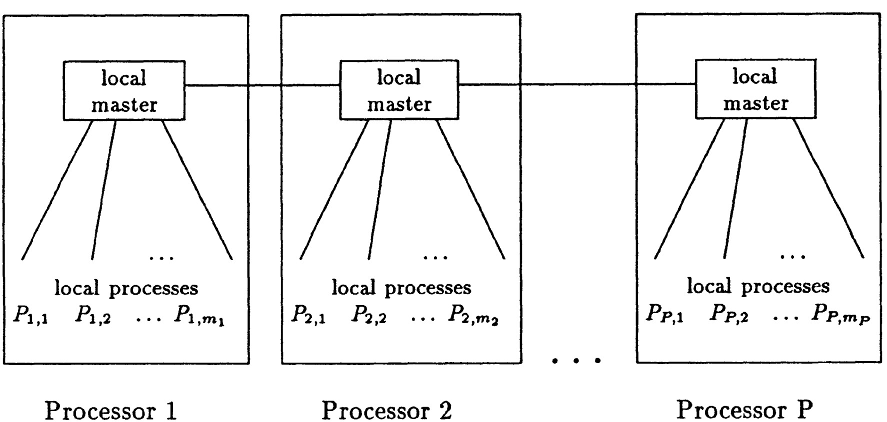


Fig. 3. Configuration of processes on P different processors.

Additionally, the subproblem solver (e.g. multigrid) can be parallelized. This, however, requires a comparatively fine grain parallelization. In addition to the natural coarse grain parallelism, this can only be exploited on massively parallel systems with thousands of processors. Alternatively, the subproblems are ideal for vectorization In this paper, we focus on the coarse grain parallelization approach only.

Let the solution $u_{i,j,k}$ of each subproblem on the grid $\Omega_{i,j,k}$ that arises in the formula (1) be computed by a multigrid-like method where the number of operations involved is proportional to the number of grid-points of $\Omega_{i,j,k}$. We will not go into details but consider the respective procedure as a given black-box solver that is implemented as a UNIX-process, for example. Note however, that the respective multigrid method also has to work for distorted grids, eg. by employing semi-coarsening techniques. For details on multigrid-methods, see [8]. Of course, alternatively, any other (suboptimal but available) solver could instead be plugged in.

Now, the task remains to distribute the different solution processes among the available processors or workstations and to set up an appropriate communication structure between the processes *and* between the processors. Compare also Fig. 3.

First, let us consider the problem how to distribute the different solution processes efficiently. The different sizes of the various problems suggest the following simple load balancing strategy: Assume that the problems are sorted according to their size where the size is specified by the number of interior grid points of the respective problem. Assume further, that P processors are available. Then, the first P problems are distributed among the P processors. If after this step not all problems have been assigned to processors, the remaining problems are distributed among the processors in such a way, that now the largest problem is assigned to one of the processors with the smallest associated problem. This scheme is continued until all problems are distributed among the processors. See Fig. 4 for a simple example with n=3 and $P = 4$.

Here, 10 problems are distributed among four processors (according to the explained

www.manaraa.com

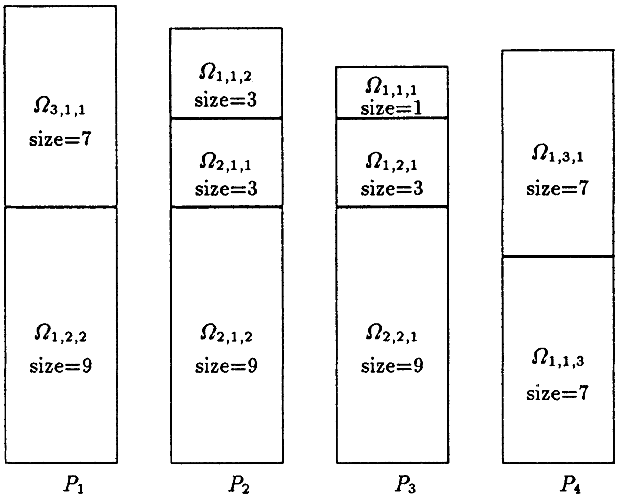


Fig. 4. A first simple load balancing strategy, $P = 4$.

load balancing strategy). So, processor P_1 has to compute problems of a total size of 16, P_2 has to compute problems of total size of 15, P_3 has to compute problems of total size of 13 and P_4 has to compute problems of total size of 14. Since we assign a whole problem to a process we are of course not able to obtain a perfectly balanced load distribution. However, for sufficiently large values of n, we obtain between two processors a

$$\begin{array}{rl} \text{maximum load difference:} & O(2^{n-1}) \\ \text{and a load per processor:} & O(h_n^{-1}(\log(h_n^{-1}))^2/P). \end{array} \tag{3}$$

To achieve a better load balancing, we would have to further subdivide the different problems by means of the domain decomposition technique. However, this complicates the subproblem solver and results in additional communication requirements.

3.2 Distributed memory, distributed data and efficient communication

In the previous section 2 we have already seen that the combination technique reduces the number of grid points and thus the memory requirements from $O(h_n^{-3})$ for the full grid case to $O(h_n^{-1}(\log(h_n^{-1}))^2)$ for the sparse grid case. Already in the sequential version, this allows the computation of problems on much finer grids than in the full grid case or, alternatively, a much faster computation of the solution on the sparse grid than on the corresponding full grid since substantially less operations are now involved.

www.manaraa.com

On a parallel computing system with distributed memory, the data for the combination solution can be processed and stored in a certain *distributed* manner that makes the combination method even more attractive.
First, we note that each problem arising in the combination formula (1) involves only $O(h_n^{-1})$ unknowns. Now, we do not perform the summation and subtraction operations of (1) to form the combination solution $u^c_{n,n,n}$ explicitly but just keep all the different solutions $u_{i,j,k}$ in the memory of the processor where they have been computed in parallel anyway. In this way, the combination solution is stored implicitly in terms of the operands of the formula (1) but it is not yet processed explicitly.
In practice, the solution is often wanted in certain special points or in a small sub-

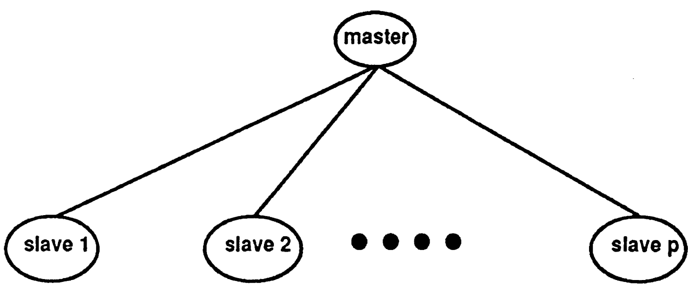


Fig. 5. Communication $1 \leftrightarrow p$ bottleneck

domain of Ω only. Furthermore, in some applications, only certain integral values of the solution, like for example the Nusselt-number ([10]) in laminar flow problems or the rms-mean values in turbulence simulation ([12]) is of interest. Thus, it is prohibitive to compute the combination solution $u^c_{n,n,n}$ explicitly and to process the sought values from it. Often, it is more advisable to compute the values of interest first from each $u_{i,j,k}$ separately (and in parallel) and only to combine these values analogously to (1). This avoids unnecessary communication between the processes and processors and overcomes the problem to maintain additional main memory for the explicitly computed $u^c_{n,n,n}$ on one of the available processors.
Now, let us explain the further consequences of the distributed, implicit storing of the combination solution in terms of its combination operands $u_{i,j,k}$ by a praxis-relevant example. Consider a non-linear problem like the Navier-Stokes equations. A common approach to deal with nonlinearity is to linearize the discretized equations in an outer loop by a suitable method (e.g. Newton) and to iterate the arising linear system in an inner loop by a few multigrid V-cycles. This results in the well known and quite robust Newton-MG-method ([8]).
Now, we substitute the combination method for the standard multigrid algorithm in the inner loop. Thus, we obtain an algorithm, where in every step an outer loop iteration needs the computation of a combination function (e.g. some sort of residual) on the sparse grid similar to $u^c_{n,n,n}$.
This type of algorithm also appears if we apply the combination method within an implicit time-stepping algorithm, together with the domain decomposition method,

www.manaraa.com

in a defect correction preconditioner for the sparse grid finite element method, together with outer Richardson-type extrapolation [2] or within a solver for parabolic problems.
At first glance, it seems that for this class of algorithms the combination solution $u^c_{n,n,n}$ and their relevant outer loop counterparts (e.g. some sort of residual) have to be accumulated explicitly. This would cause a $1 \leftrightarrow p$ bottleneck for the communication (see Fig.5) and results in a number of $O(P)$ communication steps where the size of data to be exchanged is of $O(h_n^{-1}(\log(h_n^{-1}))^2/P)$. Additionally, the storage requirement for the master process where the combination solution is assembled explicitly would be of $O(h_n^{-1}(\log(h_n^{-1})^2)$. However, this can be avoided by using the distributed implicit storage scheme of the combination solution in terms of its combination operands $u_{i,j,k}$.
The basic idea is never to assemble $u^c_{n,n,n}$ explicitly but only to exchange that data between adjacent processors that is relevant and necessary for the respective algorithm. Thus, the outer loop computation can be performed locally and in a distributed and parallel manner. This reduces the communication costs substantially and results in a slight modification of our first simple load balancing strategy. Now,

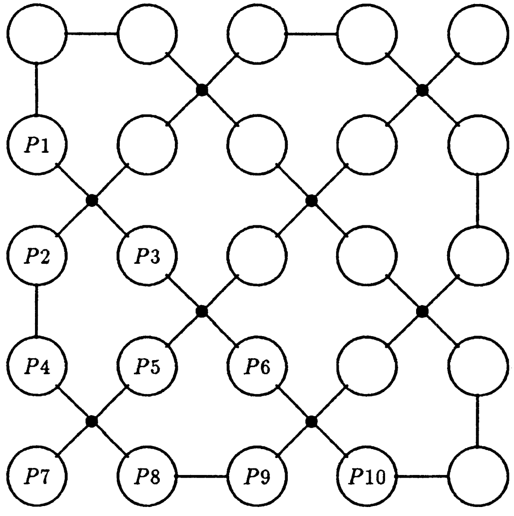


Fig. 6. An array-like topology for a network of workstations

we are not longer only interested in balancing the processor loads as good as possible but we additionally want our problems and their respective processes to be distributed in a way, that communication only has to take place between adjacent processors and that the amount of data to be communicated is minimized in some sense.

www.manaraa.com

An analysis of the outer loop type algorithm with inner loop combination method solver showed that we only have to exchange the data between two processors P_α and P_β that is associated

$$\Big(\bigcup_{\Omega_{i,j,k} \text{ on } P_\alpha} \Omega_{i,j,k}\Big) \bigcap \Big(\bigcup_{\Omega_{i,j,k} \text{ on } P_\beta} \Omega_{i,j,k}\Big) = D_{\alpha\beta}. \tag{4}$$

Now, for the approach as described in Fig. 5, it is not longer necessary to maintain memory of the size $O(h_n^{-1}(\log(h_n^{-1}))^2)$ in one processor to store $u^c_{n,n,n}$ explicitly, but the number of communication steps would still be $O(P)$ and the amount of data to be exchanged in each step would still be $O(h_n^{-1}(\log(h_n^{-1}))^2/P)$.

This can be improved by the following method. First, we arrange the processors in

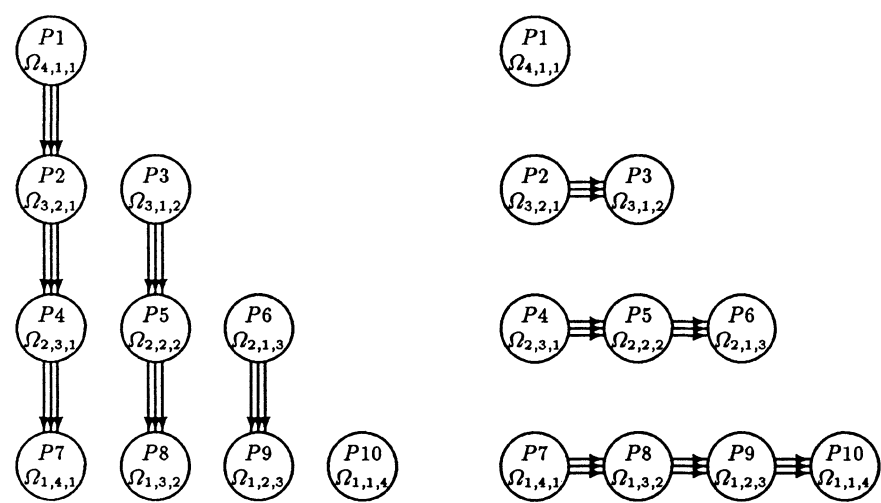


Fig. 7. Exchange of data between adjacent processors in an array-like topology

an array-like topology (see Fig.6). Now, we exploit the possibility that the processors can exchange data at the same time in parallel. The main idea is that the relevant data $D_{\alpha\beta}$ of (4) can be partitioned into disjoint subsets, that is

$$D_{\alpha\beta} = \biguplus_{i=1}^{s} D^i_{\alpha\beta}, \tag{5}$$

where s equals nearly the square root of P, i.e. $s \simeq \sqrt{P}$. Now, it can be shown that for any $D_{\alpha\beta}$ and $D_{\beta\gamma}$ of two adjacent pairs of processors (P_α, P_β) and (P_β, P_γ), a

www.manaraa.com

data partitioning (5) can be found with

$$D^i_{\alpha\beta} \quad \bigcap \quad D^i_{\beta\gamma} \quad = \emptyset \tag{6}$$

for all $i = 1 \dots s$. Thus, these disjoint sets of data $D^i_{\alpha\beta}$ and $D^i_{\beta\gamma}$ can be exchanged *fully in parallel* between any two adjacent pairs of processors (P_α, P_β) and (P_β, P_γ). The surprising advantage of this data partitioning is, that, in $O(\sqrt{P})$ communication steps, the whole relevant data $D_{\alpha\beta}$ is not only transferred between adjacent processors, but also between *any* pair of processors P_α and P_β in the given array-like topology of processors. This is achieved by using only communication between adjacent processors. Thus, even for $P = O((\log(h_n^{-1}))^2)$ processors, we need only $O(\sqrt{P})$ communication steps. In each step, the size of data to be exchanged between adjacent processors is $O(h_n^{-1}/\sqrt{P})$. Therefore, we achieve the remarkable communication complexity of order $O(h_n^{-1})$. Thus, the cost of communication is theoretically *independent* of the number of processors.
We can estimate the whole communication time by

$$t = c \cdot T + \sqrt{P} \cdot t_{setup}, \tag{7}$$

where T is proportional to the communication time to exchange data of the size $O(h_n^{-1})$, c is some constant independent of the number of processors P and t_{setup} denotes the setup-time for each communication between adjacent processors. For practical applications the setup-time is in the range of *msec* whereas T grows linear with the size of data that is of several *Mbyte*. Thus, the communication is totally dominated by T and practically independent of the number P of processors. Due to this fact, we obtain a scalable parallel communication up to the number of $O((\log(h_n^{-1}))^2)$ of processors in the array-like topology of Fig. 6.
To show the main ideas in more detail we discuss the communication algorithm for the first summation of the operands $u_{i,j,k}$ of formula (1) with $i + j + k = n + 2$ and $n = 4$. For reasons of simplicity, we neglect the other summation and subtraction operations and state that these can be managed in an analogous way.
So, we have to consider $n \cdot (n+1)/2 = 10$ problems on the different grids $\Omega_{i,j,k}$, $i + j + k = 6$. Each of the problems is computed on one of the $P = 10$ processors (Fig.6) in parallel. Now, our task is to combine the computed data of each grid $\Omega_{i,j,k}$ according to formula (1) by parallel communication only between adjacent processors. Thus, we subdivide the data of any two adjacent pairs of processors like in (5) in such a way that (6) is fulfilled. Thus, for example, we get disjoint sets of data D^i_{12}, D^i_{24} and D^i_{47}, $i = 1, 2, 3$, for the adjacent pairs of processors (P_1, P_2), (P_2, P_4) and (P_4, P_7) within the first column of processors of Fig.6. Analogous data partitioning in the other columns of Fig.6 enables us to exchange the $D^i_{\alpha\beta}$ between any pair of processors (P_α, P_β) at the same time in parallel within a column and also within a row of Fig.6.
Thus, first, the communication takes place within the columns downward for all $i = 1, 2, 3$. In 3 communication steps, all processors of a column have updated its operand $u_{i,j,k}$ by the relevant data of all processors placed above in Fig.6 according to formula (1). Secondly, the communication algorithm sweeps within a row from the left to the right in 3 further communication steps. These two first sweeps of our communication algorithm are visualized in Fig.7. For example, after the second

www.manaraa.com

sweep, P_{10} already contains all updated relevant data of $u_{1,1,4}$. At this moment all other processors need data from the processor placed right or below in Fig.6. Thus, the data exchange algorithm sweeps the other way around, first, within the rows from the right to the left and then within the columns upward. (Now, the received data must not be processed but only stored.) This procedure also needs 6 communication steps. Altogether, the number of communication steps is nearly $4 \cdot \sqrt{P}$, the size of data to be exchanged in each step is $O(h_n^{-1}/\sqrt{P})$ and the whole amount of communication is $O(h_n^{-1})$.
Furthermore, we are able to map the discussed parallelization and communication structure of Fig.7 to an array-like *network of workstations* (see Fig.6). Here, each workstation possesses only two links and is connected with up to four other workstations, e.g. by a local Ethernet. Thus, we are able to efficiently perform a broadcast from each processor to all others by using only communication between adjacent processors with communications costs of $O(h_n^{-1})$, independent of the number of processors.

4 Experiments on a network of high performance workstations

In this section we turn to the results of our numerical experiments. To simplify the presentation we consider the simple three-dimensional model problem

$$\Delta u = 0 \quad \text{in } \Omega = (0,1) \times (0,1) \times (0,1) \tag{8}$$

with Dirichlet boundary conditions on $\partial\Omega$ and the unique solution

$$u = \sin(\pi x) \cdot \sin(\pi y) \cdot \sinh(\sqrt{2}\pi z)/\sinh(\sqrt{2}\pi).$$

For the solution of the subproblems we use 10 V-cycles of a multigrid method with one step of eight color Gauss-Seidel pre-smoothing and one post-smoothing iterations. Note that our code uses full 27-point stencils that are derived from assembling tri-linear rectangular finite elements.
First, we turn to the implementation of the combination method on a network of 110 HP720 workstations. The workstations are organized in 11 clusters. Each cluster consists of a disc-server and 10 discless clients, each server with 32 MByte and each client with 16MByte main memory (see Fig. 8). It is noteworthy that the total memory capacity of this system is more than 1.9 GByte and thus larger than many machines presently considered as supercomputers. The HP720 workstation is based on HP's Precision RISC-architecture. This architecture achieves a peak-rate of 50 MFlop per second, so that the peak rate of the network would be 5.5 GFlop per second.
However, the peak rate is rarely obtained in practice. Even on a single workstation, the peak performance can only be obtained when floating point operands are stored in registers. The registers can be considered as the top level of a memory hierarchy, consisting of registers, cache (256 KByte), main memory, and, finally, virtual memory. If data must be transferred from slower to faster memory in the hierarchy, the

www.manaraa.com

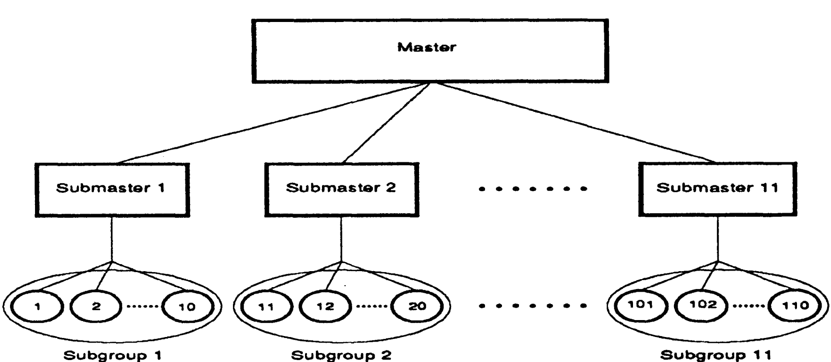


Fig. 8. Workstation network configuration.

performance may drop dramatically. Our present code (written in FORTRAN), that has been specially optimized, runs at more than 13 MFlop per second on a single processor.

Of course, the application of the network as a parallel supercomputing system is handicapped by several constraints. The Ethernet connection is very slow compared to the processing speed of a single workstation. Furthermore, it is not possible to use the workstations as a dedicated system. Even at night hours when no regular users are present, it is difficult to control the effect of background processes on the elapsed time and performance. This is further complicated by the lack of tools supporting the evaluation of distributed applications in the network.

For a first experiment we only implemented the simple communication structure of section 3.1 (see Fig. 5) on the network configuration of Fig. 8. In detail, we used the following procedure: A shell script is used to start the execution of the slave processes. This is done in two levels. The top-master starts cluster-master processes to the cluster servers. The cluster-masters then distribute the slave processes on the cluster-clients. Each master waits for the termination of its sub-masters or slaves, respectively, which stores the computed result in an associated file (e.g. a named pipe). When all slaves have terminated, the master collects the results from the files to calculate the final solution. For this purpose, we use the network file system (NFS).

With this implementation we measured the times that are shown in Table 1. Table 2 shows the performance. With 110 slave processors we achieved a rate of approximately 1.1 GFlop per second. Fig. 9 shows the resulting speed up and efficiency for varying problem sizes and processor numbers. There, an efficiency of more than 70 percent can be seen for a sufficient large value of n.

5 Experiments on the CRAY Y-MP4/464 vector computer

Now, we turn to the results gained on vector computers. First, note that the MG solver applied in the combination method for the solution of each arising problem can be vectorized easily. This results in a substantial speed up in comparison to scalar

www.manaraa.com

Table 1. Times (in sec.) for the combination algorithm on $\Omega^s_{n,n,n}$ for a network with P HP720-workstations as slaves.

$P\backslash n$	4	5	6	7	8	9	10	11	12	13	14
1	0.11	0.34	1.01	2.90	8.01	21.42	55.24	139.60	351.84	943.37	2503.38
2	0.07	0.20	0.55	1.53	4.06	10.89	28.23	71.87	179.84	481.58	1265.47
4	0.05	0.12	0.29	0.83	2.07	5.50	14.33	35.54	90.09	241.57	644.21
8	-	-	0.20	0.42	1.14	2.75	7.29	18.55	45.34	123.83	328.31
16	-	-	0.12	0.22	0.61	1.45	3.64	9.14	22.93	63.69	166.25
32	-	-	0.08	0.16	0.32	0.90	2.05	4.84	12.36	36.59	89.66
64	-	-	0.24	0.31	0.44	0.67	1.38	2.68	6.63	18.16	47.45
110	-	-	-	-	-	-	1.26	2.40	4.72	12.81	30.42

Table 2. MFlop per second for the combination algorithm on $\Omega^s_{n,n,n}$ for a network with P HP720-workstations as slaves.

$P\backslash n$	4	5	6	7	8	9	10	11	12	13	14
1	7.90	10.00	11.57	12.65	13.45	13.94	14.43	14.73	14.67	13.44	12.21
2	12.40	17.00	21.25	23.97	26.53	27.42	28.24	28.61	28.70	26.32	24.16
4	17.36	28.33	40.31	44.19	52.03	54.29	55.63	57.86	57.28	52.47	47.46
8	-	-	58.45	87.33	94.48	108.57	109.38	110.86	113.82	102.36	93.13
16	-	-	97.42	166.73	176.57	205.91	218.99	225.00	225.06	199.01	183.91
32	-	-	146.13	229.25	336.59	331.74	388.85	424.89	417.53	346.40	341.00
64	-	-	48.71	118.32	244.80	445.63	577.64	767.34	778.38	697.95	644.35
110	-	-	-	-	-	-	632.65	856.86	1093.36	989.45	1005.08

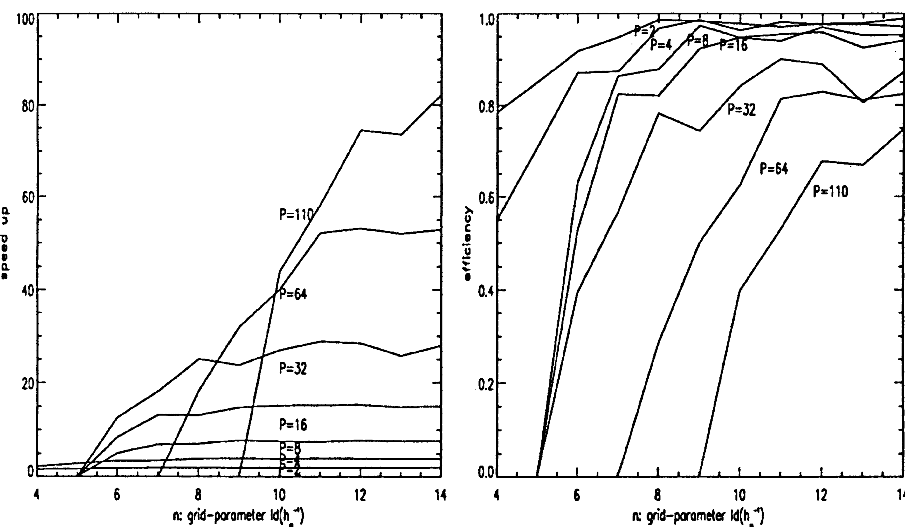


Fig. 9. Speed up and efficiency of the network measured with respect to P slave-workstations.

المنارة للاستشارات
www.manaraa.com

processors. Furthermore, if more than one processing unit is present, the storage of the machine is treated as *shared memory*. Therefore, the parallel access to the same part of the memory has to be sequentialized. To some extend, this can automatically be optimized by the compiler. However, such conflicts between successive accesses of different processes to the same memory bank result in additional time needed for conflict resolution.
We presented in the previous section the concept of distributed data storage and parallel communication by using a partitioning of the relevant data to be exchanged between adjacent processors. Now, this data partitioning concept can be used to avoid memory contention. Consequently, the parallel use of several processing units of the vector computer is possible without significant loss of performance.
For our experiments we used a CRAY Y-MP4/464 with *four* processing units and 64 megawords main memory. The CRAY Y-MP is a typical vector computer with pipelining concept and a vector length of 64 elements. It has a peak-rate of 330 MFlop per second and per processing unit, so that the peak rate of the four processing units is about 1.3 GFlop per second. The four processing units can be used to compute different problems in parallel. In practice, we obtain with our code for one processing unit about 80 MFlop per second and for 4 processing units we obtain up to 200 MFlop per second.

Table 3. Time (in sec.) for the combination algorithm on $\Omega^s_{n,n,n}$ on a CRAY Y-MP4/464 with P processing units.

$P\backslash n$	4	5	6	7	8	9	10	11	12	13	14
1	0.09	0.25	0.60	1.39	3.09	6.87	14.95	33.33	73.40	163.30	361.88
2	0.05	0.20	0.53	1.30	2.98	4.18	9.35	16.84	39.75	85.24	201.66
3	0.04	0.14	0.37	1.09	2.55	4.79	5.62	12.00	28.85	69.70	171.43
4	0.04	0.09	0.33	0.73	1.99	4.51	5.28	12.20	28.73	61.58	159.69

Table 4. MFlop per second for the combination algorithm on $\Omega^s_{n,n,n}$ on a CRAY Y-MP4/464 with P processing units.

$P\backslash n$	4	5	6	7	8	9	10	11	12	13	14
1	14.56	18.14	23.08	23.31	36.97	44.81	53.80	61.41	69.31	76.01	82.15
2	25.33	22.05	26.03	31.35	38.34	73.65	86.05	121.60	128.02	145.53	147.41
3	31.97	31.95	37.24	37.25	44.77	64.25	143.34	170.58	188.91	177.92	173.44
4	31.97	50.36	41.74	55.90	57.27	68.20	152.32	167.67	177.14	201.26	186.24

We implemented the combination method on a CRAY Y-MP4/464 in FORTRAN using the autotasking facility cft77 with compiler version 4.0. Here, the solution of each problem is computed by a *vectorized* version of the multigrid algorithm. The

www.manaraa.com

parallel treatment of the different problems is indicated explicitly by the compiler directive CFPP$ CNCALL that exploits parallelism in *concurrent loops*.
For the parallel version of the combination algorithm (parallel treatment of the different problems by explicit compiler directives) which was run on a CRAY Y-MP4/464 with 4 processors, we measured the run times as shown in Table 3. The MFlop per second are shown in Table 4. Due to the fact that on the used CRAY Y-MP4/464 not all four processing units but only 3 processing units are available in dedicated mode for one user, the measurements show a decrease of efficiency in the case of $P = 4$ processing units.
We achieve on the CRAY fast execution times with fairly good speed up (e.g. with 4 processors 2.83 for n=10) and efficiency (e.g. with 4 processors 70.75% for n=10). Additionally, for large n we obtain quite high MFlop rates.
In principle the gain by parallelization behaves analogous to the results of the workstation cluster, but on a much better level. This is of course due to the superior processors of the CRAY and the vectorization. Nevertheless, with a network of 110 workstations the comparable algorithm ran at about five times faster than on a CRAY Y-MP with 4 processors and nearly the peak performance of the CRAY Y-MP was reached (compare Table 2 and Table 3).

6 Concluding remarks

In this paper, we discussed the distributed and parallel solution of partial differential equations by combination type algorithms on a workstation network and on a CRAY Y-MP4/464.
The experiments show that for the combination technique a comparatively simple parallel computing model already leads to an efficient parallel implementation. However, with the discussed load balancing strategy and parallel communication on an array-like topology of processors, we are able to avoid a bottleneck in communication. Thus, in the future time we will implement the combination method on an array-like network of workstations by using the parallel communication structure discussed in 3.2. We expect this method to perform better with respect to the communication requirements. Furthermore, the scalability of the system and the combination method offers the possibility to compute very large problems with a high performance that arise in many practical fields like for example in fluid dynamics.
For a certain class of practical algorithms we have seen that a network of workstations can be a quite powerful parallel computing system. We believe that with the development of fast FDDI based communication links a network of workstations will be a real competitor to standard MIMD computers in near future.
Hopefully, with parallel development and computing systems like TOPSYS (see [1]) or PVM (see [11]), a software basis becomes available that promises better performance together with improved monitoring and evaluation tools on workstation networks.

www.manaraa.com

References

1. S. Baker, H.-J. Beier, T. Bemmerl, A. Bode, H. Ertl, U. Graf, O. Hansen, J. Haunerdinger, P. Hofstetter, R. Knödlseder, J. Kremenek, S. Langenbuch, R. Lindhof, T. Ludwig, P. Luksch, R. Milner, B. Ries, and T. Treml, *TOPSYS – tools for parallel systems*, SFB Bericht 342/13/91 A, Institut für Informatik, TU München, June 1991.
2. H. Bungartz, M. Griebel, and U. Rüde, *Extrapolation, Combination and Sparse Grid Techniques for Elliptic Boundary Value Problems*, in International conference on spectral and high order methods, ICOSAHOM 92, C. Bernardi and Y. Maday, eds., Elsevier, 1992. also available as SFB Bericht 342/10/92 A.
3. M. Griebel, *The combination technique for the sparse grid solution of PDE's on multiprocessor machines*, Parallel Processing Letters, 2 (1992), pp. 61–70. also available as SFB Bericht 342/14/91 A.
4. ———, *Sparse grid multilevel methods, their parallelization, and their applications to CFD*, in Proceedings of the conference of Parallel Computational Fluid Dynamics, J. Häuser, ed., Elsevier, May 1992.
5. M. Griebel, W. Huber, U. Rüde, and T. Störtkuhl, *The Combination Technique for Parallel Sparse-Grid-Preconditioning and -Solution of PDEs on Multiprocessor Machines and Workstation Networks*, in Proceedings of the Second Joint International Conference on Vector and Parallel Processing CONPAR/VAPP V 92, L. Bouge and M. Cosnard and Y. Robert and D. Trystram, ed., Springer Verlag, 1992. also available as SFB Bericht 342/11/92 A.
6. M. Griebel, M. Schneider, and C. Zenger, *A combination technique for the solution of sparse grid problems*, SFB Bericht 342/19/90, Institut für Informatik, TU München, October 1990. Also to be published in: Proceedings of the International Symposium on Iterative Methods in Linear Algebra, Bruxelles, April 2-4, 1991.
7. M. Griebel and V. Thurner, *The efficient solution of fluid dynamics problems by the combination technique*, sfb bericht, Institut für Informatik, TU München, 1992. to be published.
8. W. Hackbusch, *Multigrid Methods and Applications*, Springer Verlag, Berlin, 1985.
9. J. P. Hennart and E. H. Mund, *On the h- and p-versions of the extrapolated gordon's projector with applications to elliptic equations*, SIAM J. Sci. Comput., 9 (1988), pp. 773–791.
10. M. Peric, M. Schaefer, and E. Schreck, *Numerical solution of complex fluid flows on MIMD computers*, in Proceedings of the conference of Parallel Computational Fluid Dynamics, J. Häuser, ed., Elsevier, May 1992.
11. V. Sunderam, *PVM: a framework for parallel and distributed computing*, tech. report, Emory University, Department of Mathematics and Computer Science, Atlanta, GA 30322, USA, 1991.
12. H. Tennekes and J. L. Lumley, *A first course in turbulence*, M.I.T. Press, 1972.
13. C. Zenger, *Sparse grids*, in Parallel Algorithms for Partial Differential Equations, Proceedings of the Sixth GAMM-Seminar, Kiel, January 19-21, 1990, W. Hackbusch, ed., Braunschweig, 1991, Vieweg-Verlag.

www.manaraa.com

Numerical Simulation of Complex Fluid Flows on MIMD Computers

M. Perić, M. Schäfer and E. Schreck

Lehrstuhl für Strömungsmechanik, Universität Erlangen-Nürnberg,
Cauerstr. 4, D-8520 Erlangen, Germany

Abstract. The paper analyses the efficiency of parallel computation of incompressible fluid flows using a fully implicit finite volume multigrid method. The parallelization is achieved via domain decomposition, which is chosen for its suitability in complex geometries when blockstructured or unstrucutured grids are employed. Numerical efficiency (increase of computing effort to reach converged solution) and parallel efficiency (increase of runtime due to local and global communication) are analysed for a typical recirculating flow induced by buoyancy. Good efficiencies are found and the possibilities for further improvement by avoiding some global communication or by simultaneous computation and communication are indicated.

1 Introduction

Calculation of fluid flows in complex geometries requires the use of either block-structured or unstructured grids. Except for extremely complex configurations, the block-structured grids have the advantage of allowing the use of efficient solvers developed for structured grids.

When solving steady flow problems (either laminar or Reynolds-averaged turbulent flows), implicit methods are usually used. In most cases, the variables are decoupled and linearized equations for each variable are solved in turn. The solution method involves two iteration levels, which are here called *outer* and *inner* iterations. In the inner iterations, large linear equation systems are solved, whose coefficient matrix is sparse and within each block has a diagonal structure. The outer iteration loop provides for the update of the coefficient and source matrices in order to take into account the non-linearity and coupling of the equations for the individual variables.

The outer iterations are explicit in nature, since the coefficient and source matrices are calculated using variable values from the previous outer iteration. This part of the solution algorithm is therefore easily parallelized. On the other hand, inner iterations are implicit and pose special requirements to be performed in parallel. While the Jacobi, the so called "red-black" Gauß-Seidel iteration methods and conjugate gradient solvers can be parallelized in a straightforward way, the parallelization of the ILU method without changing the algorithm introduces idle times for some processors and is not always efficient (Bastian and Horton, [1]).

www.manaraa.com

Recently, the present authors presented a parallel implicit solution method based on grid partitioning technique, using the ILU solver after Stone [9] for the inner iterations and a full-approximation multigrid scheme for the outer iterations (Schreck and Perić, [8]; Perić *et al*, [7]). This approach is easy to implement and can be used for both block-structured and unstructured grids. The present paper concentrates on the analysis of the numerical and parallel efficiency of the parallel solution method and indicates the ways of improving them.

2 Description of Solution Method

The solution method used in this study is described in detail by Demirdžić and Perić [2], so only a summary of main features will be given here. The method is of finite volume type and uses non-orthogonal boundary-fitted grids with a colocated arrangement of variables. Figure 1 shows a typical control volume (CV). The working variables are the cartesian velocity components, pressure and temperature. The continuity equation is used to obtain a pressure-correction equation according to the SIMPLE algorithm (Patankar and Spalding, [6]). Second order discretization is used for all terms (central differences, linear interpolation). The part of diffusion fluxes which arises from grid non-orthogonality is treated explicitly. The convection fluxes are treated using the so called "deferred correction" approach: only the part which corresponds to the first order upwind discretization is treated implicitly, while the difference between the central differencing and upwind fluxes is treated explicitly. The effect of non-orthogonality is also treated explicitly in the pressure-correction equation. In the first step, the pressure-correction equation is solved with these terms excluded (which suffices if the grid is not severely non-orthogonal). In the second step the non-orthogonal contribution is evaluated using the pressure correction calculated in the first step, and a second pressure-correction equation is solved. It has the same coefficient matrix as the first one but a different source term.

Equations for the cartesian velocity components U and V, pressure correction P' and temperature T are discretized and solved one after another. Linear algebraic equation systems are solved iteratively using either the Gauß-Seidel method (GS) or the ILU-decomposition (SIP) after Stone [9]. Inner iterations are stopped either after reducing the absolute sum of residuals over all CVs by a specified factor, or after a prescribed number of iterations has been performed. Outer iterations are performed to take into account the non-linearity, coupling of variables and effects of grid non-orthogonality; this is why the linear equations need not be solved accurately. Computation is stopped when at the beginning of an outer iteration the sum of absolute residuals over all CVs in all equations becomes 4 to 5 orders of magnitude smaller than the initial values. This roughly corresponds to a 4–5 digit accuracy. For steady flow calculations considered in this study, under-relaxation is used to improve the convergence of outer iterations. Typical values of under-relaxation factors range between 0.5 and 0.8 for the velocity components and temperature. The fraction of pressu-

www.manaraa.com

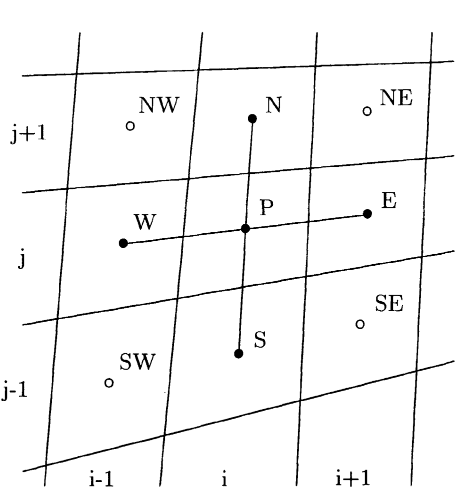


Fig. 1. A typical control volume and a computational molecule

re correction added to pressure after solving the pressure-correction equation is typically equal to $1 - \alpha_u$, where α_u is the under-relaxation factor for velocities. This leads to a nearly-optimum convergence rate. A flow diagram of the outer and inner iterations (for the SIP solver) is shown in Fig. 2.

The number of outer iterations increases linearly with the number of CVs, leading to a quadratic increase in computing time. For this reason a multigrid method is implemented, which keeps the number of outer iterations approximately independent of the number of CVs. The method is based on the so called "full approximation scheme" (FAS). It is implemented in the so called "full multigrid" (FMG) fashion. The solution is first obtained on the coarsest grid using the method described above, cf. Fig. 2. This solution provides initial fields for the calculation on the next finer grid, where the multigrid method using V-cycles is activated. The procedure is continued until the finest grid is reached. The coarse grids are subsets of the finest grid; each coarse grid CV is made of four CVs of the next finer grid. The equations solved on the coarse grids within a multigrid cycle contain an additional source term which describes the current solution and the residuals of the next finer grid. The multigrid method used in this study is described in detail in Hortmann *et al* [5]. It should be noted that the multigrid method is applied only to the outer iteration loop; inner iterations are performed with one of the above described solvers irrespective of the grid fineness. This is due to the fact that the linear equations need not be solved accurately, so only a few inner iterations are necessary. Only for the pressure-correction equation would the implementation of a *multigrid solver* result in a reduction of computing time on fine grids, because it converges more slowly and may require higher accuracy, especially in case of unsteady flows. Any of the above described solvers could be used as a *smoother* within such a multigrid solver.

www.manaraa.com

3 Parallelization Strategy and Efficiency

3.1 Grid Partitioning Technique

Domain decomposition technique is the basis of the parallelization strategy used in the present study. It is completely analogous to the block-structuring of grid in complex geometries. The number of blocks is dictated by the geometry of the solution domain, and the number of CVs within each block may vary substantially. However, we may create more blocks than the geometry requires. The aim is to have as many subdomains of approximately the same size as there are processors, and assigning each subdomain to one processor. If some blocks are much smaller than the others, more than one block may form a subdomain assigned to one processor.

The subdomains do not overlap, i. e. each processor calculates only variable values for CVs within its subdomain. However, each processor uses some variable values which are calculated by other processors to which neighboured subdomains are assigned. This requires, in case of MIMD computers with distributed memory, an overlap of storage: each processor stores data from one or more layers of CVs belonging to neighbour subdomains along its boundary. These data are exchanged between processors, typically after each inner iteration.

The grid may be calculated on one processor and then partitioned into subdomains which are assigned to individual processors; however, in case of large problems we may specify for each processor its subdomain boundaries and generate the grid locally. The grid coordinates at subdomain boundaries must match, since the CV faces are common to two subdomains. In two-dimensional applications the subdomains have a shape of logical rectangles. Ideally, each subdomain has four neighbours; however, in case of complex geometries this may not be always achievable, so one side of one subdomain may be shared with two or more neighbour subdomains. This increases communication overhead for processors assigned to such subdomains. Such poblems will not be considered here.

The solution strategy for block-structured grids will be described by considering a quadrilateral solution domain subdivided into four subdomains (blocks) as shown in Fig. 3. The discretization of the partial differential equation for the variable ϕ leads at each CV to an algebraic equation of the form

$$A_P\phi_P + A_E\phi_E + A_W\phi_W + A_N\phi_N + A_S\phi_S = Q_P \; , \qquad (1)$$

where the indices E, W, N, S and P represent the nodes of the computational molecule, cf. Fig. 1. The coefficients A arise from implicit parts of the convection and diffusion fluxes, and Q is the source term (which includes all explicitly treated terms from the discretized equation). For the whole solution domain there results an equation system which can be written in matrix form as

$$[A]\{\phi\} = \{Q\} \; , \qquad (2)$$

where $\{\phi\}$ is the column matrix containing variable values at CV centers ordered in a certain way, $\{Q\}$ is the corresponding column matrix containing the source

www.manaraa.com

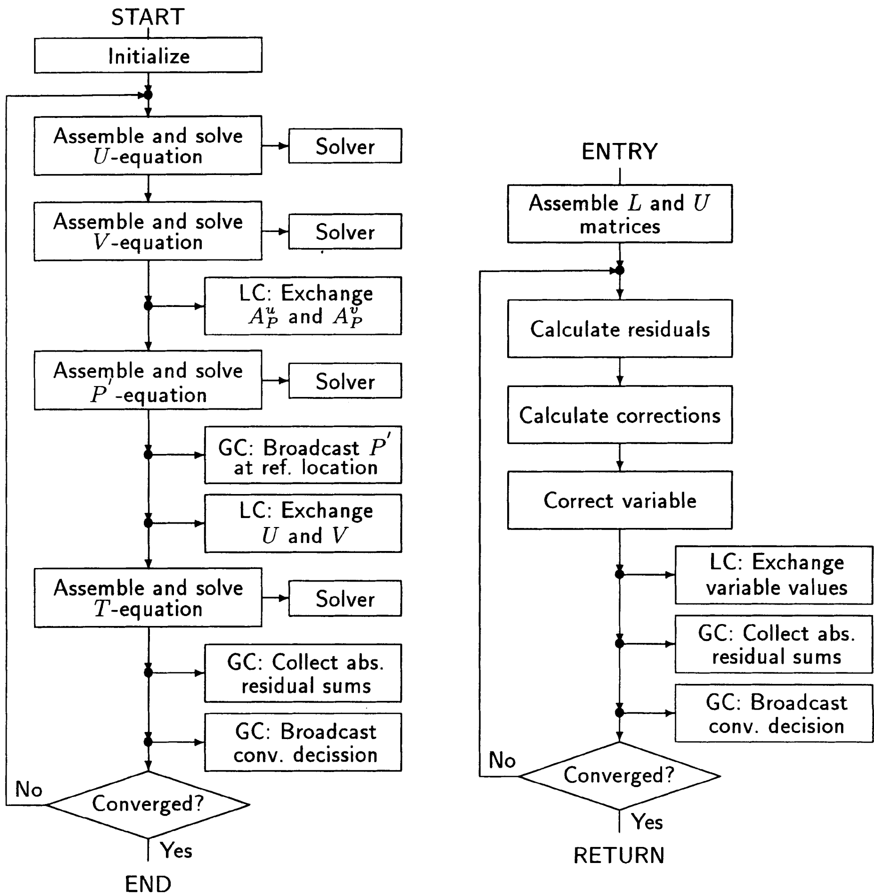


Fig. 2. Flow chart of the outer (left) and the inner (right) iteration loop (SIP solver)

term and $[A]$ is the coefficient matrix. If the nodes are ordered within each block by starting at the southwest corner, proceeding northwards along the first column of CVs and then eastwards to the last column, than in case of a five point computational molecule the coefficient matrix $[A]$ has the structure shown in Fig. 3 for an example with four subdomains. The diagonal block matrices $[A_{ii}]$ are the main matrices of the subdomains and have the same form as the matrix $[A]$ would have if the whole domain was considered as one block. The off-diagonal block matrices describe coupling of subdomains. For example, matrix $[A_{12}]$ of Fig. 3 describes coupling of block 1 with block 2 through the coefficient A_N in CVs next to the north boundary of block 1. The block matrix $[A_{13}]$ in Fig. 3 describes coupling of block 1 with block 3 through the coefficient A_E in CVs along east boundary of block 1. The block matrix $[A_{14}]$ has no non-zero entries, since blocks 1 and 4 do not have common interfaces.

www.manaraa.com

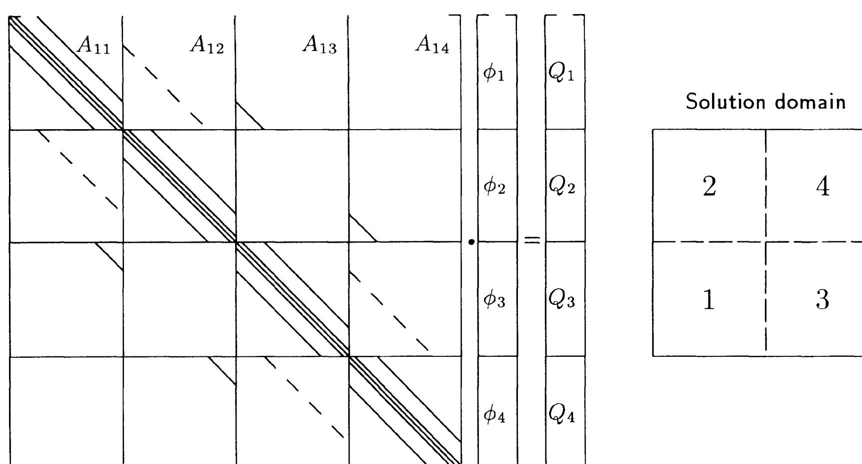


Fig. 3. Matrix structure for a quadrilateral domain subdivided into four subdomains

On a single processor, one could adopt several possible iterative solution strategies. In a parallel environment, the following iteration scheme is the simplest choice:

$$[M_i]\{\phi_i^m\} = \{Q_i\} + [M_i - A_{ii}]\{\phi_i^{m-1}\} - [A_{ij}]\{\phi_j^{m-1}\} \quad , \tag{3}$$

where $[M_i]$ is the iteration matrix in block i, m is the iteration counter and index j defines neighbour blocks ($j \neq i$; summation on j). In case of the GS solver, the matrix $[M_i]$ is the lower triangular matrix of $[A_{ii}]$, whereas in case of the SIP solver, it is the product of some lower and upper triangular matrices $[L_i]$ and $[U_i]$. The iteration procedure need not be the same in each block; for example, one can use GS for small and SIP for large blocks.

This way of iteratively solving the equation systems for the solution domain as a whole is adopted in this study. Each subdomain is treated within one inner iteration as if it were an independent solution domain; reference to nodes inside other subdomains is treated explicitly. The addition of explicit parts due to the coupling matrices shown above usually does not signinficantly slow down the convergence if the variable values along interfaces are updated after each *inner* iteration. This communication option is used in this study.

Another option is to exchange the interface variable values between processors only after each outer iteration. This obviously decouples the subdomains within inner iterations completely and is bound to increase the number of outer iterations for large number of subdomains. In that case, heavier under-relaxation has to be employed to ensure convergence. This communication option is suitable fo systems with small number of processors ans slow communication, like workstation clusters. It is investigated in Schreck and Perić [8] and will not be further considered here.

www.manaraa.com

3.2 Efficiency of Parallel Implementation

The effectiveness of parallel computing can be characterized by the total efficiency, defined as the ratio of computing time on one processor using the most efficient serial algorithm, T_s, and the n-fold computing time using the parallelized algorithm and n processors, T_n:

$$E_n^{tot} = \frac{T_s}{nT_n} = E_n^{par} E_n^{num} E_n^{lb} \ . \tag{4}$$

Schreck and Perić (1991) have shown that the total efficiency can be expressed as a product of three factors termed *parallel* (E_n^{par}) , *numerical* (E_n^{num}) and *load balancing* (E_n^{num}) efficiency. These factors describe: *(i)* the increase of elapsed time for a parallel computation due to communication between processors during which computation can not take place, *(ii)* the increase in the number of floating point operations per grid node required to reach the solution of the same accuracy when the number of subdomains is increased, and *(iii)* idle time of some processors due to uneven load. Communication can be further split into *local* and *global*; the former describes exchange of interface information between neighboured subdomains, and the latter gathering of some information (e. g. level of residuals) from all processors to the "master" and broadcasting of some information (e. g. decision on convergence) from the master to all other processors. The difference is that the local communication runs in parallel, i. e. all processors are – except for the effect of unequal size of interfaces – involved in communication simultaneously. In case of global communication, only a certain number of processors is involved in communication at any time between begin and end of gathering or scattering of information. This is why the global communication is the limiting factor for massive parallelization, unless communication and computation are allowed to take place simultaneously.

One of the major factors affecting the efficiency of parallel flow prediction is the numerical efficiency. Domain decomposition introduces additional decoupling of linear equations for most solvers (exceptions are e.g. Jacobi and "red-black" Gauß–Seidel). This usually leads to an increase in the number of both inner and outer iterations which are required to obtain a solution of prescribed accuracy compared to the calculation on one processor. The effect of domain decomposition on the performance of the multigrid scheme for outer iterations as used in the present solution method is also of crucial importance. We shall investigate these effects through test calculations in the next section.

Schreck and Perić [8] have shown that – for a chosen algorithm and communication pattern – the local and global communication can be expressed as a function of the computer parameters like latency time (μs), communication speed (MB/s), calculation speed (Mflops), grid size and number of processors used. This allows the prediction of parallel efficiency and the optimization by varying the communication modes and patterns. Comparisons of predicted and measured parallel efficiencies for various grid sizes, number of processors and different computers showed good agreement.

www.manaraa.com

As noted before, the parallel efficiency increases substantially if the computation and communication can take place simultaneously. Most new generation parallel computers offer this possibility by using "communication coprocessors". In Fig. 2 the local (LC) and global (GC) communications within the present solution algorithm are indicated. The solution algorithm can be rearranged to allow local communication to take place while doing calculations for the inner region, and performing calculations for CVs along interfaces at the end of each sequence of operations. Only in case of coarse grids (which are always encountered in multigrid methods) may computation have to be halted until communication finishes. Global communication can also be overlayed with computation. For example, collection of residual levels and broadcasting of decision on convergence can be allowed to take time of one whole outer iteration: the convergence decision can be based on the residual level of the previous outer iteration and the extrapolated convergence rate. The level of pressure correction at the reference location may also be taken from the previous iteration. This is possible since at convergence the level of pressure correction will be zero everywhere.

In this study the computation is in principle halted while communication takes place. Although some parallel computers used allow for some kind of overlapping of communication and computation, these options were not used. The results of test calculations presented here thus represent the worst case: the efficiency will be improved when communication and calculation can be overlapped. Local communication is done in pairs of processors: some are sending while others are receiving the data. Since all subdomains were of the same size and there was only one neighbour per interface, the communication pattern is simple and also allows the use of hard-wired communication channels on transputer systems. The gobal communication propagates in one direction like a wavefront, and then in the other direction sequentially from processor to processor. For a square array configuration of processors and given grid size, the local communication time is proportional to $\frac{1}{\sqrt{n}}$ while the global communication is proportional to $\sqrt{n}$, where n is the number of processors. The parallel efficiency depends on the ratio of communicating to computing time; its dependence on n is (cf. Schreck and Perić, [8]):

$$E_n^{par} = \frac{1}{1 + an^{1/2} + bnn^{1/2}} ,$$

where a and b are constants. For a constant number of processors and a varying grid size, the parallel efficiency depends on the grid size as

$$E_n^{par} = \frac{1}{1 + \dfrac{a}{\sqrt{N_n^{cv}}} + \dfrac{b}{N_n^{cv}}} .$$

where N_n^{cv} is the number of CV per processor. These are the limiting factors which show that the global communication effects are the dominant ones. This will be demonstrated by test calculations presented in the next section. The effect of avoiding global communication within inner iterations (or of performing it while computing) will also be demonstrated in the next section.

المنارة للاستشارات
www.manaraa.com

The load balancing effects arise from the fact that in complex geometries the subdomains may not have the same size or the same boundary surface. Furthermore, complex boundary conditions on external boundaries – which exist only for some subdomains – may also cause delays. Algorithms for automatic grid partitioning usually optimize the size and shape of subdomains and the communication pattern, since many parallel computers communicate faster between directly connected processors than between remote ones. However, optimization of load balancing sometimes consumes more computer time than flow simulation, which requires for compromises to be made. In this study blocks of equal size are used, so that a load imbalance may arise only through the effects of boundary conditions on the outside domain boundary.

4 Test Calculations and Analysis of Performance

The test case analysed in this study is the buoyancy driven flow in an inclined cavity. Figure 4 shows predicted streamlines and isotherms. This is one of the benchmark test cases presented by Demirdžić *et al* [2]. All cavity walls are of the same area: the horizontal walls are adiabatic and the inclined walls (angle 45°) are isothermal. The temperature difference, cavity size and fluid properties are chosen such that for Prandtl number $Pr = 0.1$ the Rayleigh number $Ra = 10^6$ is obtained. Calculations were performed on each grid – starting with the coarsest one – until the sum of absolute residuals for each variable is reduced 5 orders of magnitude. Because the number of inner iterations was fixed, each outer iteration involves the same number of computing operations. Therefore, the number of outer iterations needed to reach the converged solution directly reflects the numerical efficiency of the grid partitioning. The increase in computing time per outer iteration is then due to communication only and reflects the parallel efficiency. In order to check the adequacy of the prescribed number of inner iterations, calculation was also done on a workstation where the sum of absolute residuals in the inner iteration loop was required to be reduced one order of magnitude. Table 1 shows the numbers of outer iterations and the computing times for these two cases.

The number of outer iterations and computing times are almost identical for the three finest grids, which means that a good choice has been made for the prescribed number of inner iterations. Since the numerical efficiencies of the two approaches are identical, one can investigate the effect of the global communication in the inner iterations on the total efficiency. To this end calculations were performed on three parallel computers using prescribed number of inner iterations; however, in one set of calculations the global communication (GC) for the checking of convergence of inner iterations was performed (but the convergence criterion was such that it could not be satisfied before the maximum specified number of inner iterations was reached), and in one set it was omitted, cf. Fig. 2.

The three computers used were: Parsytec Supercluster, based on Transputers T805 (30 MHz); Meiko Computing Surface, based on Transputers T800

www.manaraa.com

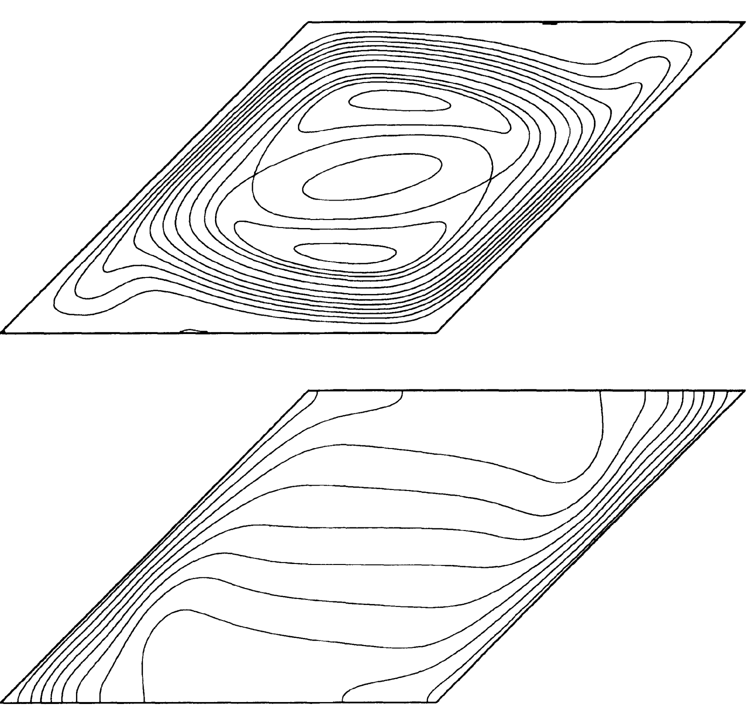


Fig. 4. Streamlines (above) and isotherms (below) for buoyancy driven flow in inclined cavity at $Ra = 10^6$, $Pr = 0.1$

Table 1. Number of outer iterations and computing time (seconds) on a SUN Sparcstation 1+ for calculations with variable and prescribed number of inner iterations

Grid (CV)	Variable No. Inner iter.		Fixed No. Inner iter.		
	No. outer iter.	Comp.-time.	No. outer iter.	Comp.-time	No. inner iter.
10 × 10	200	14.3	181	13.4	5 (u,v,t) 10(p)
20 × 20	129	45.4	135	46.3	5 (u,v,t) 10(p)
40 × 40	87	140.7	86	135.5	4 (u,v,t) 8(p)
80 × 80	46	323.4	46	311.0	4 (u,v,t) 7(p)
160 × 160	25	777.4	25	753.7	4 (u,v,t) 6(p)
320 × 320	22	2493.3	22	2402.6	4 (u,v,t) 6(p)

المنارة للاستشارات

www.manaraa.com

(25 MHz); Intel IPSC/860, based on i860 Processor (40 MHz). The main factors effecting the performance (start-up or latency time t^{st}, data transfer rate R_{tr}, and computing speed τ) are given in Table 2.

Table 2. Characteristic data of computers used in calculations

Computer	τ (Mflops)	$t^{st}(\mu s)$	R_{tr} (MBs)
Parsytec	0.55	84	1.5
Meiko	0.5	20	1.5
Intel	2.5	80	2.8

Communication between processors was performed via hard-wired channels for the Meiko CS, and using Parix library for the Parsytec SC. On the Meiko CS and the IPSC/860 calculations were performed with 5×5 processors, while on the Parsytec SC, 5×5 and 10×10 processors were used. In Tables 3, 4 and 5 numbers of outer iterations and computing times for the different computers and number of processors are presented.

Table 3. Numbers of outer iterations and computing times (in seconds) for calculations on Meiko CS with 5×5 processors

Grid	No. outer iter.	Computing time (Efficiency %)	
		with GC in inner iter.	without GC in inner iter.
10×10	200	8.9 (18)	6.6 (25)
20×20	135	15.1 (26)	12.2 (45)
40×40	94	31.3 (51)	27.6 (58)
80×80	51	55.5 (65)	52.5 (69)
160×160	25	104.6 (84)	101.5 (87)
320×320	22	294.1 (95)	291.0 (96)

Comparison of table 1 with tables 3, 4 and 5 reveals that with 25 processors the number of outer iterations increased only on the third and fourth grid; on the two finest grids, the numerical efficiency is 100%. Furthermore, comparison with table 6 shows that the same is true for 100 processors. Moreover, 100 Processors require less outer iterations on the third grid than 25 Processors! In some cases, the method converges faster with domain decomposition than without, but this is rather an exception. The reason is that the effects of domain decomposition and under-relaxation can complement each other. Usually, the decoupling due to domain decomposition has the same effect on convergence as the reduction of the under-relaxation factor. Depending on the choice of under-relaxation factors

www.manaraa.com

Table 4. Numbers of outer iterations and computing times (in seconds) for calculations on IPSC/860 with 5×5 processors

Grid (CV)	No. outer iter.	Computing time (Efficiency %)	
		with GC in inner iter.	without GC in inner iter.
10×10	200	18.6 (2)	10.3 (3)
20×20	135	23.3 (5)	13.1 (8)
40×40	94	32.5 (10)	19.4 (16)
80×80	51	38.3 (19)	26.2 (28)
160×160	25	47.3 (37)	35.8 (50)
320×320	22	85.0 (65)	73.3 (76)

Table 5. Numbers of outer iterations and computing times (in seconds) for calculations on Parsytec Supercluster with 5×5 processors

Grid (CV)	No. outer iter.	Computing time (Efficiency %)	
		with GC in inner iter.	without GC in inner iter.
10×10	200	18.8 (8)	11.2 (13)
20×20	135	30.0 (16)	17.8 (28)
40×40	94	49.7 (29)	34.0 (42)
80×80	51	71.2 (46)	56.5 (58)
160×160	25	114.7 (69)	101.2 (78)
320×320	22	289.8 (86)	278.0 (90)

the convergence rate may be slowed down through domain decomposition (as is usually the case), but it may for some flows and grids also slightly improve. In any case, the domain decomposition method in conjunction with the multigrid algorithm for outer iterations is numerically very efficient.

The reason for the good performance of FMG in connection with domain decomposition lies in the nature of multigrid methods. On the finest grid, only the high frequency error components are smoothed. These error components are of local character, so nodes near subdomain interfaces require only the information from few CVs across the interface. For this purpose, few inner iterations with an exchange of variable values after each are sufficient. The low frequency errors spread accross the whole solution domain; they are eliminated on the coarsest grids, where the CVs are so large that one exchange of variable values carries information far across the domain. By doing usually few more inner iterations on the coarsest grid, the result matches single block performance.

The parallel efficiencies – which in the considered case are equal to the total efficiencies for the two finest grids – are strongly dependent on communication performance and the ratio of latency time to computing speed of the processor. This effect is best illustrated by comparing efficiencies for calculations with and without global communication (GC) in inner iterations. This global communica-

www.manaraa.com

Table 6. Numbers of outer iterations and computing times (in seconds) for calculations on Parsytec Supercluster with 10×10 processors

Grid (CV)	No. outer iter.	Computing time (Efficiency %)	
		with GC in inner iter.	without GC in inner iter.
10×10	200	34.5 (1)	13.9 (3)
20×20	135	43.4 (3)	18.7 (7)
40×40	88	55.7 (6)	26.4 (14)
80×80	52	65.7 (12)	36.3 (23)
160×160	25	78.6 (25)	53.5 (37)
320×320	22	144.4 (43)	119.7 (52)

tion involves sending one 8 byte word (absolute residual sum) to the master and receiving one word (decision of convergence) from the master. The time required for this communication is – for a given number of processors – independent of grid fineness.

With 25 processors the Meiko CS needs for the global communication within inner iterations about 3.1 s on the last three grids, the Parsytec SC needs about 12 s and the IPSC/860 about 11.5 s. With 100 processors, the Parsytec SC needs about 25 s. This is expected, since in the communication model used, the doubling of the number of processors in each direction means that the longest message path (measured by the number of sendings from one processor to another – diameter of the network) also doubles. The times consumed for the global communication on the three computers are also directly proportional to the latency time, cf. Table 2: Parsytec SC and IPSC/860 need about the same time and the Meiko CS needs four times less time.

In case of the Meiko CS, the avoiding of global communication in inner iterations (or – which would be equivalent– simultaneous communication and calculation) does not result in a signinficant improvement of efficiency. This is due to the fact that the latency time is relatively low compared to the computing time (equivalent to 10 Flop). The effect becomes more important for the Parsytec SC (about 4% longer computing time on the finest grid and 13% on the next coarser one), due to the fact that its latency time is four times longer and its computing performance is somewhat better than that of the Meiko CS. The IPSC/860 suffers most from this effect: it needs 16% more computing time on the finest grid and 32% more on the next coarser grid, if the global communication in the inner iterations is performed. This is due to the fact that the latency time is the same as for the Parsytec SC but the computing performance is five times better.

The global communications for the broadcast of the reference pressure and the convergence check consume less computing time than those in inner iterations (cf. Fig. 2), since these are done only once per outer iteration, whereas within inner iterations there are 4 to 10 global communications for each equation solved, i.e. over 20 per outer iteration. For high processor numbers the global

المنارة للاستشارات
www.manaraa.com

communication clearly becomes the limiting factor of the computations, which is obvious from a comparision of tables 5 and 6: the parallel efficiency drops from 90% to 52% when the number of processors is increased from 25 to 100 (for the finest grid). Simultaneous communication and computation is a way to further improve the efficiency of parallel computing. Special communication coprocessors can be optimized for fast communication, since they need not do computation. Transputers appear to be suitable for this purpose.

5 Conclusions

Results of fluid flow and heat transfer calculations with the parallelized version of an implicit multigrid finite volume solution method and the analysis presented in the preceeding sections allow for the following conclusions to be made:

- The test calculations showed that the numerical efficiency of the multigrid algorithm is close to 100% when domain decomposition is used as the basis for parallelization on MIMD computers, and when interface data are exchanged between processors after each inner iteration.
- It is computationally more efficient to prescribe the number of inner iterations rather then checking convergence for each inner iteration. The possible increase in the number of outer iterations is more than compensated by the reduction in the number of inner iterations and by avoiding much of the global communication. This is especially true for computers with relatively high latency time compared to computing speed.
- Efficiency of parallel computation can be signifcantly improved by overlaying communication and computation. The present solution algorithm allows this to a large degree and the new generation parallel computers offer that possibility.

It is therefore the most essential matter that the numerical method retains its effectiveness when parallelized, i.e. to achieve high numerical efficiency. The efficiencies are expected to be higher for three-dimensional applications, since the number of floating point operations per CV and iteration is much higher than in the two-dimensional case, so that the effect of latency diminishes. It is thus obvious that parallel computing is very suitable for computational fluid dynamics. Implicit finite volume solution methods are adaptable for efficient application on all parallel systems, from workstation clusters to massively parallel computers.

6 Acknowledgements

The Comission of the European Communities sponsored via "Parallel Computing Action" a part of the Meiko Computing Surface used in this study. The institute for applied mathematics of the research center Jülich and the institute for computer science of Technical University Munich provided access to IPSC/860 computer. The authors thank for this support.

www.manaraa.com

References

1. P. Bastian and G. Horton: "Parallelization of robust multi-grid methods: ILU factorization and frequency decomposition method", in W. Hackbusch and R. Rannacher (eds.), *Notes on Numerical Fluid Mechanics, Vol. 30*, Vieweg, Braunschweig, 1989, pp. 24-36.

2. I. Demirdžić and M. Perić:"Finite volume method for prediction of fluid flow in arbitrarily shaped domains with moving boundaries", *Int. J. Num. Methods in Fluids*, **10**, 771-790 (1990).

3. I. Demirdžić, Ž. Lilek and M. Perić: "Fluid flow and heat transfer test problems for non-orthogonal grids: benchmark solutions", *Int. J. Num. Methods in Fluids*, **15**, 329-354 (1992).

4. M. Hestens and E. Stiefel: "Methods of conjugate gradients for solving linear systems", *Nat. Bur. Standards J. Res.*, **49**, 409-436 (1952).

5. M. Hortmann, M. Perić and G. Scheurer: "Finite volume multigrid prediction of laminar natural convection: bench-mark solutions", *Int. J. Numer. Methods Fluids*, **11**, 189-207 (1990).

6. S. V. Patankar and D. B. Spalding: "A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows", *Int. J. Heat Mass Transfer*, **15**, 1787-1806 (1972).

7. M. Perić, M. Schäfer and E. Schreck: "Computation of fluid flow with a parallel multi-grid solver", in K.G. Reinsch *et al.* (Eds.), *Proc. Int. Conference on "Parallel Computational Fluid Dynamics"*, Elsevier, Amsterdam, 1991.

8. E. Schreck and M. Perić: "Computation of fluid flow with a parallel multi-grid solver", *Int. J. Num. Methods in Fluids*, 16, 303 - 327 (1993).

9. H.L. Stone: "Iterative solution of implicit approximations of multi-dimensional partional differential equations", *SIAM J. Numer. Anal.*, **5**, 530-558 (1968).

www.manaraa.com

Index

ab initio 266
actions 192
adaptive partitioning 246, 254
address hashing 102, 103
agents 196
algorithm 305
aliasing 173
AND-problems 50
application concept 25
approximate state space method 84
approximation method 80
assignment 166
automatic parallelization 118
automatic test pattern generation 234
bandwidth 1, 56
barrier synchronization 106, 107, 109, 111, 112, 116
basis functions 267
beamsplitter 7
benefit functions 53
benefit rate 55
block-structured grids 292, 295
block-structuring 295
bounding methods 80
broadcast 286
bulk-synchronous parallelism (BSP) 106, 116
cache coherence 103, 133
candidates 92
capsules 170
causality 134
central control management 239
centralized components 89
checker 33
checkpointing 43
CHORUS 90
class 181, 186
cluster 172
coarse 215
coarse grain parallelism 276
collocation 183, 184
collocation association 170
combination technique 276
combinational circuits 235
combinatorial optimization 219
combining network 112, 116
communication channels 196
communication memory 17
communication memory interface 18
communication system 191
completion times 98
components be located 94
concurrency control 133
concurrent file system 258
connectionist semantic networks 216
constant factors 102, 103, 115
control 211
control algorithms 204, 207
control flow checking 35
control loop 90
control volume 293
cooperation class 175
cooperations 174
coordination 22
correctness 200
cost rate 55
coupling unit 18, 20
CPU 91
CPU-utilization 52
creation of new processes 88
crosstalk 12
crystal orbital 266
data skew 246, 257
data-flow analysis 120
decision tree 237
decision units 89
design specification 190
diagonalization 273
dislocation association 170
dislocation 183, 187
distributed memory 87, 133, 279
distributed memory multiprocessors 246
distributed object model 181

www.manaraa.com

distributed self-management 238
distributed shared memory (DSM) 15, 102, 116, 133, 159
distributed snapshots 43
distributed systems 190
distributed user programs 25
distributed-memory machine 230
distribution 166
distribution associations 170
distribution groups 172
distribution language 166
distribution model 169
distribution system 167
distribution transparency 168
domain decomposition 298, 302, 303
domain decomposition technique 295
dynamic load balancing 241
dynamic load managment 88
effective scalability 250
efficiency 62, 102, 103, 105, 106, 108, 109, 110, 111, 112, 113, 115, 270, 298
eigenvalue problem 266
elliptic partial differential equations 276
environment components 192
error detection 33
evalution unit 90
event recorder 71
event-driven monitoring 66
excess parallelism 105, 106, 109, 111, 112, 113, 115
external sorting 246
farming 268
fault dependency 240
fault handling 31
fault parallelism 234
fault simulator 238
fault tolerance 31
fault tolerant interconnection network 38
fault treatment 43
fetch&op 116
fiber concentrator 8
final placement 219
fine-grained parallelism 118
Finite State Machine (FSM) 235
finite volume 293
fluid flow 292, 305
formal method 190
frequency of synachronization 98
functional program 191
gang scheduling 26
gate 173
Gau"s-Seidel 292, 293
Gaussian functions 267
global communication 298, 299, 300, 303, 304
global instruction scheduling 118
global resource scheduling 88
global synchronization 95, 96
grand challenges 31
granularity 99
graph models 80
graph search 211
hard faults 44
hardware monitoring 67
Hartree-Fock 266
hashing 104, 105, 106, 107
heuristics 95, 96, 98
holographic permutation elements 9
horizontal parallelism 247
horizontal parallelization 246
hybrid monitoring 67
HyperCube 241
idle-time 89, 91, 93, 95, 96, 98, 99
ILU method 292
ILU solver 293
ILU-decomposition 293
image analysis and understanding 203
imaging 6
immutable objects 185
implementation 190
implicit finite volume solution methods 305
implicit methods 292
indifference association 170
inefficiencies 103, 109, 110, 113, 115
inheritance 187

www.manaraa.com

inner iteration 292, 293, 294, 295, 297, 299, 300, 303, 304, 305
instruction scheduling 119
Intel 270
interconnection network 17, 19
interconnection topologies 4
interface 73
interprocedural 120
interrupt mechanism 26
iPSC/2 258
island 173
iteration matrix 297
knowledge-based image understanding 203
large data volumes 246
latency hiding 103, 106, 113
latency reduction 103
latency time 303, 304, 305
leader task 26
lengths of the ready queues 95
liveness properties 193
load balancing 87, 88, 279, 298, 300,
load balancing effects 300
load evalution unit 92, 94
load imbalance 87, 92, 95, 98
load management 87
load managements testbed 93
load measure 92
load parameters 87, 89
load sharing 87, 88
local communication 298, 299
local components 89
local global communication 298, 299
MACH 90
mapping 87, 96, 98
massive parallelism 102, 103, 111, 116
massively parallel computers 31
master 33
master application 26
master-checker mode 33
McCulloch Pitts neuron 92
mean runtime 80
measurement unit 90
medium 215
memory access analysis 121
memory access latencies 102, 103, 105, 115
memory management 150
memory randomization 103, 104
MEMSOS 24
MEMSY 15, 75, 274
message-passing 27
meta-abstraction 152
meta-layer 152, 160
meta-object 152
meta-recursion 153
method 305
migration candidate 93, 95
migration of data 89
migration of running processes 90
migration unit 90, 92
MMK 247
mobility 169
monitoring system 90
move_multicycle_op 126
multigrid algorithm 305
multigrid finite volume 305
multigrid method 294, 303
multiple beam splitter 8
multiprocessor 87
multiprogramming control 49
multistage networks 9
network of workstations 243, 276
neutral networks and linear optimization 211
non-orthogonal 293
non-series-parallel graphs 83
numerical 298
numerical efficiencies 298, 300, 302, 305
numerical functions 81
object 181
object cache 158
object management 150
object space 156
object store 155, 160
object-migration 159
object-oriented operating system 150
OID 183
open operating system 150, 161

www.manaraa.com

optical backplane 4
optical bus 7
optimization 212
optimization problem 204
opto-ASIC 10
OR-problem 50
outer iteration 292, 293, 294, 297, 299, 300, 302, 303, 304, 305
page fault 93
parallel 298
parallel checkpoint coordination 46
parallel communication 285
parallel efficiency 298, 299, 300, 303, 305
parallel programs 80
Parallel Random Access Machine (PRAM) 104
parallel slackness 105
parallel solution 276
parallel sorting 246
parallelism 3
parallelization 215
partial order semantics 139
partitioning 219
PEPP 80
percolation scheduling 120
performance 103, 106, 108
performance criteria 94
performance evaluation 80, 106
persistency 41
persistent object 150, 154
Petri nets 139
power of a parallel system 63
PRAM 105, 109
PRAM emulation 104, 106
preprocessor 35
process migration 87, 93, 99
process switching 106, 107, 109, 110, 112, 113, 114, 115, 116
processor nodes 17
profit rate 55
program analysis 118
program graph 124
programming model 22
protection 42
protocol development 191
protocol hierarchy 195
quantum mechanical methods 265
Randomized Shared Memory (RSM) 102, 103, 115, 116
ready queue length 91, 96, 98, 99
real-time computer vision 203
redundancy 31
refinement 195
reflection 153, 162
relative distribution 170
requirements specification 190
response time 55, 88
rollback-recovery 43
runtime distribution 80
safety properties 193
sampling 254
scalability 31, 102
SCF 266
semantic network 203, 204, 205
sequential circuits 234
series-parallel 83
service time 50
shape functions 219
shared disk multiprocessor architecture 246
shared disk multiprocessors 249
shared memory 23, 28, 102, 105, 115, 159,
shared-memory machine 229
sharing of nodes 87
signatures 35
SIMPLE algorithm 293
simulation 102, 103, 106
single user / single program 88
slicing tree 220
smart pixels 10
soft faults 44
software monitoring 67
sparse grid 276, 277
specification 137
specification formalism 197
speed-up 49, 270
speedup characteristic 62
stable storage 41

www.manaraa.com

static load management 88
stenning protocol 195
stream-processing functions 190
superlinear speedups 51
superstep 105, 106
SUPRENUM 270
symbolic processing 204
synchronisation 12
synchronization frequencies 95
synchronization tree 112
system call for monitoring 77
system components 192
system development 190
target faults 237
tasks 25
test pattern 234
test pattern generation 234
test problem 235
test sequence 234
theorem prover 50
threshold values 95, 96, 99
throughput 49
time slice simulation 60
TOPSYS 247
total efficiency 298
traces 107, 108
trace specifications 190
TransBase 247
transputer 107
two-electron integrals 267
two-level rollback 44
two-phase commit protocol 44
type 182
type checking 188
uniform object model 180, 181
universal hashing 106
variable 181, 185
vector computers 276
verification 133
VLIW 125
VLIW-Scheduler 129
VLSI Layout 219
watchdog processor 35
weak coherence 135
workstation network 228
ZM4 68

المنارة للاستشارات
www.manaraa.com

Springer-Verlag and the Environment

We at Springer-Verlag firmly believe that an international science publisher has a special obligation to the environment, and our corporate policies consistently reflect this conviction.

We also expect our business partners – paper mills, printers, packaging manufacturers, etc. – to commit themselves to using environmentally friendly materials and production processes.

The paper in this book is made from low- or no-chlorine pulp and is acid free, in conformance with international standards for paper permanency.

المنارة للاستشارات

www.manaraa.com

Lecture Notes in Computer Science

For information about Vols. 1–660
please contact your bookseller or Springer-Verlag

Vol. 661: S. J. Hanson, W. Remmele, R. L. Rivest (Eds.), Machine Learning: From Theory to Applications. VIII, 271 pages. 1993.

Vol. 662: M. Nitzberg, D. Mumford, T. Shiota, Filtering, Segmentation and Depth. VIII, 143 pages. 1993.

Vol. 663: G. v. Bochmann, D. K. Probst (Eds.), Computer Aided Verification. Proceedings, 1992. IX, 422 pages. 1993.

Vol. 664: M. Bezem, J. F. Groote (Eds.), Typed Lambda Calculi and Applications. Proceedings, 1993. VIII, 433 pages. 1993.

Vol. 665: P. Enjalbert, A. Finkel, K. W. Wagner (Eds.), STACS 93. Proceedings, 1993. XIV, 724 pages. 1993.

Vol. 666: J. W. de Bakker, W.-P. de Roever, G. Rozenberg (Eds.), Semantics: Foundations and Applications. Proceedings, 1992. VIII, 659 pages. 1993.

Vol. 667: P. B. Brazdil (Ed.), Machine Learning: ECML – 93. Proceedings, 1993. XII, 471 pages. 1993. (Subseries LNAI).

Vol. 668: M.-C. Gaudel, J.-P. Jouannaud (Eds.), TAPSOFT '93: Theory and Practice of Software Development. Proceedings, 1993. XII, 762 pages. 1993.

Vol. 669: R. S. Bird, C. C. Morgan, J. C. P. Woodcock (Eds.), Mathematics of Program Construction. Proceedings, 1992. VIII, 378 pages. 1993.

Vol. 670: J. C. P. Woodcock, P. G. Larsen (Eds.), FME '93: Industrial-Strength Formal Methods. Proceedings, 1993. XI, 689 pages. 1993.

Vol. 671: H. J. Ohlbach (Ed.), GWAI-92: Advances in Artificial Intelligence. Proceedings, 1992. XI, 397 pages. 1993. (Subseries LNAI).

Vol. 672: A. Barak, S. Guday, R. G. Wheeler, The MOSIX Distributed Operating System. X, 221 pages. 1993.

Vol. 673: G. Cohen, T. Mora, O. Moreno (Eds.), Applied Algebra, Algebraic Algorithms and Error-Correcting Codes. Proceedings, 1993. X, 355 pages 1993.

Vol. 674: G. Rozenberg (Ed.), Advances in Petri Nets 1993. VII, 457 pages. 1993.

Vol. 675: A. Mulkers, Live Data Structures in Logic Programs. VIII, 220 pages. 1993.

Vol. 676: Th. H. Reiss, Recognizing Planar Objects Using Invariant Image Features. X, 180 pages. 1993.

Vol. 677: H. Abdulrab, J.-P. Pécuchet (Eds.), Word Equations and Related Topics. Proceedings, 1991. VII, 214 pages. 1993.

Vol. 678: F. Meyer auf der Heide, B. Monien, A. L. Rosenberg (Eds.), Parallel Architectures and Their Efficient Use. Proceedings, 1992. XII, 227 pages. 1993.

Vol. 679: C. Fermüller, A. Leitsch, T. Tammet, N. Zamov, Resolution Methods for the Decision Problem. VIII, 205 pages. 1993. (Subseries LNAI).

Vol. 680: B. Hoffmann, B. Krieg-Brückner (Eds.), Program Development by Specification and Transformation. XV, 623 pages. 1993.

Vol. 681: H. Wansing, The Logic of Information Structures. IX, 163 pages. 1993. (Subseries LNAI).

Vol. 682: B. Bouchon-Meunier, L. Valverde, R. R. Yager (Eds.), IPMU '92 – Advanced Methods in Artificial Intelligence. Proceedings, 1992. IX, 367 pages. 1993.

Vol. 683: G.J. Milne, L. Pierre (Eds.), Correct Hardware Design and Verification Methods. Proceedings, 1993. VIII, 270 Pages. 1993.

Vol. 684: A. Apostolico, M. Crochemore, Z. Galil, U. Manber (Eds.), Combinatorial Pattern Matching. Proceedings, 1993. VIII, 265 pages. 1993.

Vol. 685: C. Rolland, F. Bodart, C. Cauvet (Eds.), Advanced Information Systems Engineering. Proceedings, 1993. XI, 650 pages. 1993.

Vol. 686: J. Mira, J. Cabestany, A. Prieto (Eds.), New Trends in Neural Computation. Proceedings, 1993. XVII, 746 pages. 1993.

Vol. 687: H. H. Barrett, A. F. Gmitro (Eds.), Information Processing in Medical Imaging. Proceedings, 1993. XVI, 567 pages. 1993.

Vol. 688: M. Gauthier (Ed.), Ada-Europe '93. Proceedings, 1993. VIII, 353 pages. 1993.

Vol. 689: J. Komorowski, Z. W. Ras (Eds.), Methodologies for Intelligent Systems. Proceedings, 1993. XI, 653 pages. 1993. (Subseries LNAI).

Vol. 690: C. Kirchner (Ed.), Rewriting Techniques and Applications. Proceedings, 1993. XI, 488 pages. 1993.

Vol. 691: M. Ajmone Marsan (Ed.), Application and Theory of Petri Nets 1993. Proceedings, 1993. IX, 591 pages. 1993.

Vol. 692: D. Abel, B.C. Ooi (Eds.), Advances in Spatial Databases. Proceedings, 1993. XIII, 529 pages. 1993.

Vol. 693: P. E. Lauer (Ed.), Functional Programming, Concurrency, Simulation and Automated Reasoning. Proceedings, 1991/1992. XI, 398 pages. 1993.

Vol. 694: A. Bode, M. Reeve, G. Wolf (Eds.), PARLE '93. Parallel Architectures and Languages Europe. Proceedings, 1993. XVII, 770 pages. 1993.

Vol. 695: E. P. Klement, W. Slany (Eds.), Fuzzy Logic in Artificial Intelligence. Proceedings, 1993. VIII, 192 pages. 1993. (Subseries LNAI).

Vol. 696: M. Worboys, A. F. Grundy (Eds.), Advances in Databases. Proceedings, 1993. X, 276 pages. 1993.

www.manaraa.com

Vol. 697: C. Courcoubetis (Ed.), Computer Aided Verification. Proceedings, 1993. IX, 504 pages. 1993.

Vol. 698: A. Voronkov (Ed.), Logic Programming and Automated Reasoning. Proceedings, 1993. XIII, 386 pages. 1993. (Subseries LNAI).

Vol. 699: G. W. Mineau, B. Moulin, J. F. Sowa (Eds.), Conceptual Graphs for Knowledge Representation. Proceedings, 1993. IX, 451 pages. 1993. (Subseries LNAI).

Vol. 700: A. Lingas, R. Karlsson, S. Carlsson (Eds.), Automata, Languages and Programming. Proceedings, 1993. XII, 697 pages. 1993.

Vol. 701: P. Atzeni (Ed.), LOGIDATA+: Deductive Databases with Complex Objects. VIII, 273 pages. 1993.

Vol. 702: E. Börger, G. Jäger, H. Kleine Büning, S. Martini, M. M. Richter (Eds.), Computer Science Logic. Proceedings, 1992. VIII, 439 pages. 1993.

Vol. 703: M. de Berg, Ray Shooting, Depth Orders and Hidden Surface Removal. X, 201 pages. 1993.

Vol. 704: F. N. Paulisch, The Design of an Extendible Graph Editor. XV, 184 pages. 1993.

Vol. 705: H. Grünbacher, R. W. Hartenstein (Eds.), Field-Programmable Gate Arrays. Proceedings, 1992. VIII, 218 pages. 1993.

Vol. 706: H. D. Rombach, V. R. Basili, R. W. Selby (Eds.), Experimental Software Engineering Issues. Proceedings, 1992. XVIII, 261 pages. 1993.

Vol. 707: O. M. Nierstrasz (Ed.), ECOOP '93 – Object-Oriented Programming. Proceedings, 1993. XI, 531 pages. 1993.

Vol. 708: C. Laugier (Ed.), Geometric Reasoning for Perception and Action. Proceedings, 1991. VIII, 281 pages. 1993.

Vol. 709: F. Dehne, J.-R. Sack, N. Santoro, S. Whitesides (Eds.), Algorithms and Data Structures. Proceedings, 1993. XII, 634 pages. 1993.

Vol. 710: Z. Ésik (Ed.), Fundamentals of Computation Theory. Proceedings, 1993. IX, 471 pages. 1993.

Vol. 711: A. M. Borzyszkowski, S. Sokołowski (Eds.), Mathematical Foundations of Computer Science 1993. Proceedings, 1993. XIII, 782 pages. 1993.

Vol. 712: P. V. Rangan (Ed.), Network and Operating System Support for Digital Audio and Video. Proceedings, 1992. X, 416 pages. 1993.

Vol. 713: G. Gottlob, A. Leitsch, D. Mundici (Eds.), Computational Logic and Proof Theory. Proceedings, 1993. XI, 348 pages. 1993.

Vol. 714: M. Bruynooghe, J. Penjam (Eds.), Programming Language Implementation and Logic Programming. Proceedings, 1993. XI, 421 pages. 1993.

Vol. 715: E. Best (Ed.), CONCUR'93. Proceedings, 1993. IX, 541 pages. 1993.

Vol. 716: A. U. Frank, I. Campari (Eds.), Spatial Information Theory. Proceedings, 1993. XI, 478 pages. 1993.

Vol. 717: I. Sommerville, M. Paul (Eds.), Software Engineering – ESEC '93. Proceedings, 1993. XII, 516 pages. 1993.

Vol. 718: J. Seberry, Y. Zheng (Eds.), Advances in Cryptology – AUSCRYPT '92. Proceedings, 1992. XIII, 543 pages. 1993.

Vol. 719: D. Chetverikov, W.G. Kropatsch (Eds.), Computer Analysis of Images and Patterns. Proceedings, 1993. XVI, 857 pages. 1993.

Vol. 720: V.Mařík, J. Lažanský, R.R. Wagner (Eds.), Database and Expert Systems Applications. Proceedings, 1993. XV, 768 pages. 1993.

Vol. 721: J. Fitch (Ed.), Design and Implementation of Symbolic Computation Systems. Proceedings, 1992. VIII, 215 pages. 1993.

Vol. 722: A. Miola (Ed.), Design and Implementation of Symbolic Computation Systems. Proceedings, 1993. XII, 384 pages. 1993.

Vol. 723: N. Aussenac, G. Boy, B. Gaines, M. Linster, J.-G. Ganascia, Y. Kodratoff (Eds.), Knowledge Acquisition for Knowledge-Based Systems. Proceedings, 1993. XIII, 446 pages. 1993. (Subseries LNAI).

Vol. 724: P. Cousot, M. Falaschi, G. Filè, A. Rauzy (Eds.), Static Analysis. Proceedings, 1993. IX, 283 pages. 1993.

Vol. 725: A. Schiper (Ed.), Distributed Algorithms. Proceedings, 1993. VIII, 325 pages. 1993.

Vol. 726: T. Lengauer (Ed.), Algorithms – ESA '93. Proceedings, 1993. IX, 419 pages. 1993

Vol. 727: M. Filgueiras, L. Damas (Eds.), Progress in Artificial Intelligence. Proceedings, 1993. X, 362 pages. 1993. (Subseries LNAI).

Vol. 728: P. Torasso (Ed.), Advances in Artificial Intelligence. Proceedings, 1993. XI, 336 pages. 1993. (Subseries LNAI).

Vol. 729: L. Donatiello, R. Nelson (Eds.), Performance Evaluation of Computer and Communication Systems. Proceedings, 1993. VIII, 675 pages. 1993.

Vol. 730: D. B. Lomet (Ed.), Foundations of Data Organization and Algorithms. Proceedings, 1993. XII, 412 pages. 1993.

Vol. 731: A. Schill (Ed.), DCE – The OSF Distributed Computing Environment. Proceedings, 1993. VIII, 285 pages. 1993.

Vol. 732: A. Bode, M. Dal Cin (Eds.), Parallel Computer Architectures. IX, 311 pages. 1993.

Vol. 733: Th. Grechenig, M. Tscheligi (Eds.), Human Computer Interaction. Proceedings, 1993. XIV, 450 pages. 1993.

Vol. 734: J. Volkert (ed.), Parallel Computation. Proceedings, 1993. VIII, 248 pages. 1993.

Vol. 735: D. Bjørner, M. Broy, I. V. Pottosin (Eds.), Formal Methods in Programming and Their Applications. Proceedings, 1993. IX, 434 pages. 1993.

Vol. 736: R. L. Grossman, A. Nerode, A. P. Ravn, H. Rischel (Eds.), Hybrid Systems. VIII, 474 pages. 1993.

Vol. 737: J. Calmet, J. A. Campbell (Eds.), Artificial Intelligence and Symbolic Mathematical Computing. Proceedings, 1992. VIII, 305 pages. 1993.

www.manaraa.com