NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

Table of Contents

Workshop on Performance Efficient Parallel Programming
Seven Springs, Pennsylvania
September 8-10, 1986

Call For Papers

. . . . page 3

Workshop Participants

. . . . page 5

Position Papers

Fran Berman, University of California at San Diego . . . . .page 13

David L. Black, Carnegie-Mellon University . . . . page 15

Jordan Brower, University of Washington .page 17

J.C. Browne, University of Texas at Austin . . . . page 19

Ingrid Y. Bucher, Los Alamos National Laboratory . . . . page 21

David E. Culler, Massachusetts Institute of Technology . . . . .page 25

Janice E. Cuny, University of Massachusetts at Amherst . . . . page 35

John Feo, Lawrence Livermore National Laboratories . . . . page 37

Jeanne Ferrante, IBM . . . . page 39

Raphael Finkel, University of Wisconsin . . . . page 41

Charles L. Forgy, Carnegie-Mellon University . . . . page 45

Leah H. Jamieson, Purdue University . . . . page 47

H.T. Kung, Carnegie-Mellon University .page 49

Ronald L. Larsen, University of Maryland . . . . page 51

Edward D. Lazowska, University of Washington . . . . page 105

Jerrold Leichter, Yale University . . . . .page 53

Joanne L. Martin, IBM T.J. Watson Research Center . . . . page 55

Heinz Mühlenbein, GMD . . . . page 57

David M. Nicol, NASA Langley Research Center . . . . page 61

David Notkin, University of Washington . . . . page 63

Terrence W. Pratt, University of Virginia . . . . page 65

Tom Reinhardt, Massachusetts Institute of Technology . . . . page 67

David P. Rodgers, Sequent Computer Systems . . . . page 71

Larry Rudolph, Hebrew University . . . . page 75

Vijay Saraswat, Carnegie-Mellon University . . . . page 77

Karsten Schwan, Ohio State University . . . . page 79

Herb Schwetman, MCC . . . . page 81

Zary Segall, Carnegie-Mellon University . . . . page 83

H.J. Siegel, Purdue University . . . . page 85

Lawrence Snyder, University of Washington . . . . page 87

Thomas L. Sterling, Harris Corporation . . . . page 89

Salvatore J. Stolfo, Columbia University . . . . page 93

Michael Stumm, Stanford University . . . . page 97

Bob Thomas, BBN Advanced Computers . . . . page 101

Dalibor Vrsalovic, Carnegie-Mellon University . . . . page 103

John Zahorjan, University of Washington . . . . page 105

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213

Call for Participation

Workshop on Performance Efficient Parallel Programming
Seven Springs, Pennsylvania
September 8-10, 1986

In 1983, a "Workshop on Multiprocessors for High Performance Parallel Computation" was held at Seven Springs, Pennsylvania. This workshop has been devoted to a number of computer engineering research issues in the areas of designing, implementing, programming and performance evaluating multiprocessor systems employed for speeding up some classes of applications. The workshop has been successful in devising the research directions and strategies, and bringing together researchers from university and industry interested in the above topics.

At this time, a specialized workshop devoted to aspects of parallel programming technology and challenges seems appropriate. This workshop will investigate and formulate the main research directions in the areas of computer engineering related to the design and implementation of correct and performance efficient parallel programs, as well as the programming technology required to support the task of programming in the context of new parallel computer architectures.

The technology limitations in producing faster sequential machines together with the ever increasing need for computer cycles in solving numerical, symbolic and real-time problems places the burden for substantial advance in computer performance on new computational models. Parallel processing has been long heralded as a method to build computers executing an extraordinary number of instructions per unit of time. Today, commercial parallel processors have become a reality. The main challenge is whether these machines can be programmed to make effective use of the increased computer power. Hence, the opportunity for parallel processing cannot be successfully exploited without developing the basic parallel programming technology. The diversity of architectures and the wide variation in their underlying computational models makes it particularly hard to find general techniques for developing efficient parallel programs and general guidelines for choosing the appropriate machine for a set of applications.

Accordingly, this workshop will emphasize the parallel programming technology required to apply parallel solutions to problems with the objective of improving execution speed. We are considering the following research issues as integral parts of this thrust:

- Performance efficient mapping of parallel algorithms into parallel architectures and parallel programs
- Application-oriented parallel programming (for numerical, symbolic, real-time and integrated applications)

- Languages to efficiently support new parallel architectures
- Parallel language-oriented architectures (highly programmable parallel processors)
- Programming environments for performance efficient parallel programming
- Compiler techniques for performance efficient parallel programming
- Performance debugging
- Performance contrasting of two hardware/software parallel architectures in an application independent way

In conclusion, we foresee this workshop to play a key role in bringing together the concerns of designers, researchers and users of parallel processors in the area of parallel programming. In addition, we hope to draw the main research directions in this area focussing the scope of research in this field of computer engineering.

List of Participants

NSF-Sponsored Workshop on Performance Efficient Parallel Programming
Seven Springs, Champion, PA 15622

Marco Annaratone
Dept. of Computer Science
Carnegie-Mellon Univ.
Pittsburgh, PA 15213
(412) 268-3049
mxa@vlsi.cs.cmu.edu

Fran Berman
Department of EE & Computer Science, Mail Code C-014
University of California at San Diego
La Jolla, CA 92093
(619) 452-6195
berman@ucsd

David Black
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213
(412)268-7555
black@a.cs.cmu.edu

Beth Bottos
Dept. of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213
(412) 268-7694
bab@g.cs.cmu.edu

Jordan Brower
Department of Computer Science, FR-35
University of Washington
Seattle, WA 98195
jordan@uw-beaver.arpa
jordan@uw-bluechip.arpa

Ingrid Y. Bucher
C-3, MS B265
Los Alamos National Laboratory
Los Alamos, NM 87545
(505)667-2830/7028

Larry Carter
IBM T.J. Watson Research Lab
P.O. Box 218
Yorktown Heights, NY 10598

Bernie Chern
DMCE Division, Room 640
National Science Foundation
1800 G Street, N.W.
Washington, D.C. 20550
(202) 357-7373
chern@a.isi.edu

David E. Culler
Laboratory for Computer Science
Massachusetts Institute of Technology
545 Technology Square, Room 254
Cambridge, MA 02139
(617) 253-8854
culler@mit.xx.arpa

Janice E. Cuny
Computer and Information Science [COINS, GRES]
Lederle Graduate Research Center
University of Massachusetts at Amherst
Amherst, MA 01003
(413)545-4228

John Feo
Lawrence Livermore Nat'l Laboratories (L-419)
P.O. Box 808
Livermore, CA 94550
(415)423-9832
feo@LLL-CRG.ARPA

Jeanne Ferrante
IBM Hawthorne H2-B54
P.O. Box 218
Yorktown Heights, NY 10598
(914)789-7529

Raphael Finkel
On leave from U. of Wisconsin at:
Computer Science Dept.
Patterson Office Tower
University of Kentucky
Lexington, KY 40506
(606)257-6743
raphael%f.ms.uky.csnet@csnet-relay.arpa

Lanny Forgy
Dept. of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213
(412)268-3725

Robert Grafton
DCIE, Room 640
National Science Foundation
1800 G St., N.W.
Washington, D.C. 20550
(202)357-7853.

George Hetrick
(DEC employee at MCC)
c/o MCC
9430 Research Blvd.
Echelon Bldg # 1, Suite 200
Austin, TX 78759
(512)834-3411

Charles Holland
Office of Naval Research, Code 1133
800 N. Quincy St.
Arlington, VA 22217-5000

Leah H. Jamieson
School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907
lhj@ee.ecn.purdue.edu
(317)494-3653

H.T. Kung
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213
(412)268-2568
ktn@n.sp.cs.cmu.edu

Ronald Larsen
Dept. of Computer Science
University of Maryland
College Park, MD 20740
larsen@umdc

Ed Lazowska
Dept. of Computer Science, FR-35
University of Washington
Seattle, WA 98195
lazowska@uw-krakatoa.arpa
(206)543-4755

Ted Lehr
Department of Electrical and Computer
Engineering
Carnegie-Mellon University
Pittsburgh, PA 15213
(412)268-6645
lehr@faraday.ece.cmu.edu

Jerry Leichter
Department of Computer Science
Box 2158
Yale Station
Yale University
New Haven, CT 06520-2158
leichter-jerry@yale.arpa

Joanne L. Martin
IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
(914) 789-7508

David Mizell
Information Sciences Institute
USC
4676 Admiralty Way
Marina del Rey, CA 90291
(213) 822-1511
mizell@isi

Heinz Muhlenbein
GMD
Postfach 1240
Scholss Birlinghoven
D-5205 St. Augustin 1
West Germany
02241/142366

David Nicol
ICASE Mail Stop 132C
NASA Langley Research Center
Hampton, VA 23665
dmn@icase.arpa

David Notkin
Department of Computer Science
University of Washington
Seattle, WA 98195
notkin@uw-timor.arpa
notkin@ward.cs.washington.edu
(206)545-3798

Terrence W. Pratt
Department of Computer Science
Thornton Hall
University of Virginia
Charlottesville, VA 22903
(804)924-1043
"Terrence W. Pratt"@csnet-relay.arpa

Tom Reinhardt
Artificial Intelligence Labs
Massachusetts Institute of Technology
545 Technology Sq.
Cambridge, MA 02139
(617)253-5871
reinhardt@XX.LCS.MIT.EDU

David P. Rodgers
Sequent Computer Systems, Inc.
15450 S.W. Koll Parkway
Beaverton, OR 97006-6063
(503)626-5700
sequentldave@decwrl.dec.com

Larry Rudolph
Department of Computer Science
Hebrew University
Jerusalem, Israel
rudolph%hujics@wiscvm
rudolph%hujics.bitnet@csnet-relay
2 58-5261

Vijay Saraswat
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213
(412) 268-3075

Karsten Schwan
Department of Computer
and Information Science
2036 Neil Ave. Mall
Room 228, Civil & Aeronautical Engineering Building
The Ohio State University
Columbus, OH 43210
(614)422-8658
schwan.ohio-state@csnet-relay.arpa

Herb Schwetman
MCC
9430 Research Blvd.
Echelon Building #1, Suite 200
Austin, TX 78759-6509
(512)834-3428
hds%pp@mcc.com
hds@balance.pp.mcc.com

Zary Segall
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213
(412)268-3736
segall@a.cs.cmu.edu

H.J. Siegel
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907
(317)494-3444

Daniel Siewiorek
Dept. of Computer Science
Carnegie-Mellon Univ.
Pittsburgh, PA 15213
(412)268-2570
siewiorek@a.cs.cmu.edu

Larry Snyder
Department of Computer Science, FR-35
University of Washington
Seattle, WA 98195
snyder@washington
(206) 543-9265

Tom Sterling
Advance Technology Dept.
Harris Corporation,
Government Systems Sector (GSS)
M/S 3A-2105
P.O. Box 37
Melbourne, FL 32902
(305)729-7098
tron@mit-vax@mit-mc
tron@trantor

Michael Stumm
Department of Computer Science
Stanford University
Stanford, CA 94305
(415) 723-4003.
stumm@su-pescadero.stanford.arpa

Bob Thomas
BBN Advanced Computers
10 Fawcett St.
Cambridge, MA 02238
(617)497-3483
bthomas@bbn.arpa
bthomas@bfly-vax.bbn.com

Dalibor Vrsalovic
Dept. of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213
(412)268-3813
vrsalovic@k.cs.cmu.edu

Ralph Wachter
Office of Naval Research, Code 1133
Department of the Navy
800 N. Quincy St.
Arlington, VA 22217-5000
(202)696-4304
umcp-cs!aplvax!rfw

John Zahorjan
Dept. of Computer Science, FR-35
University of Washington
Seattle, WA 98195
(206)543-0101
zahorjan@uw-krakatoa.arpa

Position Paper: Mapping in Parallel Computation

Fran Berman, UC San Diego

In theory, it is easy to argue that a multiprocessor is faster and more efficient than a single processor computer. In practice, the process of implementing an algorithm on a multiprocessor is usually machine-specific and rife with low-level detail. The algorithm must be expressed in a language and format recognizable by the multiprocessor which may include explicit references to the organization of memory, the processor interconnection configuration, synchronization protocols, etc. In addition, the algorithm must be molded to fit the interconnection architecture of the multiprocessor: if the number of required processes in the algorithm exceeds the resources of the multiprocessor, or if the communication structure of the algorithm differs from the processor interconnection configuration, the algorithm must be mapped and multiplexed by the programmer onto the multiprocessor.

Such difficulties in translating an algorithm onto a multiprocessor render parallel computing inaccessible to many programmers and limit the multiprocessor user community to those willing to do systems or low-level programming. This contrasts strongly to the sequential environment where users can simply translate their algorithms into one of the many supported high-level languages, and generally depend upon existing systems software to perform the implementation and execution of their program on the machine. We expect that parallel computing will be accessible to a wide community when it can offer not only faster and more efficient machines but programming tools with which to use them.

A problem fundamental to the implementation difficulties posed by parallel computing is the mapping problem. The *mapping problem* occurs when there is a mismatch between the required number of communicating processes in the algorithm and number of processors in the multiprocessor, or between the communication requirements of the algorithm and the processor interconnection configuration of the multiprocessor, or both.

Here is an example: Suppose we are interested in implementing an FFT algorithm on a MIMD multiprocessor whose processor interconnection is configured as a 63 node complete binary tree. The communication pattern of the FFT requires 128 processes and performs communication between processes using the connections of a 7-bit shuffle-exchange graph. The mapping problem is then the problem of allocating the 128 algorithm processes to the 63 processor machine, designating paths in the tree configuration which represent the shuffle-exchange communication links in the algorithm, and multiplexing the processes in such a way that the execution of the mapped algorithm will produce the same results as the execution of the original 128 process FFT when it completes. Generally, the process of mapping and multiplexing the algorithm is left to the programmer. As part of our investigation of the mapping problem, we have been interested in designing and developing software tools which perform the mapping and multiplexing automatically, enabling the programmer to interface with the system at a higher level.

We began by studying the mapping problem at a theoretical level, investigating different strategies for performing mappings ([BS]). One fruitful approach was to separate the activities of partitioning processes into groups likely to provide performance-efficient multiplexing *(contraction)*, allocating those groups to processors *(placement)*, designating the communication paths needed by the algorithm *(routing)*, and simulating the process groups at each processor site *(multiplexing)*. In separating these activities, we could seek performance-efficient

solutions for each of these tasks independent of the constraints of the others. Preliminary results of a strategy based on efficient contraction, placement, routing and multiplexing algorithms produced optimal and near-optimal mappings on a diverse set of benchmark examples. Encouraged by these results, we next decided to design software tools based on this strategy which could give us more realistic performance measurements. The system we designed and are continuing to develope is the Prep-P system.

The *Prep-P* system ([BGKRS]) is a software tool which automatically maps and multiplexes a large-sized parallel algorithm into a non-shared memory MIMD multiprocessor. In the system, the parallel algorithm is represented by an undirected graph of bounded degree each of whose nodes is associated with a process, all of which may run in parallel. The target multiprocessor is represented by a fixed-size CHiP machine ([S1]), or more accurately the Poker system ([S2]) which simulates a fixed-size CHiP machine. Prep-P contracts the input graph to an intermediate size, places and routes the intermediate-sized graph on an 8x8 CHiP lattice, and multiplexes the process codes assigned to each PE in the lattice. The resulting code can then be run on the Poker system and will simulate the execution of the original parallel algorithm.

The design and development of the Prep-P system has given us a chance to investigate performance-efficient algorithms and protocols for all stages of this mapping strategy: contraction, placement, routing, and multiplexing. Algorithms under investigation for these tasks have included simulated annealing and local neighborhood search for contraction, simulated annealing and Kernighan and Lin for placement, an optimized breadth-first search for routing, and several different approaches to multiplexing. Substantive modifications based on the requirements of external and internal I/O, available PE memory, and the constraints of the Poker system have made the design and development of these algorithms a complex and engaging research activity.

We are currently configuring and testing the different modules for distribution to interested users with the Poker software (available from Larry Snyder at the University of Washington) sometime next year. We are hopeful that the strategies designed and developed for the Prep-P system will be useful in developing mapping tools for many multiprocessing environments.

References

[BS] Berman, F. and L. Snyder, "On Mapping Parallel Algorithms into Parallel Architectures," Proceedings of the 1984 International Conference on Parallel Processing.

[BGKRS] Berman, F., Goodrich, M., Koelbel, C., Robison, W. and K. Showell, "Prep-P: A Mapping Preprocessor for CHiP Architectures," Proceedings of the 1985 International Conference on Parallel Processing.

[S1] Snyder, L., "Introduction to the Configurable, Highly Parallel Computer," Computer, January 1982.

[S2] Snyder, L., "Introduction to the Poker Parallel Programming Environment," Proceedings of the 1983 International Conference on Parallel Processing.

Position Paper

David L. Black
Carnegie-Mellon University

I suppose this position paper could be subtitled "Parallel Programming for the Masses." In the begining, sequential programs required exclusive use of the computer for their execution; this limited the use of computers to those who could afford to dedicate machine usage to their programs. The development of multiprogramming operating systems and the 'virtual machine' concept was a major step forward in expanding the applicability and availability of computing; many users could now share a(n expensive) computer without fundamental changes in the programming model. (i.e. The user programming model involved exclusive use of the virtual machine, but its implementation did not require exclusive use of the physical machine.) Much of the work in parallel processing is at a similar stage; most parallel programming models envision exclusive use of the machine (usually for scheduling reasons), whereas very few of us have parallel machines available for our exclusive use. This suggests efficient support of parallel programming on multiprogramming multiprocessor operating systems as a promising research direction. At present I am working on implementing an operating system for a symmetric shared memory multiprocessor.

I see three areas of fundamental interest in this regard:

- Shared memory.
- Inter-process communication.
- Scheduling.

The minimalist approach to this area has the operating system hand over a bunch of memory with a group of processors and then get out of the way. This seems no better than the dedicated machine approach; if three users each take a third of the machine, what do you do with the fourth user?? On the other hand, porting existing multiprogramming operating systems to multiprocessors can result in systems that introduce performance bottlenecks and make bad scheduling decisions. I believe a middle ground does exist and can be productively used in many applications; at the workshop I hope to explore ideas of what the middle ground might be, particularly in the areas of scheduling and resource (processor, memory) management.

Programming and Evaluating Computer Architectures

Jordan Brower and Jean-Loup Baer

Department of Computer Science
University of Washington
Seattle, Wa 98195

Until very recently, parallel computations were used primarily to solve time-consuming numerical applications or a restricted class of problems, such as those found in image processing. With the advent of fast, less expensive commercial multiprocessors and the development of experimental academic architectures, parallel processing is now open to a wide variety of applications. Given the diversity of parallel architectures, it is necessary to investigate techniques that determine which architectural and programming language features are best suited for a given application.

Predicting the appropriate target architecture and programming methodology for a given application requires an understanding of the parallel computation at each level of its translation from the nature of the algorithm to its instantiation as a program to its machine level implementation. We plan to develop a series of models corresponding to the three levels just mentioned that will define a hierarchy of important design dimensions for a given application and a mapping that identifies key design decisions between each level.

This three-level hierarchy is the one we are working on now, but it will naturally evolve when we will have had more experience in implementing some numeric and non-numeric applications (e.g., AI production rules and server models) on various architectures (e.g., shared-bus Sequent, shared-memory Butterfly, and possibly a CRAY) using shared memory and message passing program methodologies. We hope to conclude which design dimensions are important to the computation at each level and to those mappings that lead to efficient implementations.

This work was supported in part by a grant from the Washington Technology Center and by NSF Grant DCR-8503250.

At the present time we foresee:

- Application level: A graph model of computations could give a good idea of the nature of parallelism (e.g., SIMD, vectorizable, MIMD with small or large granularity).

- Language level: An appropriate language for the application will reflect the architecture on which it is to be implemented. We can foresee an interaction between the first two levels in the form of an iconic programming language for the graph model (e.g., by defining a first order recurrence) and a series of templates for various architecture-specific langauges at the second level. In any case, the user at this point will need to be concerned with methods of synchronization, process creation, scheduling, and partitioning.

- Architecture level: At this level, the model specifies the implementation of high-level synchronization primitives and memory access. It is in the interaction between the second and this level that we hope to learn the most about which architectures are best suited for given applications.

Presently, we are investigating those design dimensions that adequately express the nature of memory access (shared memory vs. message passing) and synchronization primitives. We are using Larry Snyder's Poker programming environment to implement algorithms (e.g., bounded buffer and scheduled waiting) that use standard programming language synchronization primitives (e.g., monitors, rendezvous, semaphores) on architectures that support a different set of primitives (e.g., HEP's full/empty bit, RP3's Fetch&Add, and Dragon's Conditional Wait). Eventually, we will want to develop dimensions and mappings that describe scheduling, process creating, and network topology, to name a few.

PORTABILITY VERSUS EFFICIENCY FOR PARALLEL PROGRAMS

J. C. Browne
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

ABSTRACT

Attainment of maximal efficiency in execution has always required handtailoring to the execution environment, even for sequential architectures. Vector architectures required a massive investment in restructuring to obtain the benefits of the architecture. Optimizations seldom persist even across vector architectures.

The situation will be much more complex for parallel structuring of computations since there is such a diversity of architectures and execution environments. Further, there is no common basis for expression of parallelism across different vendors' programming systems.

We propose and describe a programming environment where parallel structure is specified declaratively so that ready translation to a spectrum of procedural implementations of dependency relations. This approach allows selection of code which is known to be near-optimal for a given architecture. The declarations are made through a graphical interface and may be applied to programs in most higher level languages.

LA-UR

Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36

TITLE: PARALLEL PROGRAMMING: A USER'S PERSPECTIVE

AUTHOR(S): Ingrid Y. Bucher

SUBMITTED TO: Workshop on Performance Efficient Parallel Programming
Champion, Pennsylvania, September 8-10, 1986

By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes

The Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy

Los Alamos

Los Alamos National Laboratory
Los Alamos, New Mexico 87545

FORM NO 836 R4
ST NO 2629 5/81

Parallel Programming: A User's Perspective

by

Ingrid Y. Bucher

Computer Research and Applications Group
Los Alamos National Laboratory
Los Alamos, NM 87545

Los Alamos National Laboratory, with a tradition of fast scientific computing of more than three decades, has as an organization embarked on the unchartered path of parallel computing. Our users thirst for ever more computing power. We believe that the speedups needed to satisfy that thirst (speedups by factors of 100 or more) can be achieved only by parallel machines. Parallel computing is thus a necessity for us. Members of a small research team have experiences in programming parallel Crays, the Denelcor HEP, the Intel iPSC hypercube, and several makeshift parallel computers of the past, all with very little or no software to ease the task. The machines include both common and distributed memory machines. We are only now starting to gain experiences on data flow architectures. Conversions of major codes to multiprocessor CRAY X-MPs are in progress using Fortran with a multitasking library.

Although the author will take full responsibility for the positions stated in this paper, they should be viewed as fairly characteristic of the people involved with parallel computing at Los Alamos.

1. Speedup Limits

There is a maximum speedup and an associated maximum number of processors that can be usefully employed to the solution of a given problem on a given architecture. These maxima may be but are generally not determined by the maximum amount of parallelism in the problem.

In a very simple model of parallel computations according to Ware [1], the problem consists of a fraction f of the work that cannot be performed in parallel, while the remaining fraction $1-f$ can be executed on any number of processors p in parallel, with perfect load balancing. The speedup S for this model is given by

$$S = 1/((1-f)/p + f) . \tag{1}$$

The maximum speedup for this model is reached for an infinite number of processors and is

$$S_m = 1/f .$$

According to this simple model, increasing the number of processors p will not hurt the execution speed, but of course the efficiency will deteriorate.

However, it is known only too well that with increasing number of processors or processes, communication and data movement costs become a serious problem on distributed memory machines and common memory machines may start choking from memory contentions with or

without hot spots. It seems reasonable to assume that these costs associated with parallel computations are a non-decreasing function $W(p)$ of the number of processes p involved in the solution of the problem. The speedup S resulting from this slightly refined model is given by

$$S = 1/((1 - f)/p + f + W(p)) . \tag{2}$$

Unless $W(p)$ is a constant, the speedup exhibits a maximum S_m for a number of processors p_m. Increasing the number of processors beyond p_m will actually decrease the execution speed as well as the efficiency. The maximum speedup S_m and the maximum number for processors p_m are dependent on the characteristics of the problem and the hardware and software architectures, but most importantly they are also dependent on the mapping of the problem onto the parallel architecture. To achieve high speedups, minimizing overhead costs is, in our opinion, a more difficult and important task than finding the maximum amount of parallelism in a problem. This applies at least to scientific applications that are naturally highly parallel. This parallelism is usually clearly visible to the designer of a code but much harder to unearth from existing codes.

Because the maximum number of usable processors is problem dependent, it will be essential to run several jobs concurrently on a large parallel computer. Research of the behavior of parallel computers in a multiprogramming environment is, therefore, urgently needed.

2. Common Versus Distributed Memory Machines

Common memory machines are much easier to program than distributed memory machines. This is born out by our experiences for manual program conversion but will apply to automated tools as well, especially if efficiency is important. As an example, data movements associated with programming in functional languages should be considered. It is our experience that even if data can be mapped efficiently for a particular computation in a large program, another computation will generally require restructuring of the data. These data movements consume increasing amounts of time with increasing number of processors and increasing problem size. They constitute, in our opinion, the main performance limitation for distributed memory machines. Programming of a complex problem on a distributed architecture, therefore, often requires a complete rethinking of the solution process. An extraordinary amount of artificial intelligence will be required to automate the efficient mapping of such complex problems onto a distributed memory architecture. The task completely defies the divide-and-conquer strategy essential to most modern programming techniques and is, therefore, particularly nasty.

We feel that user input for data organization will be needed for efficiency. New languages should provide for this.

3. Top-Down Versus Bottom-Up

In parallel processing of scientific applications (and probably others as well), the crucial problem is not to find enough parallelism but to find a way of subdividing a problem into subtasks that minimize costs associated with communications, synchronization, data movements, memory contentions, and all other costs that increase with the number of processes.

The bottom-up approach examines innermost loops first. It uncovers parallel subtasks of small granularity. These small subtasks usually involve high communication costs for all parallel

architectures and frequent and complex data movements in distributed memory machines. The bottom-up approach has been successfully automated, e.g., by Kuck and his students [2].

The top-down approach initially examines the outermost loops and, therefore, uncovers the coarsest parallelism first. It can be pursued until a sufficient number of parallel subtasks are found. Because of the coarser granularity, overhead costs are usually much smaller than from the bottom-up approach. However, it is harder to automate. Interactive user input might be helpful. The basic parallel structure of the problem is usually obvious to the code designer. Even static data dependency analysis tools at this point would be very helpful.

4. Debugging and Tuning

Debugging parallel programs is much more difficult than designing them. We feel that dynamic debugging tools will be more successful if they trace data rather than control flow.

In addition to debugging tools, dynamic tuning tools are needed. This will require development of new concepts for the performance assessment of parallel programs, especially in a multiprogramming environment.

5. Conclusions

Efficient parallel programming is a considerable challenge that will require a wealth of new ideas to become reality.

6. References

1. W. Ware, "The Ultimate Computer," *IEEE Spectrum*, March 1972, pp. 84-91.

2. D. T. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," in *Proc. 8th ACM Symp. Principles Programming Languages*, 1981.

LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Amdahl's Law Revisited: Measurements of Dataflow Programs

August 29, 1986

David E. Culler

Position Paper for the
1986 Workshop on Performance Efficient Parallel Processing

This work was performed at the M.I.T. Laboratory for Computer Science under the Tagged-Token Dataflow project. Funding is provided in part by the Advanced Research Projects Agency of the U.S. Department of Defense, contract N00014-75-C-0661, and in part by International Business Machines Corporation, T. J. Watson Research Center.

Source File = POSITION.MSS.33, Last updated 29 August 1986 at 2:38pm

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

David E. Culler
Position Paper

Amdahl's Law Revisited: Measurements of Dataflow Programs

The Computation Structures Group[1] at M.I.T. has constructed fairly powerful tools for developing, executing and evaluating dataflow programs. The Id dataflow compiler [7, 9] and an interpreter for the Tagged-Token Dataflow Architecture have been integrated with the programming environment provided on various Lisp machines [8] to facilitate development of dataflow applications. The interpreter can model a variety of abstract architectures, and forms the basis of a 32-processor dataflow emulator. This paper offers some preliminary results derived using these tools and puts forward certain points that the author feels are critical to assessing the viability of dataflow processing.

1. Parallelism in Programs

Parallel processing with all its glamour should be approached with a healthy respect for Amdahl's law. That is to say we should attempt to quantify the amount of parallelism present in real applications, as this ultimately determines the scale of machines to consider. We know that many algorithms offer tremendous parallelism, but in large applications it is possible that the "glue" between such well-behaved sections may undermine the potential parallelism as a whole. Also, the language in which an algorithm is coded may prohibit or obscure certain kinds of parallelism. Lastly, the machine on which the program is executed may be unable to exploit certain kinds of parallelism.

Our approach is to determine the *potential parallelism in programs* by considering only the essential data dependencies involved in computing the result and then try to understand how this potential parallelism is attenuated in coding and execution. The formal model of computation embodied in the U-interpreter [3] embellished with I-structures is important in this regard, as it places minimal constraints on execution order beyond the data dependencies, and yet allows applications to be evaluated in full detail. It is assumed that instructions execute in unit time and results are transmitted to wherever they are required with zero latency. An instruction executes as soon as its operands are available; and an arbitrary number of instructions may execute in a single timestep. Under this model we can compute the *parallelism profile* (*i.e.*, number of concurrent

---

[1] This group, lead by Professor Arvind, was formerly called the Functional Languages and Architectures Group.

operations over time) of a program. Figure 1 shows the parallelism profile for a version of SIMPLE, a sizable hydrodynamics and heat conduction code, on a 32x32 mesh.

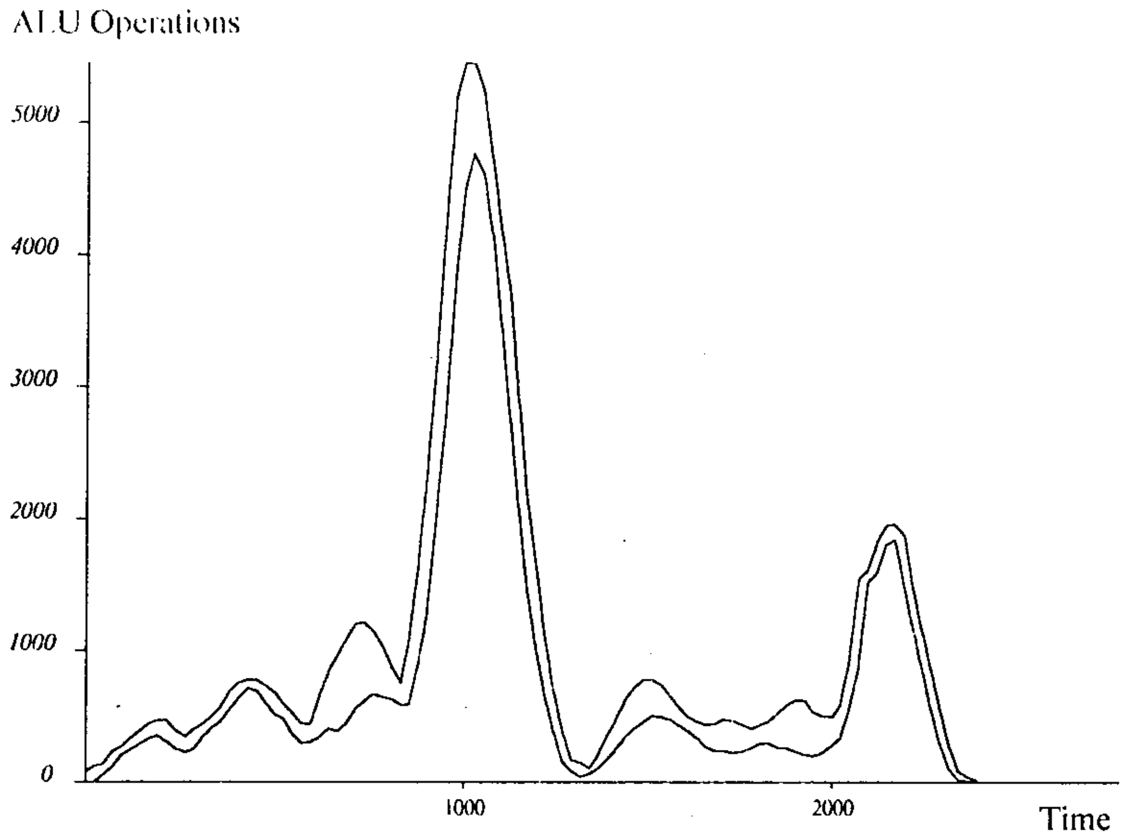


Figure 1: Parallelism Profile for a Sizable Application

The fluctuations in this profile over time are notable. In the first phase of the application, the force is computed at each point based on neighboring points, giving $O(n^2)$ parallelism for an n by n mesh. (Note, this checks against the large spike in the profile.) Results of force calculation are reduced to a single value, as suggested by the constriction point in the profile, that is used in the latter heat conduction phase, which involves a recurrence over the rows and then over the columns. A "real" problem would involve 100,000 iterations of a 100x100 mesh. Fluctuations in potential parallelism such as appear here are quite typical, although in many cases can be reduced with careful application design. It should be clear that even though the application has thousand-fold maximum parallelism, the utilization on a thousand processors would be rather poor.

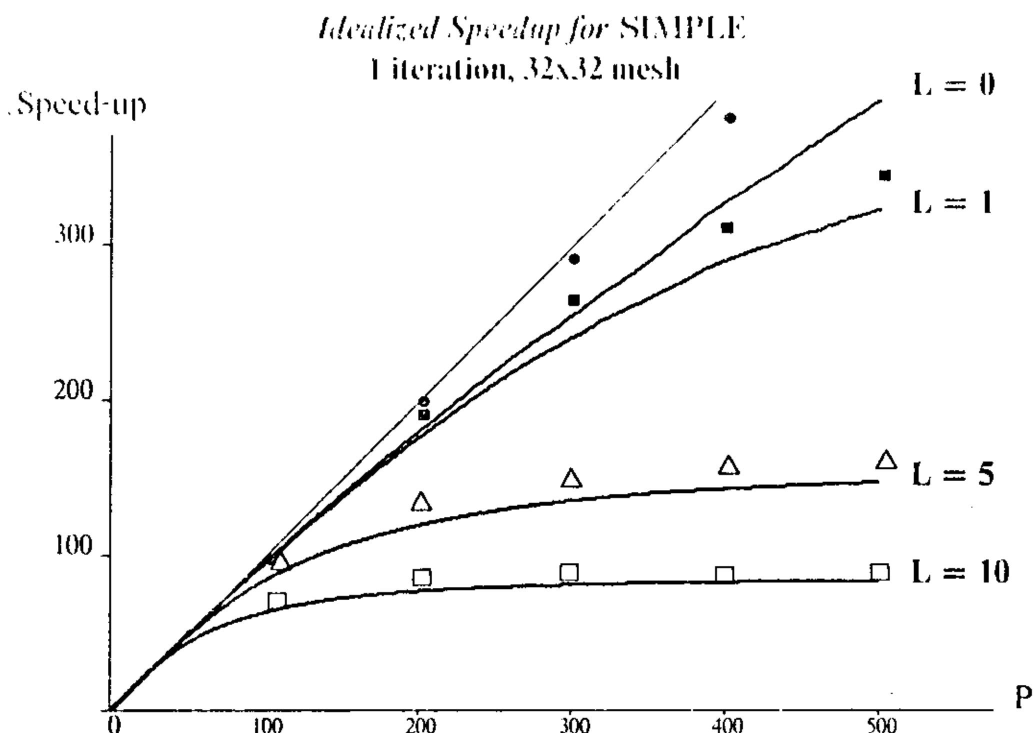
Given such a parallelism profile we can employ the sort of analysis embodied in Amdahl's law

with unusual precision to identify the point of diminishing returns for the program. Let $T(p)$ be the number of timesteps required to execute the program, where at most p operations are performed per timestep. By definition, $T(\infty)$ is the length of the profile and $T(1)$ the area. For intermediate values of p we can derive a conservative, yet reasonably accurate, approximationof $T(p)$ by assuming it takes $\lceil PP(t)/p \rceil$ timesteps to do the work involved in step t of the parallelism profile, PP(t). This correctly accounts for under-utilization when there is less than p-fold parallelism, and is somewhat conservative when more parallelism is available. The curve labeled $L=0$ in Figure 2 shows the potential speedup using this formula for the profile in Figure 1.

Calculating potential parallelism in this manner is a step toward separating two concepts which are often confused: the parallelism present in a program and the parallelism exploited in running a program on a particular machine. The former determines the best that can be achieved in the latter. If we fail to achieve substantial speedup on a problem, the parallelism profile provides a basis for determining whether the application is intrinsically sequential or the implementation has compromised the potential parallelism. In particular, it should be possible to make substantive statements about the amount of parallelism present in conventional applications.

To better predict the potential speedup on a real machine we must either embellish the interpreter to more closely reflect the machine, or refine the analysis. For example, it is argued that a dataflow machine can tolerate communication latency if sufficient parallelism is present, because enabled activities are processed while recently generated results are enroute to their destinations [4]. This can be incorporated in the analysis by assuming that the computation involved in $PP(t+1)$ can not start until at least $1+L$ time units after that in PP(t) has started. Figure 1 shows the result of this analysis for four different values of L. To substantiate this analysis, the interpreter was embellished to model an ideal dataflow machine characterized by two parameters P, the maximum number of operations per timestep, and L, the communication latency involved in each token transfer. Other aspects of the model are completely idealized, *e.g.*, unbounded resources and distribution of work by oracle. The dots of various shapes in Figure 2 show results obtained for this idealized machine for values of L corresponding to the four curves. Note that as the latency term becomes dominant the approximation becomes quite accurate.

This kind of analysis can be extended to incorporate the effects of granularity of distribution of work, load non-uniformity, and locality, for example. However, as the analysis becomes more detailed the interactions between various aspects become more complex.



Curves are based on the Parallelism Profile using:

$$T(P,L) = \sum_{t=0}^{T_\infty} \mathrm{MAX}\left(1+L, \left\lceil \frac{PP(t)}{P} \right\rceil \right)$$

$$\text{Speed-Up}(P,L) = \frac{T_1}{T(P,L)}$$

Dots represent speed-up measured on an indealized machine:

• L = 0 ■ L = 1 △ L = 5 □ L = 10

Figure 2: Potential Speedup Based on a Parallelism Profile

An equally interesting and probably more subtle question is how to account for the loss of potential parallelism due to the mechanisms employed for synchronization and limitations of the programming language. For dataflow processing, the graph schemas used in compiling code, the mechanism for detecting enabled activities, and the model of arrays all effect the amount of parallelism that can be exploited. For conventional multiprocessors the question becomes very complex since many different synchronization mechanisms are employed.

2. Overhead

A second crucial issue in assessing the viability of an approach to parallel processing is the total work required to execute a program. There is undeniably a cost in logically partitioning a program so that it may run in parallel, and we should try to quantify this cost. Speed-up curves often obscure this issue by presenting performance relative to single processor executing code with all the auxiliary operations required for parallel execution. Although it would be reasonable to consider hardware complexity and other factors in a cost metric, we will focus on the number of instructions executed.

Critics have postulated that dataflow programs will require many more instructions than corresponding programs on a conventional machine [6]. Our experiments support this concern, but also suggest that the difference is not as large as has been assumed. Table 1, generated as part of a study in conjunction with Dr. K. Ekanadham of IBM Research, shows dynamic instruction mixes for variations of a Guassian relaxation code compiled from the dataflow language Id and from FORTRAN. The precise numbers are less important than the general trend, but a few remarks are in order. The FORTRAN versions are highly optimized for the 370, so some instructions include memory references and arithmetic operations. Separating these operations as would be required for a load/store architecture narrows the gap slightly. The Id versions can be improved substantially with even trivial peep-hole optimizations. Thus, with comparable compiler sophistication the ratio of total instructions executed may be closer to 2:1. On the other hand, without relatively sophisticated graph schemata, *e.g.*, support for loop constants, the ratio would be much worse.

The missing piece of this comparison is how dataflow instruction counts compare against that for programs running on conventional *parallel* machines, where synchronization, etc. plays a major role. Our belief is that there is a certain cost in simply generating code that can run in parallel, and there may be additional costs, *e.g.*, more busy-waiting or more context swaps, as more parallelism is exploited. The various transformations to expose parallelism in FORTRAN programs should be examined in this light. With dataflow, the full cost is borne up front.

3. Resource Requirements

Another crucial issue which has been largely overlooked in the literature is how the resource requirements increase as parallelism is exploited [5]. This may be the most severe obstacle for approaches based on implicit expression of parallelism. A case in point is the token storage

Table I: Instruction Mix comparison between ID and FORTRAN

Inner loop - roughly 10,000 iterations: Each iteration has:

	5 Fl-ops	5 Fl-ops	6 Fl-ops	7 Fl-ops
	opt	std	G1+sum	G2+cond
Op category	G0	G1	G2	G3
ID version				
Float (+-*/ compare)	48,050	48,050	57,660	67,270
Fixed (+-*/ compare)	44,409	63,515	63,515	63,515
Loads	55,387	113,650	113,982	133,522
Stores	16,828	15,352	15,684	16,004
Switches	24,057	24,253	34,504	83,174
D	23,306	23,496	33,426	52,646
All other	100,996	135,612	146,869	139,469
Total	313,033	423,928	465,640	555,600
FORTRAN version:				
Float (+-*/ compare)	48,186	48,186	57,796	67,406
Fixcompare)	496	20,336	20,336	20,336
Loads	20,836	20,846	30,817	30,817
Stores	12,104	12,104	21,735	21,735
Switches	11,708	11,708	11,708	21,318
Logic\compare)	892	20,102	20,102	29,712
All other	30	30	30	30
Total	94,252	133,312	162,524	191,354

requirement of programs running on a dataflow machine. The upper left portion of Figure 3 show the parallelism profile and token storage requirement of a 16 by 16 matrix multiple on an unbounded processor machine. With 500 to 1,000 fold parallelism, a storage requirement of 3,000 tokens does not seem unreasonable. However, if this program is run on relatively small machine, *cf.* the lower left portion of Figure 3 where the machine can exploit only 20 fold parallelism, the resource requirements remain large. The upper right portion of Figure 3 shows how the resource requirements can be reduced by a simple graph transformation that limits the potential parallelism by constraining loop unfolding. This constrained program is well matched to our 20 processor machine, as can be seen in the lower right portion of Figure 3.

This resource problem will appear in any system that supports a general facility for dynamic generation of parallel activity [1]. It arises with almost all resources related to program execution,

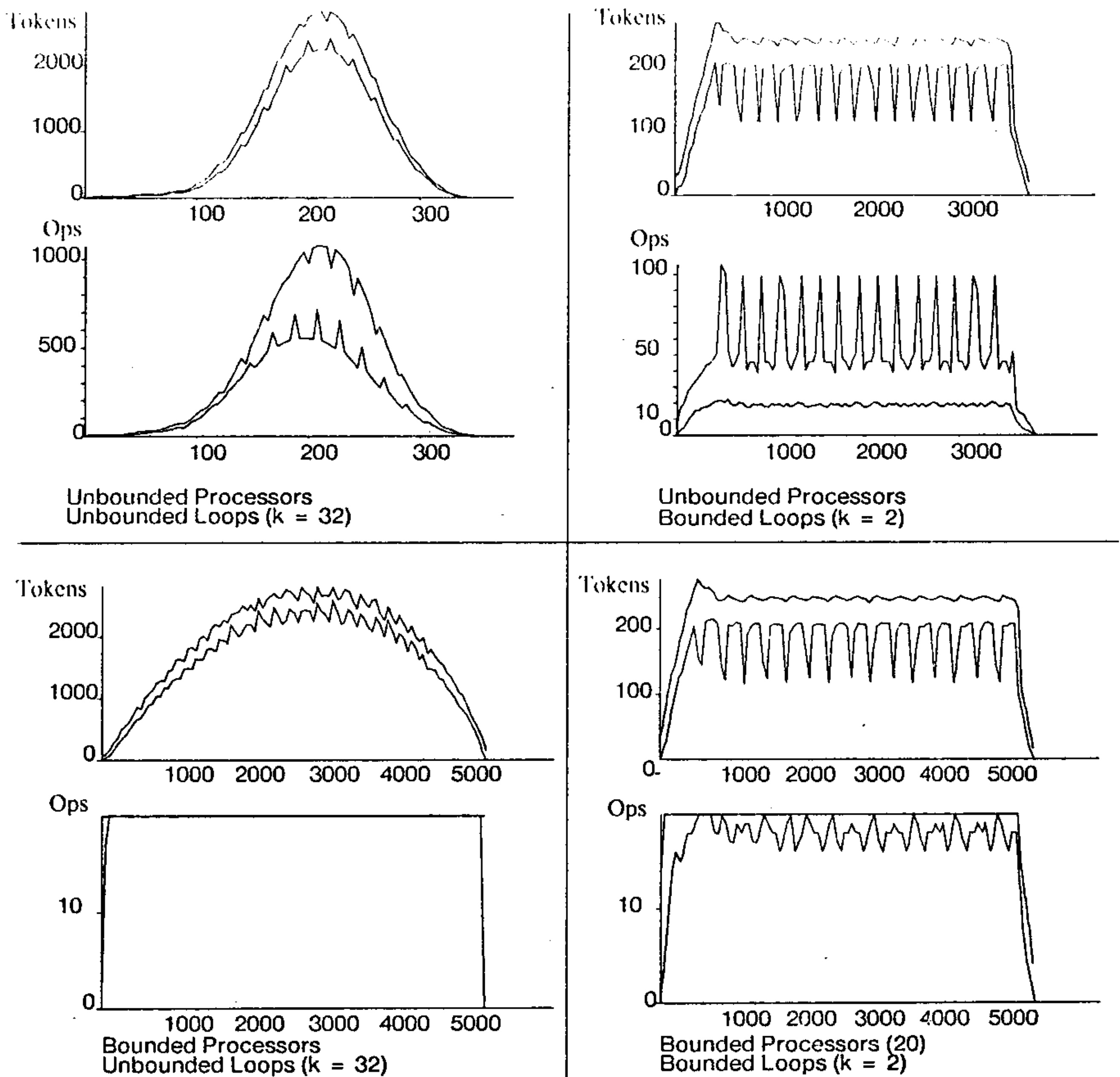


Figure 3: Effects of Constraining Parallelism to Reduce Resource Requirements

not just scalar storage. Techniques such as constraining loop unfolding are extremely important, but it remains an open question how to best apply such controls.

This problem arises in a slightly different form in many of the transformations used to expose parallelism in sequential programs, *e.g.*, scalar expansion, even though the form of parallelism is relatively limited.

4. Distribution of Work

The concern which has received most press in the parallel processing literature is how to distribute work and data over a collection of processors. Much of this research alludes to trade-offs between techniques for achieving uniform load and those for achieving locality. Our studies have raised a number of concerns which bear on this debate.

We have conducted a variety of experiments in which each of the 32 TI Explorers that comprise the MIT Multiprocessor Emulation Facility [2] emulate a (rather slow) dataflow machine. We took an expedient approach, distributing work on a code-block basis, *i.e.*, relatively course granularity, using a variety of simple distribution policies. With this approach substantial load imbalances are not uncommon, even when the number of processors is quite small.

The argument for course granularity is generally based on the locality issue. If larger chunks of computation are assigned to each processor, less communication is required. However, the instruction mixes above appear to reduce the force of this argument; one quarter of the instructions involve access to an arrays and optimizing data placement to maximize locality of array references is extremely difficult, even potentially detrimental. Thus if array references are relatively uniform, one quarter of the data values produced in executing a program will generate network traffic, regardless of granularity. Nonetheless, there is a strong argument for course granularity: in designing a code-block engine, rather than an activity engine, many optimizations are possible. There are aspects of our current graph schemata that are crucial to keeping the number of instructions reasonable and are more difficult to implement if work is distributed willy nilly across the machine.

5. Conclusion

Dataflow has long offered the potential for high performance parallel computation, but only recently have tools become available which will allow us to move from vague statements of "potential" to firm statements of viability. It is extremely important to assess the limits of the approach and to identify the most promising regime, in term of number of processors, application domain, etc., to focus further architectural development. Part of that assessment hinges on an understanding of the potential parallelism in programs, the overhead implied by the approach, the resource requirements of programs, and our ability to effectively distribute work. Another important part of the assessment is a clear comparison with competing approaches.

References

1. Arvind, and D. E. Culler. Managing Resources in a Parallel Machine. Proceedings of the IFIP TC-10 Conference on Fifth-Generation Computer Architecture, Manchester, U.K., July, 1985.

2. Arvind, M. L. Dertouzos and R. A. Iannucci. A Multiprocessor Emulation Facility. TR-302, Laboratory for Computer Science, MIT, Cambridge, Mass., October, 1983.

3. Arvind, and K. P. Gostelow. "The U-interpreter". *Computer 15*, 2 (February 1982), 42-49.

4. Arvind, and R. A. Iannucci. A Critique of Multiprocessing von Neumann Style. Proceedings of the 10th International Symposium on Computer Architecture, Stockholm, Sweden, June, 1983, pp. 426-436.

5. Culler, D. E. Resource Management for the Tagged-Token Dataflow Architecture. TR-332, Laboratory for Computer Science, MIT, Cambridge, Mass., January, 1985.

6. Gajski, D. D., Padua, D. A., Kuck, D. J., and Kuhn, R. H. "A Second Opinion on Dataflow Machines and Languages". *COMPUTER 15*, 2 (Feburary 1982), 58-69.

7. Heller, S. K., and Traub, K. R. The Id Compiler User's Manual. CSG Memo 248, Laboratory for Computer Science, MIT, Cambridge, Mass., May, 1985.

8. Morais, D. Id World: User's Manual". CSG Memo 266, Laboratory for Computer Science, MIT, Cambridge, Mass., June, 1986.

9. K. R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Master Th., Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass.,August 1986.

DEBUGGING HIGHLY PARALLEL COMPUTATION*

Janice E. Cuny
Department of Computer and Information Science
University of Massachusetts, Amherst

Highly parallel computation – in which large numbers of closely coupled processes cooperate on a single task – is not amenable to existing sequential debugging techniques. Parallel programs do not have the consistent global states, manageable quantities of potentially relevant information or reproducibility of results that have formed the basis for sequential debugging paradigms. Instead, their behavior is best understood in terms of the flow of data and control resulting from interprocess communication. These behaviors are often structured: low grain, closely coupled processes communicate across regular interconnection networks resulting, at least logically, in very patterned data and control flows. We believe that these patterns of data and control flow will form the basis for highly parallel debugging paradigms.

Parallel debuggers must provide the programmer with facilities for monitoring and manipulating the patterns of activity within his system. Before this can happen, however, there are at least three areas that need development:

The presentation of information. Because patterns are best understood visually, it will be necessary to design graphic displays and animations of communication structures.

The selection of relevant information. Because enormous amounts of information are available in an executing parallel system, it will be necessary to provide the user with facilities for restricting the presented information according to a variety of criteria.

The interpretation of system behavior. Because programmers understand their code in terms of intended, logical patterns of activity that are often obscured in asynchronous execution, it will be necessary to be able to create mechanism for detecting

* The Parallel Programming Environments Project at the University of Massachusetts is supported by the Office of Naval Research under contract N000014-84-K-0647.

logical patterns within execution traces.

We are investigating these issues with the development of an extensible debugger, called Belvedere, that will allow programmers to easily construct a variety of post-execution animations of process interactions within highly parallel systems. It will feature both a view generator and an animator. The view generator will filter data for relevance based on fish-eye views [1] in which "neighborhoods" are shown in great detail while more distant areas are shown in successively less detail. We are developing further generalizations to the fish-eye model that include multiple distance criteria and foci of attention. The animator will interpret the execution trace of the system according to user supplied descriptions of intended behavior. We expect to describe behavioral patterns with an extension of the Event Definition Language [2]. Belvedere runs as part of the Simple Simon Programming Environment [3] which, in turn, runs as a front end for the Simon Multiprocess Simulator [4][5] allowing us to consider the suitability of our debugging techniques for a variety of architectures.

REFERENCES

[1] George W. Furnas, "Generalized Fisheye Views," *Proceedings of the Conference on Human Factors in Computing Systems*, pp. 16-23 (April 1986).

[2] Peter C. Bates and Jack C. Wileden, "High-level debugging of distributed systems: the behavioral approach," *Journal of Systems and Software* 3, pp. 255-264 (1983).

[3] Janice E. Cuny, Duane A. Bailey, Alfred A. Hough, Mary E. Larson, and Neville D. Newman. The Simple SIMON Facility for Research in Programming Environments for Highly Parallel Computation. In preparation.

[4] D. E. Heller, "Multiprocessor Simulation Program SIMON," Shell Development Corporation (1985).

[5] R. Fujimoto, "SIMON's user's manual," Univ. of Cal. at Berkeley (1984).

Single-Assignment Languages in the Flavor of SISAL

John Feo

Single-assignment languages, by requiring that each variable or instantiation be unique and defined (i.e., assigned a value) only once, avoid many of the problems inherent in using conventional languages for parallel programming. We believe that such languages, as exemplified by SISAL, can provide an expressive and efficient efficient medium in which to write large-scale scientific parallel programs.

By insisting on single-assignment, one removes side-effects and aliases from the language. Obviously, if a variable can be defined only once, its value can not be changed once assigned either in or out of a subroutine. Similarly, all parameters must be passed to subroutines by value, since each instantiation of a variable is unique; therefore, there are no aliases. These features reduce the complexity of a process' data dependencies, thereby simplifying the analysis required to extract the inherent parallelism. This in turn leads to simpler (even naive) compilers, and makes the parti- tioning and scheduling of subtasks easier and more straightforward. A good solution to the latter is particularly important if processes are to execute efficiently on actual machines.

The advantages of single-assignment languages can be enhanced by including various specific features in the language. Two such features provided by SISAL are: 'no use-before-definition' and explicit parallel constructs (ForAll and Multi-expressions). The former removes all data cycles, reducing a process to an acyclic graph. This is a sufficient condition to guarantee that no set of concurrently executing tasks can deadlock. By providing explicit parallel constructs in the language, the user can clearly express the coarse-grain parallelism of his algorithm; such may not be the case without the constructs. At the very least it relieves the compiler of the task of finding such parallelism.

Unfortunately, single-assignment languages have a number of draw- backs including: memory management problems,interfacing with other languages and a restricted problem domain. Since a name can be assigned a value only once, every update requires a new copy. In the case of large data structures, as are typically found in scien- tific codes, it can be very expensive. Furthermore, since all para- ameter passing is by value, each invocation of a function re- quires its own copy of the arguments. The lack of side-effects and persistent state (i.e., common blocks) makes it difficult to inter- face processes written in a single-assignment languages with libraries written in conventional languages (especially, FORTRAN). Finally, implementing an algorithm requiring state is extremely inefficient in such languages since the state must be scattered and gathered at every step; further, implementing a non-determinis- tic computation may be impossible.

Our current work focuses on many of these problems. One project is investigating the advantages of preallocating memory and perform- ing updates-in-place to solve the memory management problem. A second is developing compilers and systems for efficient code generation. The approach is to reduce a source to graph form, then to an intermediate code and finally, to machine code. A number of target machines have been selected, including: a VAX 780, a Cray- X/MP, a Cray-2, a Loral Data-Flow and a Sequent 21000. A third project is concerned with finding optimal heuristics to partition process graphs to minimize communications and maximize parallelism and machine

utilization. Two projects still in embryonic form are: an investigation into how SISAL and FORTRAN routines can be inter- faced and how a target machine's architecture affects partitioning and granularity decisions.

Position Paper

Jeanne Ferrante
IBM T. J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

As noted in the announcement of the workshop, as yet there is no one model of parallel computation underlying the diverse architectures now being proposed and built. Such computational models are the basis not only of our machines but also of our high level languages. Hence any high level parallel programming language can be compiled easily and efficiently only when it matches the underlying computational model of the machine we are compiling for. Given a high level parallel programming language which does not fit a particular architecture well, in order to produce a translation which can run efficiently on the machine, it is necessary to map the language's computational model to that of the machine. This is essentially the same problem as automatically detecting the parallelism in a sequential language in order to map to a parallel machine. **The same techniques useful for automatic parallelization of sequential languages are useful for compiling parallel high level languages to different parallel architectures.**

Many of our high level languages embody various aspects of the Von Neumann machine in their underlying computational model; one such aspect is reference to storage. Languages like Fortran and Pascal allow storage to be directly referenced and manipulated. In a parallel machine context, this reference to storage forces a sequencing of these references, and thus less parallelism. In translating such languages to run efficiently

on a parallel machine, the storage model embodied in the user program must be drastically altered. Because of the diversity of parallel architectures, this alteration itself can take vastly different forms. I therefore propose the following general technique. **In compiling high level languages which allow references to storage for a parallel machine, the references to storage should be removed and a totally VALUE-ORIENTED representation should replace it.** This will increase the potential parallelism and thus be a better basis for further transformations to increase parallelism.

Language tools for large-grain parallelism

Raphael Finkel
University of Wisconsin — Madison

Position paper
Workshop on performance-efficient parallel programming

1. Algorithms for large-grain parallelism

The range of parallel architectures is quite wide. In this paper, we focus on multicomputers, which consist of many computers, each with its own memory, cooperating by sending messages. Such machines lend themselves to large-grain parallelism, where cooperation events occur at a maximum rate of hundreds per second, not thousands (as in medium-grain parallelism) or millions (fine-grain). In practice, algorithms in this class strive to keep the amount of communication as low as possible to avoid message-passing costs.

Many algorithms fit into the following classes.

- **Generate and solve.** A problem can be subdivided into subordinate problems, each of which can be solved independently of the others. A pool of **slave** processes stands ready to solve these problems as they are generated and distributed by a **master** process. This category includes tree-search algorithms such as alpha-beta search.
- **Iterative relaxation.** The data space can be divided into adjacent regions, which are then parcelled out to different processes. Each process carries out activities local to its region, communicating with neighbors when necessary. This category includes solution of numerical problems like PDEs and graph problems like finding a minimal spanning tree. Termination is often difficult to determine in such algorithms.
- **Passive data pool.** A large data space is managed by many processes, which support queries and updates on that space. Queries from **client** processes are directed to the appropriate data **server** processes. This category includes distributed file systems and other data structures such as hash tables. These algorithms try to allow a high throughput of queries by letting non-interfering queries proceed simultaneously.
- **Systolic.** Data values flow through a set of processes, undergoing modification along the way. The processes are often arranged in a regular structure, such as a linear array or a square mesh. This category includes many numerical algorithms, pipeline algorithms, and multi-pass transformers such as compilers and scene analyzers.

Algorithms may engage in **restructuring** during the course of a computation. First, the allocation of data to processes may change. Data motion can be a result of attempts to balance load among processes or to bring values to

where they are needed. Second, the process structure may change. New processes may be created as the size of the problem warrants, and new inter-process communication paths may be opened to fulfill new requirements.

A less dynamic form of restructuring comes from **quotient** schemes, in which a single **physical** process simulates the activities of several **virtual** processes. Quotient schemes are particularly useful in pipeline and iterative-relaxation algorithms when the grain of parallelism is too fine and communication is needed too often. All communication between virtual processes within the same physical process can be done cheaply, and communication that crosses the physical-process boundary can often be batched, so that one physical message represents many virtual messages. The boundary between physical machines is often a boundary between two representations of data as well. Within a physical machine, the fact that work is distributed often remains implicit in loops across arrays or traversals of graphs. Between physical machines, the fact that the data structures are distributed is explicit. Data often have two representations, therefore, depending on their proximity to this boundary.

Algorithms based on iteration often can be described either as synchronized or chaotic. **Synchronized** algorithms go through well-defined rounds, between which information is passed among the processes. The exchange of information becomes a bottleneck. One way to reduce that bottleneck is to use a **chaotic** algorithm, in which one process may start the next round before others have finished. The cost of such a scheme is often an increased number of rounds.

To reach agreement on shared data, an algorithm can either broadcast the data, which incurs communication costs, or it can compute the data independently on several processes. Under this latter strategy, identical work is conducted by several processes, reducing communication cost at the price of reduced parallelism.

There may be many ways to advance to a goal. A serial algorithm may sort them and try the best ones first. A distributed algorithm may speculate and try several at once. Although this strategy may occasionally be very lucky, leading to a speedup anomaly, it will often waste the efforts of those processes searching less useful paths when another process is pursuing the best path.

2. Programming languages

Some important trends in programming language design must be encouraged for parallel programming[Cashin80, Andrews83a, Scott84a]. At the least, a reasonable language provides ordinary sequential operations and a way to send messages between processes. Inter-process communication should be abstracted as a form of remote procedure call, although there are rare situations in which this paradigm is not quite right (for example, requests that have both an immediate answer and a delayed answer). Typically, the programmer must specify how work is divided among processes; the compiler does not attempt this

division.

Several features seem essential.

- **Light-weight tasks.** Such tasks are relatively inexpensive to create and share memory with each other, although they may be subject to scoping restrictions for data access. Light-weight tasks are especially useful for maintaining the state of a server-client conversation while other conversations are taking place. Synchronization mechanisms are needed to prevent unwanted interference between tasks; these mechanisms include priority schemes, explicit conditional waiting, semaphores, and monitors. Of these, explicit conditional waiting is the most expressive from the programmer's point of view, but is not always inexpensive to implement.
- **Implicit and explicit message receipt.** Explicit receipt makes sense when the algorithm has reached a point where it knows that it cannot proceed unless a particular message arrives. However, messages that arrive during the course of other computation must also be dealt with. Implicit receipt starts a new light-weight task for each such message; the task begins its execution in whatever procedure the message is trying to call.
- **Context.** When an implicitly received message starts a new light-weight task, that task should have a context appropriate to the particular process that sent the message. This context can be provided by scope rules that provide global variables that are not necessarily shared by other tasks.
- **Message-based type checking.** Wherever possible, the compiler should ensure that messages are properly formed. For complex algorithms built out of several compilation units that are compiled at different times, declaration libraries can be used. Inexpensive run-time checks are also possible and should always be used[Scott84b].
- **Selectivity.** A process may want to provide only a subset of its exported entry points at a given stage of an algorithm. It is necessary to provide a way to dynamically change the interface, that is, the set of legal entries for implicit receipt. In addition, it must be possible to present different interfaces to different peers of the process.
- **Exceptions.** When a process sends a request that is misformed or illegal in some sense, it is possible to respond with an error indication. However, inspecting all responses for the presense of this indication is a heavy burden, and most programmers are not particularly careful about it. Instead, an exception mechanism such as is found in Ada[United States Department of Defense83] should be used to propagate errors. This mechanism can also be used to send signals between light-weight tasks within the same process. It is also essential for terminating speculative work.

The Lynx language[Scott84c], which has been implemented both on the Crystal multicomputer[DeWitt84] and the BBN Butterfly[Scott86], has all of these features.

Argus[Liskov83] includes the ideas of type checking and light-weight processes, and also has a well-developed notion of **transaction**, which is important in recovering from failures.

References

Andrews83.
Andrews, G. R. and F. B. Schneider, "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys* **15**(1) pp. 3-44 (March 1983).

Cashin80.
Cashin, P. M., "Inter-process communication," Technical Report 8005014, Bell-Northern Research (June 1980).

DeWitt84.
DeWitt, D., R. Finkel, and M. Solomon, "The Crystal multicomputer: Design and implementation experience," Technical Report 553, University of Wisconsin—Madison Computer Sciences (September 1984). To appear, IEEE Transactions on Software Engineering

Liskov83.
Liskov, B. and R. Scheifler, "Guardians and actions: Linguistic support for robust, distributed programs," *ACM TOPLAS* **5**(3) pp. 381-404 (July 1983).

Scott84.
Scott, M. L., "A framework for the evaluation of high-level languages for distributed computing," Computer Sciences Technical Report #563, University of Wisconsin—Madison (October 1984).

Scott84b.
Scott, M. L. and R. A. Finkel, "A simple mechanism for type security across compilation units," Computer Sciences Technical Report #541, University of Wisconsin—Madison (May 1984).

Scott84c.
Scott, M. L. and R. A. Finkel, "LYNX: A dynamic distributed programming language," *1984 International Conference on Parallel Processing*, (August, 1984).

Scott86.
Scott, M. L., "Lynx reference manual," BPR 7, Computer Science Department, University of Rochester (March 1986).

United States Department of Defense83.
United States Department of Defense,, "Reference Manual for the Ada Programming Language," ANSI/MIL-STD-1815A-1983 (February 1983).

Parallelism in Production Systems

Charles L. Forgy
Carnegie-Mellon University

The PSM (Production System Machine) group at CMU is exploring ways to use parallelism to achieve cost-effective increases in the speed of production system interpreters. For the past few years the emphasis has been on speeding up conventional production systems like OPS5 or SOAR. The emphasis is now shifting to the tasks being performed by the production systems; the PSM group is exploring the explicit use of parallelism by the application programs. This could substantially increase the total amount of parallelism that the programs can use. For example, if the interpreter achieves an average speed up of 10 through parallelism and the application program achieves a speed up of 5 through application parallelism, the combined speed up factor would be 10 * 5 = 50.

The work on parallelism within the interpreter has shown that modest speed ups -- perhaps one decimal order of magnitude -- can be achieved on the right machine. There are a number of factors that prevent the speed up from being greater. The most time-consuming operation in a production system interpreter is performing the match, and consequently it is the part that one must concentrate on. (The match is responsible for evaluating the condition parts of the production rules to determine which are satisfied on each cycle of the system.) After looking at a number of alternatives, the PSM group concluded that state-saving algorithms had to be used in the match. A state-saving algorithm keeps track of the contents of working memory and the degree of satisfaction of all the condition parts of the rules as the system runs. As each change is made to the system's data, the stored state is updated rather than being recomputed from scratch. With a state-saving algorithm, the amount of parallelism that can be exploited is limited. Simulation studies of existing production systems indicate that for most applications, there is no advantage to having more than 32 to 64 processors available. For these systems, speed-ups of around a factor of 10 are expected.

The obvious way to increase the amount of exploitable parallelism when a state-saving algorithm is used is to increase the rate with which the system makes changes to its data. This could be achieved either by increasing the semantic content of the productions (so that each production does more) or by permitting the system to execute more than one rule on each cycle. The latter seems more appropriate for knowledge-intensive applications, and it is the course that the PSM project is now following. Presently some large applications are being analyzed to determine how they can be decomposed to make use of explicit parallelism. The production systems languages are being examined to determine how they must be changed to support application parallelism. Languages such as OPS5 will require some relatively minor extensions; SOAR already contains mechanisms to support parallel firing of rules.

DIMENSIONS FOR DESCRIBING PARALLEL ALGORITHMS

Leah H. Jamieson

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Position Paper

For effective use of parallel systems, it is essential to obtain a good match between algorithm requirements and architecture capabilities. What information can facilitate the mapping of a parallel algorithm to a parallel architecture? Such information can be of use in a number of different ways. First, it bears directly on the algorithm design process. General knowledge about what constitutes an effective match between a parallel algorithm and a parallel architecture can accelerate the process of developing new parallel algorithms for a given machine. Second, an understanding of the relation between algorithms and architectures is a prerequisite for the fast, efficient design of algorithmically-specialized systems. Given a fixed set of algorithms, architectures tailored for the execution of those algorithms can be developed if the architectural requirements of the algorithms are understood. Third, a general method of relating algorithms and architectures will allow efficient use of reconfigurable parallel systems. Integral to the effective use of these flexible parallel systems will be the ability to select machine configurations based on knowledge about the algorithms to be executed. In order to accomplish this automatically, the operating system will need to use information about the characteristics of the algorithms to select successive configurations of the parallel architecture.

Using the application areas of image, speech, and signal processing as our frame of reference, we have identified a preliminary set of characteristics that relate to the mapping of parallel algorithms to parallel architectures:

Type of parallelism: data parallelism versus function parallelism
Degree of parallelism
Data granularity
Module granularity
Uniformity of the operations
Synchronization requirements
Data dependencies
Static/dynamic character of the algorithm
Fundamental operations
Data types and precision
Data structures
I/O characteristics

There are a number of issues associated with the problem of mapping algorithms to architectures: How robust are the algorithm characteristics across problem domains and architecture models? What algorithm representations will facilitate the automatic extraction of the algorithm characteristics? To what extent can the process of relating the algorithm characteristics to architecture characteristics be formalized? The problem of gaining an understanding of the relationships between algorithms and architectures is a critical one. The development of an effective means of describing the salient attributes of a parallel algorithm is one step in this process.

Highly Parallel Processor Arrays Can Be Inexpensive, Programmable and High-performance

H.T. Kung
Department of Computer Science
Carnegie Mellon University
September, 1986

Warp is a programmable systolic array machine designed by Carnegie Mellon. The machine has a linear array of 10 or more cells, each capable of performing 10 million 32-bit floating-point operations per second. Two wire-wrap prototypes, built by Carnegie Mellon and its industrial partners--GE and Honeywell, have been operational since spring 1986. These machines are being used for signal and vision processing and for scientific computing. For these computations, the new machines are typically several hundred times faster than the VAX 11/780. GE is under contract to build eight printed circuit board versions of the machine. Carnegie Mellon is also working with Intel in developing a VLSI version of the machine.

The Warp project has shown the viability of "programmable" systolic machines. More precisely, the project has demonstrated (1) a compiler capable of generating efficient code for systolic arrays; (2) algorithms and application software capable of making efficient use of large processor arrays; and (3) prototype hardware systems that can be reproduced economically.

Performance Evaluation Models for Parallel Programming

Ronald L. Larsen
University of Maryland

9-5-86

Recent progress in the development of highly parallel computing systems draws attention to the importance of effective programming techniques which combine the ability to advantageously use hardware parallelism while presenting a sufficiently natural and expressive power to the programmer. Research in this area has largely focussed on the language constructs required to virtualize the machine parallelism, delaying the binding of machine operations to processors until run time. Performance analysis of parallel programming techniques and language constructs at the macroscopic level has attracted rather little attention, with the result that few theoretical results or validated performance models exist for asynchronous concurrent programs.

Research at the University of Maryland is concentrating on techniques for better understanding the parametric behavior of asynchronous distributed programs. Most recently, attention has been focussed on predicting process synchronization overhead using source code analysis techniques. Parameters of interest include the process execution time, communication delay, process initiation sequence, and the synchronization architecture.

Recent results include the analysis of dual process, unconditionally synchronizing programs. This class of programs is shown to rapidly enter a small number of cyclic steady state modes. Parametric analysis indicates that these programs will stay in a single mode over extended ranges of parameters, but that critical values exist at which a mode change occurs, resulting in sudden changes in observed performance. The effects of observed mode changes are often counter-intuitive, requiring formal analysis to understand. As an example, we have found instances in which making the code for one process run faster causes the aggregate waiting time of all processes to increase.

Tuning parallel software for maximum performance appears to be a very complex, non-intuitive problem about which little is understood. Parallel programming language constructs supporting performance instrumentation and evaluation are needed. Performance modeling techniques for highly parallel machines are required to support language development and to develop programming styles targetted at fully realizing the potential of novel parallel architectures.

Paper: Jerrold Leichter
Yale University

Position Paper for the Yale Linda Group

Robert Bjornson
Nicholas Carriero
David Gelernter
Jerrold Leichter (workshop participant)

What kind of programming languages and environments should we be developing for parallel programming? Historically, we can discern several approaches:

1. Take a conventional language and add synchronization constructs, typically monitors.
2. Take a conventional language and add message passing. Often, it is the operating system that provides message passing as a set of system calls.
3. Develop a new programming style, and re-cast everything in its terms. Data-flow and functional languages are typical examples.

The Linda group's approach, while superficially similar to approach 2, in detail is quite distinct:

Take a conventional language and MERGE IN the Linda shared tuple space operators.

While the Linda operators can be added on top of an existing language, they imply a semantics that is closely coupled to the host language. For example, the Linda operators assume a type system that should be integrated with the host language's type system.

The advantages we see in this approach include:

- Writing parallel Linda programs is not substantially different from writing non-parallel programs in the host language. The programmer does not have to discard 25 years of software engineering experience.
- The Linda operators are very flexible, but a sufficiently clever compiler can make use of the lack of variety that appears in any given Linda program to produce good code. Further, our early experience indicates that the techniques required are very similar to those used in traditional optimizing compilers; that is, they are well understood.
- Where the compiler cannot discover good approaches on its own, it is often possible for fairly simple run-time algorithms to do so and make appropriate adjustments.

More generally, we have found the following:

- While the Linda operations are "higher level" than typical message-passing primitives -- thus raising the question of their appropriateness when combined with low-level languages such as C - it's been our observation that they are efficiently implementable, and that ultimately what matters for efficiency is careful tuning of the support system at a low level, not the high-level interface the programmer sees.
- Shared-memory systems are ideal for implementing Linda, and Linda seems to provide a

good way of programming them effectively. Further, shared-memory hardware seems to be very practical for an interesting range of machines -- up to perhaps a hundred CPU's.

- All sorts of interconnects, from buses to hypercubes, seem to be practical for hardware configurations of up to several hundred nodes. Linda is implementable on such systems as well, but hardware assistance for communications seems to be important. We've further noted that:

  - Existing hardware support is heavily biased toward message-passing.

  - Reliable broadcast mechanisms are often easy to build into hardware, but difficult to build in software on top of unreliable hardware mechanisms. There has been relatively little interest in this since few parallel programming environments provide broadcast services. Linda's tuple space operations are naturally viewed as broadcasts, and the existence of a good broadcast mechanism makes implementation much easier. This is true even if the system's effective broadcast bandwidth is smaller than its point-to-point bandwidth.

  The preliminary results of a project to implement a Linda support chip to be used with a bus-based, non-shared-memory system indicate that effective hardware assistance is practical and should produce very competitive performance.

The most important element we've missed so far is a database of measurements characterizing real-life parallel programs. It's impossible to design an optimizing compiler without some idea of what "cliches" are common in typical programs. We know what these are for sequential programs. The number of real parallel programs not closely wedded to specialized hardware is rather small, and the amount of data published on them is minuscule. We are just beginning to gather such data for Linda programs, and we would encourage the research community to do the same with whatever systems they are developing.

Position Paper

Workshop on Performance Efficient Parallel Programming

Joanne L. Martin
IBM T.J.Watson Research Center
Yorktown Heights, New York

One of the research issues that has been recommended for discussion concerns the contrasting (or comparing) of two parallel architectures in an application independent way. Although this sounds good on paper, I believe that this is an unrealistic goal. The performance of vector and parallel computers is dependent on the applications being executed and, to divorce architectures from applications would be to develop performance information too general to be useful.

A more realistic goal is to identify specific, measurable hardware parameters that affect performance and to relate these to established application characteristics. For example, Hockney has described the parameters $n_{1/2}$, $f_{1/2}$, and $s_{1/2}$ to indicate the overheads associated with using vectors, accessing memory, and exploiting parallel architecture. In combination, these architecture-dependent parameters begin to classify a system as to the type of application for which it is suitable. They do not, in themselves, provide a performance statement about the system. An $n_{1/2}$ that is very large may imply that an architecture will have difficulty on applications with small vectors, but it says nothing about its performance on scalar code. The performance of the system will depend on the characteristics of the workload being executed. Research on classifying applications and their various implementations can be pursued, and would be beneficial to system architects, performance analysts, and applications programmers. In this approach, classes of architectures and classes of applications would be identified and comparisons or contrasts of architectures would take place within the classes of architecture for specific classes of applications.

For example, computational fluid dynamics in its traditional implementations and magneto hydrodynamics are generally highly vectorizable applications (90 - 95%) that also exhibit a significant amount of large-grain parallelism (85 - 95%). In contrast, many Monte Carlo, circuit analysis, and quantum chromodynamics codes exhibit negligible amounts of vectorization but have extremely high (95 - 99%) parallelization. Different systems could be beneficially analyzed by restricting to one or the other of these classes of application.

Furthermore, a single system's performance will vary widely depending on the choice of implementation of an application. Consider the following two examples:

1. A production code at the Los Alamos National Laboratory

   Executed on a Cray-1, the net performance of one particular production code can be either approximately 20 MFLOPS or approximately 2 MFLOPS. The difference is the result of the choice of the path taken through the code at execution time. If the solution is accomplished via the hydrodynamics path, the higher performance is obtained. The lower performance follows from the Monte Carlo path execution.

2. A Navier-Stokes CFD problem.

   The standard solution is numerical and generally requires accessing data in successive passes through orthogonal directions, making parallelization speedup difficult on a non-shared memory system and vectorization difficult on a system that is sensitive to the presence of large strides. A new solution technique is being considered that involves cellular automata. This implementation is extremely parallelizable and has been shown to produce high performance on the Connection machine, an architecture on which the more traditional solution would be expected to perform poorly.

In summary, the proliferation of computer architectural models has magnified the range of performance possibilities considerably relative to the range possible on single processor serial machines. Because some classes of computational models exploit hardware characteristics that are

left untouched by others, careful selection of both models must be involved in the evaluation of these systems. Rather than application independent performance techniques, we need to develop the correct dependence of applications and architectures in order to best understand the overall performance of complicated systems.

PARALLEL PROGRAMMING

Position Paper for the Workshop
Performance Efficient Parallel Programming

H. Müehlenbein GMD
P.O 1240 D-5205 Sankt Augustin 1

INTRODUCTION:

Parallel programming cannot be discussed in isolation but has to be considered in the framework of parallel processing. In the past parallel processing was restricted to array or vector processing. Today parallel processing itself needs an interdisciplinary approach. Multiprocessors operating asynchronously and routing messages within a network have problems common to computer networks and distributed systems. This we can call the horizontal connection. Parallel processing also needs a vertical connection. Within parallel processing we have a multidimensional design space. We need experiments to investigate the relationship between programming languages and operating systems, between applications and suitable network topologies. In order to obtain a breakthrough in parallel processing an interdisciplinary approach is needed. There are at least three reasons to jump to parallel processing

performance
fault tolerance
simplicity

The last issue may be surprising because parallel programming is considered to be complicated. But as C. Mead asks: "How much of the problem is due to anachronism in languages and how much isdue to the way in which the parallel semantics has been applied?" Parallel programming needs a fresh start. "A fresh start often gives us insights into the structure of a problem which reveals an underlying unity and simplicity."

We claim that for certain applications the fresh start could be a uniform computational model based on communicating processes. To support this model sophisticated graph models have to be developed and applied. We will discuss this topic in more detail in the following chapters.

SIMPLICITY

There is a widespread believe that parallel programming is a difficult task. We believe that there can be a parallel programming style which can lead (for certain applications) to simpler programs than sequential ones. The basic observation is as follows: Many systems in nature work in parallel. With the right programming paradigmn it should be easy to obtain a one-to-one mapping of the problem structure to the implementation structure. This paradigm can be phrased "macro data flow oriented and object oriented". It is interesting to observe that this style has been advocated in different disciplines. The following table gives some examples.

Discipline	Example
science	cellular automaton
software engineering	SA, dataflow graphs
languages	communicating processes
operating systems	message passing
VLSI	VHDL behavior desription

In all these examples we describe how the input and output of an entity relate, by expressing the associated input/output transformations and by connecting the input/output parts. Thus the system is described or constructed from a local point of view. Using on every level a similar model should make the programming task easier. In structured analysis for instance the system is described using a hierarchy of data flow diagrams so that non-specialists can understand what is happening. why not viewing it as a maximally parallel set of processes communicating via well defined data flows ? This means that a language based on this pronciple can directly implement the specification ans is a great improvement over sequential language, where the first step is to turn data flow diagramms into a sequential module calling hierarchy. There are also strong arguments to use just the opposite programming paradigmn - we will phrase it "functional" or "structure oriented". Both styles are integrated in the VLSI design language VHDL whereas in parallel processing they are considered to be totally different computational models and should not be combined. It looks as if there are almost wars between these two schools...... We believe that both styles are necessary and have advantages in different application areas. Within the area of scientific computation we advocate the use of the data flow approach, which can be shown to lead to simple programs and massive parallelism. The drawback of the dataflow approach is that the global behavior of the system cannot easily be observed or predicted. All activities which require a total system state like debugging, dead lock handling, checkpointing etc . are complicated to implement.

Fault Tolerance

Multiprocessor systems allow fault-tolerant execution of programs. A variety of graph models has been developed for diagnosis and reconfiguration of multiprocessors in the case of hardware failures. Test results obtained locally can be routed through the system to allow the remaining intact processors to compute a system-wide self-diagnosis. System-wide self-diagnosis enables the system to reconfigure and to run the application programs on the remaining intact components. This problem can be solved by isomorphic embedding of graphs or by special scheduling. This demonstrates that models incorporating fault tolerance are a natural extension of the macro data flow model.

Performance

The performance gain of parallelism is constrained by how much parallelism is in the problem. Basically, we see at least two kinds of parallelism:

algorithmic parallelism
spatial parallelism

The first group contains applications where the parallelism has been obtained by decomposing the algorithm into a number of simpler components which can be executed in parallel. The second group contains applications where the parallelism has been obtained by distributing the data to be processed between a number of processes in such a way that the geometrical structure of the data is preserved. Algorithmic parallelism is not easy to explore and seems to be highly irregular whereas geometric parallelism is proportional to the problem size. The performance problem of spatial parallelism is well understood.The performance is more or less influenced by the ratio of communication time and compute time. Kruskal stated the following conjecture: With rare exceptions, any real life problem can execute efficiently on any reasonable parallel computer - as long as the problem is large enough.

This conjecture can be proven informally by the following observation. With M objects stored locally we often have O(M**i) compute time and O(M**j) communication time. When j<i we can adjust the quotient of compute and communication time. The important question therefore is: How large does a given problem have to be to run efficient (50%) on a parallel architecture. With P equal to the number of processors and N equal to the problem size, we can classify the problems into three classes at least for the ring machine.

N = O(P)
N = O(P**k)
N = O(exp(P))

Conclusion:

For spatial parallelism there seems to be a uniform computational model to obtain efficient, fault tolerant and simple programs. To support this model new tools have to be developed or integrated into a coherent framework. These tools have to be based on formalized graph models. Hopefully we end up with a set of similar graphic oriented tools supporting program specification, program implementation and program visualization at runtime. This programming model is neither revolutionary like functional models nor evolutionary like parallel languages with parallel loops. It needs reprogramming in a clean and safe style. In our opinion supercomputers with more than 100 gflop should not be driven by the old dusty FORTRAN deck.

Performance Critical Decision Problems in Parallel Scientific Computations

David M. Nicol

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center, Mail Stop 132C
Hampton, Virgina 23665

Scientific computations are often composed of numerical calculations at each point in a discretized spatial (or transformed) domain. Workload assignment for parallel processing on message passing architectures involves partitioning the discretized domain points into *regions* which are then mapped to processors; usually the number of regions equals the number of processors. A processor's workload is then a function of the domain points in the region it receives. Our work has focused on two problems arising from this type of computation. One problem recognizes that a computation's workload distribution may change in time (or iterations). This is especially true for adaptive methods, which dynamically create and destroy domain points. The second problem recognizes that performing a convergence check after every iteration is unnecessary, and degrades performance.

Our initial efforts in treating the dynamic workload imbalance problem involved development and study of analytic models which exhibit time variant behavior. These models fall into two classes; one class models situations where run-time performance changes abruptly; the other class models situations where performance declines gradually, and continuously. The key issue for all of these models is to decide when and if a new partition of the domain should be calculated and implemented. Intuitively, this decision should depend on the cost of remapping, the performance gain achieved by remapping, and the performance decline suffered by not remapping. In [1] we examine a abrupt change model, develop an optimal decision policy for the model, and then show that a simple decision heuristic which requires no estimation of seemingly critical model parameters achieves nearly optimal performance. In [2] we examine two gradual change models, develop a decision heuristic which attempts to minimize the overall cost per unit time, and show that this heuristic is effective for both models. Currently we are beginning to implement these policies for specific numerical problems on hyper-cube type architectures. Our overall goal is to develop remapping decision mechanisms which are transparent to the programmer.

Naive convergence checking of an iterative numerical method requires that every processor report at every iteration whether its subdomain has converged. The computation stops only when the solution has converged globally, at every subdomain on the same iteration. The overhead cost of convergence checking can be quite high: we have measured it (on an SOR solution of the heat equation) running on the Intel iPSC to be as high as 50% of the running time [3]. In [3] we discuss two means of reducing this overhead. One method dynamically schedules the next convergence check, allowing for the possibility of not testing convergence at all during intervening iterations. This method is akin to our remapping heuristics in its effort to balance the cost of checking convergence against the "overshoot" cost of continuing to calcuate iterations after global convergence occurs. The second method does not explicitly balance these costs; rather, it requests convergence information only when certain necessary conditions for global convergence are satisfied. For the problems we studied, these methods performed equally well.

The research reported here recognizes that the dynamic behavior of parallel computations gives rise to decision problems which must be dealt with if the computation is to run efficiently. Ideally, our treatments of these problems must become invisible to the application programmer; the decision

mechanisms we develop should be implemented at a system level, rather than an application level. We are continuing our efforts to achieve this goal.

[1] D. Nicol, P. Reynolds Jr., Dynamic Remapping Decisions in Multi-Phase Parallel Computations, *ICASE Report No. 86-58*, September 1986.

[2] D. Nicol, J. Saltz, Dynamic Remapping of Parallel Computations with Varying Resource Demand, *ICASE Report No. 86-45*, July 1986.

[3] J. Saltz, V. Naik, D. Nicol, Reduction of the Effects of Communication Delays in Scientific Algorithms on Message Passing MIMD Architectures, to appear in *SIAM Journal of Scientific and Statistical Computing*, January 1987.

Position Paper for the Workshop on Performance Efficient Parallel Programming

David Notkin
Department of Computer Science, FR-35
University of Washington
Seattle, WA 98195
(206) 545-3798
notkin@washington (arpanet or csnet)

September 1986

Programming environments improve the productivity of programmers in two ways. First, environments relieve programmers from concern with details not strictly related to programming. For example, structure-editing environments ensure that programmers need not be concerned with whether a semicolon is a statement separator or terminator. Second, environments can enforce or encourage the use of a specific methodology.

Most of the research and development of programming environments has primarily focused on sequential systems and languages such as C and Pascal. The motivation for environments, however, carries over even more strongly in the domain of parallel computation. The details required for parallel programming are immense: without environmental support, it seems nearly impossible to produce high-quality parallel programs. Also, since the search space for parallel programs that solve a problem seems to be greater than that for sequential programs, suitable methodological guidance will be needed.

(As an aside, I predict that most of the software engineering problems of documentation, maintenance, parallel programming-in-the-large, and such, will come to the forefront of parallel programming not too far in the future. As soon as there are many problems that require multiple programmers and long periods of time to solve, we can expect our focus to shift from environments for constructing parallel programs to environments for engineering parallel programs.)

It is difficult to perform research in programming environments (either sequential or parallel). The key reason for this is that modifying the environments to meet changing needs and requirements is costly. To test a new idea often takes significant resources and time. One approach to reducing this inflexibility is to construct environments with their modification and enhancement in mind. Several sequential environments have taken this approach in varying degrees. Consider the single language programming environments such as Interlisp and Smalltalk. The

uniformity and open nature of these environments encourage experimentation. Also consider the generation of environments based on structure-editors, such as Gandalf and the Cornell Program Synthesizer. These generators permit us to construct environments for various languages with relative ease.

Given the relative immaturity of parallel programming (with respect to sequential programming), it is no surprise that the first efforts in supporting parallel programming were quite inflexible. Adding new interconnection schemes is an example of the kinds of change that environments have difficulty accommodating. Now that the "first-generation" support systems are maturing, we have an opportunity to focus on the dimensions along which we believe that environments must have flexibility (so that we can experiment even further).

In what areas do we need to support such flexibility? Here are two examples:

Object Definition and Manipulation Parallel programming environments do and will rely heavily on non-textual objects, such as graphic versions of communication graphs. If one is constructing an environment on top of a systems such as UNIX, the underlying file system does not facilitate the storage and manipulation of objects that are defined non-textually, since the file systems are oriented towards streams of bytes. It is not likely, though, that replacing the byte-stream support with a single other abstraction will work, since non-textual objects other than graphs – for instance, abstract syntax trees for representing programs – may be needed as well. We are searching for a unified way in which to define, manipulate, and coordinate various classes of objects.

Tool Definition Parallel programming environments play many roles: communication graph editor, serial program editor, compiler, debugger, simulator, among others. The parts of the program that implement these varied roles are usually closely integrated so as to support the programmer as fully as possible. Despite this benefit to the user, the tight integration often makes it difficult to introduce new tools. We are searching for ways in which we can define and integrate new tools into an environment at reduced cost. This will necessarily increase our flexibility in experimenting with the environment.

As part of my research with Larry Snyder, I am interested in solving these problems along with others that currently inhibit the flexibility of parallel programming environments.

Position Paper

FINDING THE RIGHT VIRTUAL MACHINE FOR PARALLEL APPLICATIONS PROGRAMMING

T.W. Pratt
University of Virginia

September 1986

Two questions of importance to performance efficient parallel programming:

1. What 'virtual machine' should be provided to the applications programmer?

2. How independent of the underlying hardware architecture can this virtual machine be if you still want good performance?

The 'virtual machine' that the applications programmer uses is formed out of the various layers of software on the system: programming language, run-time library, operating system, plus the hardware itself. In designing this software, the designer considers what aspects of the machine and lower layers of software to hide, and what to augment. The applications programmer uses this virtual machine.

The virtual machine may match the architecture closely (and applications programs will then be difficult to port to other architectures), or it may hide the architecture completely. Jones and Schwarz [1] note that if the virtual machine hides aspects of the underlying architecture that have important performance implications, then obtaining good performance may be difficult for an applications programmer.

Typical design questions:

1. Should the VM provide message passing if the HW provides shared memory?

2. Should the VM provide shared/global data objects if the HW has only distributed memory?

3. Should the VM provide several granularities of parallel actions if the HW provides only one grain size?

4. Should the VM be a clustered machine if the HW is not clustered?

THE PISCES PROJECT

1. Carefully defined virtual machine

2. Architecture independent applications programming (scientific/engr)

3. Programmer control of virtual machine = => hardware mapping

4. Virtual machine model:

clusters of tasks
dynamic task creation/termination
message passing among tasks
forces (a la Harry Jordan/U.Co.)
-- shared variables/parallel loops
multiple granularities of parallelism

Implementation: FLEX/32, workstation network, hypercube (planned)

REFERENCE

[1] A.K. Jones and P. Schwarz, "Experience Using Multiprocessor Architectures - A Status Report," ACM Computing Surveys, Vol. 12, No. 3, June 1980, pp. 121-166.

Tom Reinhardt

Background

Presently, the Message Passing Semantics Group at the M.I.T. Artificial Intelligence Laboratory is pursuing the design and implementation of Actor languages and architectures for open-ended, continuously evolving systems. i.e., *Open Systems.*

Actors are opaque, autonomous computational agents that communicate via message passing. In response to a communication. an actor may perform several actions:

- It may send more communications to its *acquaintances*:
- It may create more actors; or,
- It may designate a new behavior with which to process the next message.

Note that any of these actions may be performed jointly or severally.

Actor communications systems ressemble mail systems, i.e., communication proceeds asynchronously, and actors queue incoming messages if they are delivered while processing. Computation therefore occurs between communications. Moreover, parallelism is engendered by creating more actors or by creating more communications.

Current Experimentation

Current research proceeds along the following routes: Architecture, Programming Languages and Semantics, Theory and Applications for Open Systems.

Theory

Actor theory rests on a firm mathematical foundation developed and elucidated in Gul Agha's dissertation and forthcoming book, *Actors: A Model of Concurrent Computation in Distributed Systems.*† Continuing theoretical work concerns the nature of high level actor systems, such as inheritance and description systems, as well as the impact of the actor formalism on algorithm design.

Architecture

A potential architecture for actor machines has been simulated on the group's Symbolics 3600 Lisp Machines. Basically, each Lisp machine is configured with a number of *Workers* which are actors that enqueue tasks as they arrive and then execute them upon the receipt of a simulated cycle or tick message.

†Published by M.I.T. Press, this fall.

A collection of these workers distributed over one or more Lisp Machines is called an *Apiary*.

More recently, work has begun on the design of a processing chip that incorporates message passing at a fine-grained level. We are currently metering sample applications on the Apiary and are utilizing these statistics in designing and developing chips for actor machines.

Languages

Actor languages attempt to unify the inherent parallelism of pure lambda calculus with the ability to perform localized state change operations. This unification of functional and imperative programming styles results in a system that couples maximum concurrency with the ability to perform history sensitive computations.

To date, the following languages and systems have been implemented and are being utilized in our research effort:

- A Primitive Actor Language, *Pract*, based upon the notions of Actor automata developed in Agha's dissertation. Pract represents the Apiary kernal language;
- A Core Actor Language, *Acore*, which serves as the applications language;
- A debugging facility, *Time Traveler* that permits users to step through computations by single events or by entire transactions. In addition, *Traveler* provides *Biographies* which essentially represent the history of activity for a particular actor.

Future developments in this area include the design and implementation of inheritance mechanisms within Acore and A.I. applications.

Algorithms

The design and evaluation of algorithms has heretofore concerned itself with the number of steps an algorithm requires to compute a result given some data. In our experimentation with Actor systems, several questions have emerged.

The most general of these stems from the observation that maximizing parallelism doesn't insure optimal algorithm design.

At a pragmatic level, actors can be distinguished as those that change their behaviors as result of a communication, *serialized*, and those whose behaviors never change, *unserialized*. In a collection of unserialized actors, the speed at which an algorithm computes is a function of the number of steps required and the communication latency between actors. Because serialized

actors change state, however, they must remain locked until they've processed the incoming communication, in order to maintain consistency. *These serialization points can become bottlenecks.*

Hence, a fundamental question arises: Is there an optimal tradeoff between unrestrained parallel activity and administrative overhead, i.e., serialization? And, assuming there is, is it determined *ad hoc*, or are there guidelines that might be followed by designers?

In grappling with this question, theories from sociology and organizational management theory might provide fruitful discussion.

IN PARALLEL PROCESSING, THE TRIVIAL BECOMES CONSEQUENTIAL AND THE NEGLIGIBLE BECOMES DOMINANT

David P. Rodgers
SEQUENT COMPUTER SYSTEMS, INC.
Beaverton, Oregon

PARALLEL PROCESSORS WILL BECOME COMMONPLACE AND ESSENTIAL

Mirroring the commercial proliferation and user acceptance of virtual memory systems which began in the early 1960's, parallel processor systems are becoming available as standard commercial products and applications are being developed which depend on their parallel nature. The demand for parallel processor systems is driven by needs for absolute performance in excess of that attainable through circuit improvements, cost/performance to economically justify new applications, scaleable growth of execution vehicles to match growth in application size, fault tolerance through redundancy to protect application productivity and reduced support costs through commonality of spares and training across application engines.[1] As commercial machines proliferate, a body of applications and application approaches will develop which are feasible only in a parallel processor environment. Similarly, languages which express parallel computational ideas with facility, eg. Prolog, ADA, will supercede present application languages.

SHARED MEMORY PARALLEL PROCESSORS WILL BE PRODUCTIVE IMMEDIATELY

The dominating issues which must be resolved by designers of parallel processors are scaleability, processor-to-processor bandwidth for data sharing and inter-process (task, thread, computation) synchronization time. While there are a range of possibilities for tightness of coupling and degree of connectedness among processing elements, research work and commercial development seem to be focussing on three architectural styles: multicomputer clusters (networks), connection machines (including hypercubes) and shared memory multiprocessors.[2] Clusters generally are most scaleable but suffer from low performance in sharing data and synchronizing computation. Connection machines have a wide range of scalability and have better communication and synchronization facilities than clusters but these are useable in only in very specialized applications. Shared memory multiprocessors provide the maximum performance processor-to-processor communication and synchronization but are limited in scaleability (at present) to a few tens of processors. What are the factors which will make shared-memory multiprocessors most successful immediately? In a word: tools. To make effective use of parallel processors, program development environments which make use of expert knowledge of both the problem structure and the machine structure will be required. Compilers which restructure user applications within the scope of one or a few modules are presently available. Compilers which can expand the scope of restructuring to whole programs will emerge within the next year. Programming environments which interact with the application

developer to extract the structural knowledge lost in translation to programming language are being designed. All of these sophisticated tools depend on low cost data sharing and rapid inter-computation synchronization.[3]

The shared memory multiprocessor also provides a complete development environment. The complete set of conventional programming tools which exist for uniprocessor environments can be hosted and used to bootstrap the more sophisticated parallel programming environment. Architectural exploration can be done with simulation, often with better performance because of the superior communication and synchronization facilities.[4] If economics dictate, multiple workers can share use of a single parallel processor without resorting to time reservation. Performance monitoring and measurement can be done with sampling techniques at a very fine level of detail without greatly disrupting the flow of the application.

OLD RULES OF THUMB MUST BE REVISED

In parallel processing, the trivial becomes consequential and the negligible becomes dominant. Since the advent of virtual memory and caches as elements of the storage hierarchy, memory reference patterns have been important determinants of performance. In a parallel processor, references to shared data becomes a critical performance factor. Locality is not determined solely by the execution of a single process but by all the cooperating processes. The consequences of an inappropriate migration of an object from primary to secondary storage may be that all the processors in an ensemble wait. Note that this isn't a new problem, it exists for operating systems managing channel processors, just new to application programmers and compiler writers. Similarly, treatment of abnormal or rare conditions in the parallel programming environment requires revised treatment. As the execution rate of the application increases so does the rate of exceptional occurrences. If the processing of an exception blocks the execution of the application, the cost is multiplied by the parallelism factor. Rules of thumb about where a program will spend most of its time will be upset by parallel processors necessitating new analytical tools.

PERFORMANCE TOOLS SHOULD PROVIDE AN ABSTRACTION

Restructuring compilers will dramatically alter the static expression of an application. Parallel processors will dramatically alter the dynamic behavior of an application. Both of these factors will break the tenuous connection between the programmers understanding of the flow of his program and the actual execution. The stages of program development will remain the same: design, implementation, testing (debugging) and performance tuning. What must change is the level of abstraction used to understand whether the intent is correctly expressed and the execution is robust and effective. Guages have been suggested as a vehicle for shifting the emphasis from evaluation of the flow of control to the flow of data.[5] For problems expressed in object oriented programming languages, these may prove effective. For applica-

tions carried forward from earlier computer generations, some way of mapping the actuality onto the original expression may be necessary. In either case, domain specific expertise about the application and programming environment must be built into the development tools.

REFERENCES

1. Forest Baskett and John L. Hennessey, Science 231, 963 (1986).

2. Omri Serlin, Supermicro newsletter, April 1986.

3. David J. Kuck, Edward S. Davidson, Duncan H. Lawrie, Ahmed H. Sameh, Science 231, 967 (1986).

4. Robert L Brown and Peter J. Denning, "A Comparison of Multiprocessors: Sequent Balance 8000 and Intel iPSC Hypercube", RIACS Technical Report 86.5 (1986).

5. Daniel G. Bobrow and Mark J. Stefik, Science 231, 951 (1986).

Workshop on Performance Efficient Parallel Programming
Position Paper (Aug. 1986)
Larry Rudolph (Hebrew University)

- The main goal of parallel processing is fast execution (i.e. linear speedups).

- Large problems, those requiring lots of computer power, are the main application domain of parallel processing.

- A linear speedup can almost always be achieved given a large enough problem.

- Performance degredations are often due to contention for shared resources (e.g. shared memory modules) and synchronization overheads.

I would like to suggest examining the generalized parallel prefix construct as a way of partially achieving the above goals. It has applicability to programming at a high level, as a target for transforming specifications intocode for a parallel machine, and as a basic operation that should be supported by any parallel architecture.

The operation is as follows: Given a binary operation, φ, and a set of items, a_0, a_1, ..., a_{n-1}, the i-th prefix is defined to be $a_0 \varphi a_1 \varphi ... \varphi a_i$. The items can either be

1. stored statically in an array so that the i-th item is in cell i

2. stored as a linked list -- the i-th item is found by following the link from the i-1 item.

3. defined asynchronously so that the i-th item is the one that is ready to be processed after i-1 other items are processed.

- The parallel prefix construct has efficeint implementations on (i) high bandwidth shared memory machines with multiple memory modules and on (ii) fixed connection machines such as the hypercube.

- The mapping from shared memory to fixed connection can be done automatically.

- Parallel prefix allows an operationto be applied to a whole set of items at the same time. The intermediate results are often useful in subsequent steps of the computation. For example, it has been found applicable for solutions to:

    - sparse matrix multiplication

    - graph algorithms

    - unification

For such applications, the binary operation is not as simple as addition; in fact, it can often be quite complex and still have an efficient implementation.

Position paper for Workshop on Performance Efficient Parallel Programming.

Vijay Saraswat
Department of Computer Science
Carnegie-Mellon University

I am working in the area of design, semantics, implementation and use of concurrent logic programming (CLP) languages. As far as the topics of concern to this workshop, I am interested in parallel (especially 'AI') languages and in performance efficient mappings of parallel algorithms into parallel architectures, via a compiler for such a language for the given architecture.

CLP languages (e.g. Concurrent Prolog, Parlog, GHC and CP, the language I work with) are rooted in a desire to make some of the parallelism inherent in the Horn clause logic programming approach available and manipulable by the user. These languages offer a notion of recursive, non-deterministic networks of light-weight processes where parameter-passing happens via unification (which depending on the level, can be thought of as just pattern-matching or a special kind of pointer manipulation in a once-only-assignable framework).

From a programming language viewpoint, the new programming paradigm that CP offers is that of concurrent, controllable constraint systems. The user can set up his problem declaratively as a network of objects with some constraint relationships and specify the kinds of assumptions that can be made when constrant-propogation stalls. Moreover, in this framework, he can specify some control information which can guide the underlying search in a crucial way. This approach yields some novel algorithms for solving such classic combinatorial problems as the N-queens and the map-coloring problems.

While such languages have proved to be excellent tools for expressing, in software, complex systems of communicating processes, there is yet no clear understanding of how multi-processor implementations should look like. The problem is that each 'cycle' of a CP machine involves an atomic distributed commit operation, that can be rather costly to implement. Moreover 'channels' of communications (logical variables) are, in the language, very easily created and 'unified' (made the same) so that it would seem impossible, in general, to guarantee the locality of communication of a piece of program, without doing extensive data-flow analysis. Such schemes have yet to be developed.

I would be very interested in hearing a discussion, from those who have thought of it, of these and similar issues involved in efficient multi-processor implementations of general-purpose CLP languages.

My own current thinking is that a general implementation, even of a language like Flat CP is going to impose too much extra overhead. For instance, unless a compiler does very sophisticated analysis of the user program, it may not even be possible for it to detect that some program structure essentially represents a software pipeline and could hence be mapped onto a linear sequence of phase-shifted synchronous processors. There is a need for identifying some restrictions on the basic model which would allow programs satisfying those restrictions to be efficiently implemented on

conventional shared-memory architectures. As a trivial example, all Flat CP programs satisfying the syntactic property that for any goal of interest there is at most one committing clause can be efficiently implemented, by assigning one virtual processor to each goal in the current resolvent. One can think of specifying with a program, some of the properties of the communication structures inherent in the program in such a way that an implementation can make use of this knowledge directly to select an appropriate process/processor mapping.

I would be glad to discuss this and related issues in more detail at the Workshop.

Developing High-Performance Parallel Software for Real-Time Applications

Karsten Schwan

Computer and Information Science
The Ohio State University
2036 Neil Avenue
Columbus, OH 43210

schwan@ohio-state.arpa

Abstract

Complex electro-mechanical systems require computational speeds and reliabilities far beyond the capabilities of current, embedded computers. Therefore, the uses of computers and software offering substantial parallelism are becoming essential. High-performance, parallel software for such real-time systems must be flexible in functionality and adaptable in performance and reliability, and thereby be able to accommodate changes in technology and in application requirements. Flexible and adaptable real-time software requires substantial programming and operating system support.

The structure and content of this support are being investigated in the PArallel, Real-Time Systems (PARTS) Laboratory at The Ohio State University. The laboratory is undertaking the development of a real-time systems testbed, of sample, real-time applications in cooperation with the ASV DARPA robotics project, and of novel, integrated programming/operating system technology. The purpose of these developments are to create software technology that assists programmers in constructing parallel, real-time software, in adapting such software to realize performance goals, and in making adaptation decisions. This paper describes the prototype integrated programming/operating system constructed within the laboratory. Two sample adaptations of real-time software illuminate the functionalities of the different components of the system.

On Efficient Large-Grain Parallel Processing

Herb. Schwetman

Microelectronics and Computer Technology Corporation
P.O. Box 200195
Austin, TX 78720

(512) 339-3428

Introduction

Large-grain parallel processing is based on the parallel execution of groups or blocks of instruction on multiple processors. In many cases, these executing blocks of instructions are called processes. Thus, large-grain parallel processing can be cast in terms of the parallel execution of processes, with the understanding that the user (programmer) has control of the creation and synchronization of these processes.

The efficiency of process level parallel processing depends on several factors, including memory contention, synchronization delays, and process management overhead. This latter factor can exert a significant influence on efficiency. In fact, this factor is what determines the smallest "grain size" which can be efficiently executed by a parallel system. In other words, because of the impact of the overhead associated with creating and managing parallel processes, there is some minimum size (as measured in length of execution interval) process such that if a process is "smaller" than this minimum, it is better, from a performance viewpoint, to execute the process serially, than it is to start up a parallel process.

This note offers the following points for consideration:

1. it is desirable to build systems which can efficiently execute smaller sized grains of computation (so as to enlarge the class of computations which are suitable for parallel execution),

2. understanding both the factors contributing to process management overhead and the resulting effects on program performance is important to the design and implementation of cost-effective parallel systems.

3. we desperately need better tools for examining process management overhead in current systems and current parallel programs, and

4. we will eventually need better hardware to support efficient execution of process-level parallel computations.

Position Paper

Zary Segall
Carnegie-Mellon University

We will generically call the manifestation of parallel programs inefficiencies - the parallel program performance degradation or shortly the performance bottleneck. Such parallel performance bottlenecks may be rooted on any or all levels of parallel program development and execution. The main goal of performance-efficient parallel programming is to minimize or even exclude all together the effects of the performance bottlenecks.

One could group the approaches used to generate performance-efficient parallel programs into the following three categories:

- Performance Bottleneck Prevention: This category includes techniques such as:
  - o Algorithm performance prediction.
  - o Parallel implementation performance prediction.
  - o Language constructs for prevention.
  - o Programming environment tools support for prevention.
- Performance Bottleneck Detection: This may include programming environment performance debugging tools such as performance monitors, graphic user interfaces, etc., as well as, compiler-based performance bottleneck detection.
- Performance Bottleneck Avoidance: This category deals with dynamic (run-time) compensation for detected or trend-predicted bottlenecks. Examples of such tools and techniques are:
  - o Language run-time support for performance bottleneck avoidance.
  - o Operating system based performance bottleneck avoidance (i.e., load balancing, dynamic resource allocation, etc.)

At this workshop I would like to discuss some of the following issues:

- Nature of performance bottlenecks in parallel programming.
- Techniques for performance bottleneck prediction, detection, and avoidance.
- Programming environment tools combining performance bottleneck prediction, detection, and avoidance.

Workshop on Performance Efficient Parallel Programming

H. J. Siegel
PASM Parallel Processing Laboratory
Electrical Engineering School
Purdue University
West Lafayette, IN 47907
September 1986

List of Topics I feel are important to pursue INCLUDES (in random order):

mapping algorithms and architectures - models of algorithms and architectures and how they interact -- what are the salient aspects of architectures and algorithms for the mapping problem; given parallel architecture - which algorithm approach best; given algorithm - which architecture best; given a reconfigurable system - which configuration/algorithm pair best.

parallel programming - what language features needed for efficient "explicit" specification of parallelism -- including processor transfers, processor enabling (SIMD mode), switching between SIMD and MIMD modes in reconfigurable systems, specifying subtask parallelism; what features needed for effectively compilable "implicit" specification of parallelism; portable parallel languages for sharing work; common methods for expressing parallel algorithms so that researchers can share and communicate results among themselves more easily; tradeoffs between the efficiency of writing machine dependent "explicit" specification of parallelism programs vs machine independent "implicit" specification of parallelism; developing and documenting a set of parallel programming techniques; using built-in operating systems functions to go from machine independent to machine dependent code; tradeoffs between extending existing languages (e.g., parallel C) and developing completely new languages (e.g., Tranquil).

impact of architecture on language - shared memory vs local memory; type and speed of interprocessor communication mechanism available; bit serial processors vs bit parallel processors (SIMD); number of processors; hardware support for operating system functions called by language; size of memory; type of architecture -- SIMD, MIMD, reconfigurable SIMD/MIMD, pyramid, dataflow.

Lessons From Poker

Lawrence Snyder

University of Washington

The Poker Parallel Programming Environment was begun in January of 1982 and has been distributed since October 1985. With feedback from a growing user community and some time for reflection, it is possible to identify certain features of Poker that have contributed to its demonstrated efficiency and portability.

The chief contributors to its efficiency and portability are (1) the fact that the model of computation used is a nonshared memory model of parallel computation, (2) the fact that the parallelism is specified explicitly, and (3) the fact that the communication structure is given explicitly. Because these features constitute a greater burden to the programmer, the environment must provide greater support.

In addition, Poker exhibits several other features worthy of inclusion in other systems. Poker makes extensive use of synthetic graphical pictures to simplify program specification. Poker demonstrates that it is possible to have language constructs with no syntactic form. Finally, Poker demonstrates the value of including a simulator of a parallel machine as an integral part of the environment.

Parallel Processing Research at Harris Corporation

Thomas L. Sterling
Advanced Technology Department
Harris Government Systems Sector
Melbourne, Florida 32902

Who We Are

Harris is engaged in a modest multi-year research program to investigate the potential and methods of parallel processing for general computing. We have a cooperating relationship with the Concert research project under the direction of Prof. Robert H. Halstead at the MIT Laboratory of Computer Science. Also, we are a participating company in the Microelectronics and Computer Corporation's Advanced Computer Architecture research program. Research activities of our group include

1) the Concert Multiprocessor Testbed,
2) the Multilisp programming language for symbolic computing,
3) the Simultaneous Pascal programming language for conventional applications,
4) the Yarc scalar static dataflow computer for signal processing, image processing, and simulation,
5) the Propel parallel production system for expert systems, and
6) the SPoC multiprocessor execution environment for effective general purpose parallel processing.

The following brief discussion of some of our near term goals, approaches, and views reflect the perspective derived from the SPoC project. Although we, like many, are engaged in multiprocessor research, our long term policy is to diverge from this path due to limitations of the multiprocessor as is also discussed.

Near Term - effective application of multiprocessors

Objective:

Tightly coupled, medium scale multiprocessor for general purpose applications employing a shared reference space and dynamic scheduling.

Issues:

Our research is currently focusing on a fully integrated implementation of the SPoC parallel execution environment from the application domain down to the realm of hardware. The purpose of this is to study each aspect of parallel computing in the context of the other supporting parts. In particular, we feel a need to get away from studies based on toy programs, simplified analysis, and incomplete simulations. We want to investigate parallel system behavior under the forcing function of real world computing profiles to observe the intricacies of interaction between system levels in terms of sensitivity of one level to changes in another.

The primary issues of concentration are

1) semantics of parallel programming,
2) low level synchronization mechanisms,
3) sources of performance loss including overhead, contention, starvation, and latency,
4) dynamic distribution of activities and objects, and
5) the quantity and quality of parallelism in applications.

Policies: the SPoC approach

programming- We believe that the programmer has to be aware of parallelism in his application to select algorithms well suited to parallel execution, but should not have to deal with machine implementation dependent aspects of program processing. SPoC supports explicit parallel programming using a concurrent thread computing model. Threads are segments of sequential code. Active threads do not interact and a processor assigned to a scheduled thread relinquishes the thread only upon its completion. This eliminates the overhead for suspending tasks but restricts the style of programming. Simultaneous Pascal is a superset of Pascal that reflects this model. It includes simple extensions such as fork-join and forall statements, as well as, a locking discipline for compound atomic operations and fine tuning of variable scoping. Simultaneous Pascal is overly constrained in the way it represents parallelism, resulting in hour-glass like parallel computing profiles but is well suited for medium granularity execution. All scheduling is done dynamically by the underlying run time multiprocessor system in a first come first served bases as processors become availability. Ordering is not guaranteed and scheduling is unfair.

synchronization- An effective parallel execution environment must meld the semantics of parallelism delineation with efficient mechanisms for parallel flow control. SPoC synchronizes the termination of concurrent threads in Simultaneous Pascal with the rendezvous control mechanism that allocates counters at run time for each join to be performed. Since parallel statements can be nested, all of the counters are linked in a tree structure that reflects the dynamic state of statement nesting during execution.

locality- We are experimenting with ways to use the hierarchical organization of the Concert Multiprocessor and the natural locality of runtime program execution to minimize contention for shared physical and logical resources. While small multiprocessors may use cache techniques with shared busses, very large systems will have to resort to distributed methods that exploit more knowledge of program behavior. SPoC provides the empirical context with which to investigate such methods.

granularity- The overhead for scheduling and context switching of a thread is to a significant degree insensitive to the size of the thread. For moderate size multiprocessors, too little parallelism will result in performance degradation due to starvation. However, too much parallelism can undermine performance when the thread size is equal to or less than the work required to manage it. Aggregation is a technique being studied with SPoC for reconstituting fine grained concurrent threads into fewer coarse grained threads to reduce the total overhead incurred. Both compile time and runtime methods are being considered with the latter complicated by the fact that it is itself a form of overhead.

instrumentation- In an experimental environment such as ours, being able to "see" what is going in is critical to testing of hypothesis and analyzing behavior. For parallel processing in general, the complexities of action and consequence may demand that instrumentation become a standard tool of program development. SPoC embodies hardware support for instrumenting both hardware and software behavior during parallel execution.

Long Term Policies- fundamental flaws of the Multiprocessor

The multiprocessor is a parallel computer of some convenience because of the relative availability of its primary constituent element, the VLSI microprocessor. To some, it is the intuitively obvious parallel architecture based firmly on extensive experience with von Neumann uniprocessors and multiprogramming operating systems. Therefore, it is at some risk that I suggest that the multiprocessor as a parallel computer architecture is fundamentally flawed and that a new (or at less least conspicuous) architectural model be pursued. I make this assertion based on the observation that implicit in the concept of the multiprocessor are three underlying assumptions which, I submit, are false.

A computer, sequential or parallel, is a physical embodiment of a set of mechanisms that together support the execution requirements of the instruction set architecture and, in turn, programs written in or compiled down to the ISA. The amount of each mechanism (measured in time or real estate or power, units your choice) required to fulfill the needs of the abstract program is a strong function of the computational model that both the programs and the computer reflect. When the model between the program and the computer differs dramatically, software patches are used to emulate the program model with that of the computer, making imperfect use of resources due to the mismatch. I believe such a mismatch is inherent in multiprocessing.

The three assumptions of multiprocessing with which I take exception are:

1) The mechanisms embodied in the microprocessor are the same as those required by a parallel processor.
2) The amount of each mechanism required by a multiprocessor is proportional to the number of processors.
3) The inter-mechanism coupling is of a higher bandwidth than the intra-mechanism coupling.

Parallel processing requires additional mechanisms than those found in uniprocessors. Examples include task synchronization, inter-task communication, and atomic compound data manipulation. Different mechanisms have different scaling properties instead of the linear one forced by multiprocessors. Communication requirements may grow as badly as quadratically and synchronization may grow more than linearly as well due to the use of finer granularity. I believe that for a distributed mechanism to be effective, its pieces should be tightly coupled. In a multiprocessor, the pieces of different mechanisms exist in each processor and are in tighter communication than are the different pieces of the same mechanism that are in separate processors.

What results from these observations is an approach to parallel computer design that departs from that of multiprocessors. The functionality of each mechanism is defined from the parallel computing model devised for the system. Each mechanism is designed to be distributed in space, looking something like a layer in a cake. Then the distributed mechanisms are assembled into a single computing ensemble, by piling the layers one on top of another in n-dimensional space. For lack of a more imaginative name, I call this the

LDM architecture for Layered, Distributed Mechanism. The final ensemble is, in a way, a uniprocessor presenting a single interface to the assembler level programmer. However, it interprets parallel programs with each mechanism working on many parts of it at the same time. I see some hints of such an architecture in the Connection Machine, early ideas for data flow, and maybe the MPP.

Let's Stop the Dust from Collecting on OPS5

Salvatore J. Stolfo
Columbia University
New York, N.Y. 10027

18 June 1986

One of the noted impediments to achieving high performance parallel programs is the *dusty deck problem.* Important numeric-based code in use in major computational centers today have achieved ages well beyond the average computer scientist! These dusty decks of source code are not expected to go the way of the dinosaur in favor of the evolutionary offspring utilizing parallel constructs and parallel hardware for their execution. Furthermore, the parallelization of these old sequential codes are not expected to provide orders of magnitude improvements in cost effectiveness that may be possible with reimplementations of parallel software solutions.

Of great concern to me is the issue that the relatively young area of symbolic programming may be forming its own dusty deck problem. Specifically, the OPS5 language, the AI counterpart of FORTRAN in my opinion, may not provide the proper vehicle for high performance symbolic parallel computing. Unfortunately, OPS5 is becoming somewhat of an industry standard in the implementation of AI software which may potentially ferment the noted dusty deck problem.

We have reported a number of parallel algorithms to speed up OPS5 type Production Systems, as well as a number of parallel optimization schemes. (Our present work is aimed towards detailed performance measurements via experimental implementation of our ideas on the operational 1023 processor DADO2 prototype.)

The question to ask at this point is precisely what speed up can be achieved for OPS5 PS programs. Much has been written and debated about this very point. The issue centers upon two key observations.

First, it appears on first glance that the match operation of OPS5 PS's may be executed in parallel by a large number of concurrent PEs. In certain measured OPS5 PS programs, however, it has been reported that on each cycle of execution a relatively small and stable number of rules must be matched. against newly asserted data. That is, on average each rule firing in OPS5 produces new data elements which affect (or are relevant to) the LHS of a *small number of rules.* (Statistics for R1, for example, show about 30 rules out of a set of ~2000 are affected on each cycle of execution.) This small "affect set", as it has been called, thus indicates that the "number" of individual match operations executing in parallel which compute "useful" new matching rule instances is small. Hence, a large number of invoked parallel procedures compute nothing at all and the utilization of the parallel resources is thus poor.

The second observation to note is that the variance in processing times for the concurrent match operations can be quite large. That is, the total running time is proportional to the "slowest" matching rule, indicating that utilization of the parallel resources decreases even further while possibly many PEs wait for the slower PEs to catch up and synchronize for the select phase.

These two observations may lead one to the depressing view that PS programs, in particular OPS5, can best be accelerated by a small number of parallel processors rather than the large-scale approach taken in the DADO machine. We firmly believe this not to be the case at all. The DADO project at Columbia is investigating and has discovered two clever "source to source" transformational techniques as well as a "multitasking" approach that provide the means to reduce the rule matching variance time as well as increase the average affect set size. The essence is to "rewrite" the rule system in such a way that individual rules which require more processing than the "average" rule are replicated a number of times with additional distinct constraints appearing in each copy. The effect of this transformation is to preserve the same computation as the original rule, but to provide the opportunity to match the copied rules in parallel. The total effect is that a number of rules representing the single "hot spot" rule can more quickly calculate the same result in parallel! In our experimentation, we found, for example, that a single hot spot rule copied and constrained to 4 rules produces an overall speed up of the orginial rule's match time by a factor of 5!

The second technique to increase the average affect set size depends upon "concurrent rule firings" where the select phase chooses for parallel execution as large a number of rule instances as possible, rather than a single rule instance. A multitasking approach to this parallel activity on DADO2 has been reported with encouraging simulation results noted -- on the order of a factor of 7.5 times faster execution.

Another technique is quite similar to the notion of "chunking". Whereas chunking has been proposed as an AI learning method, the same technique may be used to coalesce a number of rules into larger rules with larger right hand sides. The net effect is to rewrite a set of individual rules that are often executed serially to a new larger rule (encompassing all the others) whose actions compute the same WM changes and hence create larger affect sets in one cycle of execution. Consequently, the execution of a single "chunked" rule, in place of its forebears, may lead to a larger number of useful match operations calculated in parallel. Unfortunately at the time of this writing, insufficient evidence is available to report the utility of the third approach. It is our opinion, however, from studying a number of small OPS5 programs, that these techniques provide evidence that utilization of large numbers of PEs can be increased and hence performance can be improved overall. Work is presently under way to apply these techniques to a large OPS5 program to be run on DADO2.

Why do we puruse all this work for OPS5? Our strong impression is that OPS5 provides *inherently sequential* programming constructs which necessitates the extraction of *implicit parallelism.* This clearly points out the need for *inherently parallel* constructs in rule based programming providing *explicit parallelism* for high performance.

Much of the debate that has arisen around OPS5 is based upon pessimistic statistics derived from performance of existing "sequential" OPS5 programs. Many parallel processing researchers concur that many opportunities for exploiting parallel processing cannot be derived from study of only sequential programs. After all, sequential programs were written for sequential processors, not parallel processors. This may lead researchers towards techniques of "conventional program optimization" to improve code efficiency on a single PE, the primary approach taken in the implementation of OPS83. We prefer to optimize for increased parallel performance avoiding the potential dead end of dusty decks. Hence, a

more useful approach would be to consider rewriting existing sequential code entirely with the minds eye set on parallel processing. (Fortunately, not many lines of OPS5 code exist in comparison to FORTRAN, for example.) To that end, expression of parallelism is necessary and hence parallel languages are vitally important. Much of the current activity of the DADO Project at Columbia is focused on the specification and implementation of an AI PS language, called HerbAl, designed to be downward compatible with OPS5. The essence of the initial HerbAl definition is to provide a few additional parallel constructs to permit the expression of inherently parallel activities that can be expressed only sequentially in OPS5. The expected net result is to provide improved expression as well as improved performance of AI programs. It is our expectation that the initial HerbAl effort will provide a brighter future for large scale parallel processors achieving high performance execution.

Workshop on Performance Efficient Parallel Programming

The Multi-Satellite Star

Michael Stumm

Distributed Systems Group
Department of Computer Science
Stanford University
Stanford, CA 94305

Introduction

We present a programming paradigm for structuring parallel computations for execution on MIMD parallel systems. We call the structure we use the *multi-satellite star* and believe that it will become the predominant way of structuring parallel computations on a wide range of MIMD parallel architectures, since it is performance efficient, yet simple to implement and easy to use.

The multi-satellite star maps easily onto most MIMD parallel architectures, from shared memory multi-processors to network-interconnected workstation clusters. It is performance efficient because it minimizes communication overhead and distributes the load evenly among the available processors. Moreover, it provides a simple basis for writing fault tolerant parallel programs. From a programmer's point of view, only a few modules must be written in order to generate a parallel program. Most of the implementation details can be hidden in generic run-time libraries.

The Multi-Satellite Star Model and its Implementations

A parallel program written in the pure multi-satellite star model consists of a set of functional application-level instruction sequences we call *subtasks*, alogical processor for these instructions, called *satellite module*, and a master module we call *star central* that acts as a communication, scheduling, and control mechanism for the computation. Star central maintains a priority queue of ready-to-run subtasks which are passed to satellites at their request. Several instances of the satellite module execute concurrently on multiple processors.

Satellites, in an endless loop, request from star central a subtask together with its parameters, execute the subtask without side effects and then return the result values to star central while asking for the next subtask. A satellite executes at most one subtask at a time and it does not maintain state between subtask execution (except possibly by caching portions of the state maintained by star central). Subtask execution is not preempted.

Our experience lies mainly in implementing multi-satellite star programs on workstation clusters. With efficient communication and correct operating system support, a cluster of workstations can be viewed as, and effectively used as, a parallel machine. Using the multi-satellite star model, we have structured algorithms to execute in parallel on a workstation cluster and have been able to achieve speedups comparable to those attained on currently-available multiprocessors. Algorithms we have parallelized include branch-and-bound searching, alpha-beta searching, matrix multiplication, Gaussian elimination, zero-finding, all-pair shortest path problem, dynamic programming, FFT, etc.

Multi-satellite star programs can naturally be realized in such a distributed environment if remote procedure calls and monitors, Ada-style rendezvous, or message passing is supported. Our implementations are all based on the V distributed operating system, where message passing is used and where star central is a server process that services requests from satellite processes.

Our position is that the multi-satellite star structure is suited for shared memory multiprocessors as well, since it also minimizes communication overhead and distributes the load among participating processors well for these systems. On a shared memory multiprocessor star central is implemented as a monitor and procedure calls are used to access its state. We also expect each satellite to have exclusive access to a (local) portion of the memory space so that access to it is contention free and so that it can modify state in its partition without causing overhead for maintaining consistency. In many cases it will be beneficial to replicate common read-only data structures in order to further reduce contention if this is not automatically provided for by large caches.

Even our distributed implementations contain facilities for sharing state in the form of (simulated) shared memory. Shared memory is a convenient abstraction on which to build parallel programs, since processes that wish to communicate need not directly address each other nor need they exist at the same time and because many communication details can be hidden from the programmer. Distributed shared memory is implemented using the broadcast and multicast capabilities of local area networks, and thus allows an efficient sharing of state.

Efficiency

The efficiency of parallel program execution can be significantly degraded by communication overhead, uneven load distribution, synchronization overhead, and superfluous work. The multi-satellite star structure tries to alleviate these problems.

Communication overhead can be minimized by structuring the computation appropriately, by using subtasks with grain sizes adapted to the overhead, and by exploiting broadcast capabilities where applicable. The satellite star structure reduces communication costs in several ways. Satellites are loaded initially with common code and initialized data, which are then re-used by each subtask. Subtask switching therefore reduces to transfer of subtask specification and parameters only. Communication of control is minimal, since all global state is maintained at star central. Synchronization, termination

detection and deadlock detection therefore do not require inter-machine communication. The scheduling mechanism uses only one communication per subtask and ensures that the highest priority subtask will always be executed next. The (possibly simulated) shared memory minimizes the communication necessary for sharing state. Finally, the fault tolerance mechanisms used (described below) do not require extra communication to detect satellite or star central failures during failure-free operation.

We encountered several difficulties in maximizing speedup. Interestingly, the problems occur in shared memory multiprocessor implementations as well, but are exacerbated by higher communication costs in distributed systems.

When parallelizing for-loops that contain data dependencies between iterations, satellites typically execute synchronously, i.e. in lockstep. In this case, variance in granularity can significantly degrade performance, since all satellites must wait for the slowest one to complete. Furthermore, for each interval, star central suffers from front- and back-end communication load when satellites execute synchronously.

Some of the tree searching algorithms, most notably alpha-beta searching, suffer from superfluous computations. That is, satellites can easily be kept busy computing, but they compute more than is necessary, since they cannot benefit from the results of the concurrently executing subtasks.

Fault Tolerance

The multi-satellite star model is fault tolerant in several ways. First whenever a subtask is passed to a satellite, star central retains a copy. This copy is discarded after the subtask has completed. Since subtasks are functional in nature and therefore execute without side effects, a subtask can easily be restarted whenever a satellite fails. It simply needs to be reentered into the queue of ready-to-run subtasks.

Star central must be able to detect satellite failures. It does so by periodically polling a satellite after a timeout period in which it has received no message from that satellite. This timeout period should be a function of the subtask granularity. Note that star central need not recreate a new satellite process after a failure; it can operate with a variable number of satellites. A satellite will always be able to detect a star central failure whenever it tries to communicate with it. In this case it commits suicide to avert any orphanage problems. Star central is responsible for its own recovery. It generally does this by periodically checkpointing its state and by using a backup process that is capable of detecting its failure. Should this occur, a new star central is created that creates its own new satellites and continues operation from the last checkpoint. Hence, satellites need not checkpoint their state.

Programability

The multi-satellite star model is simple as far as programming is concerned. The implementation details of the model can mostly be hidden in run-time libraries. A programmer effectively only needs to write the following mandatory and optional functions.

- mandatory:
  - An initialization function that creates the initial subtask(s) and initializes global data structures given input to the computation.
  - A definition of all subtasks:

    ```
    subtask_i( in-parameters, out-parameters )
    ```

  - For each subtask, an epilogue function that is executed at star central when a subtask has completed. For instance, the epilogue function accepts the subtask's return values and modifies global state accordingly. It also may create new subtasks and enter them into the priority queue.
  - A termination function that outputs the result of the computation, after the termination has been detected.
- optional:
  - For each subtask, a prologue function that is executed at star central before the subtask is passed to a satellite.
  - A scheduling function that choses a subtask from the priority queue for execution on a satellite.
  - A function that checks for a termination condition. (In the default case the computation will terminate when all subtasks have completed executing.)

These functions will mostly be independent of the target architecture. Instances of a multi-satellite star program will differ mainly in the granularity of the subtasks, depending on the target architecture. The optimal grain size will mainly depend on the overhead associated with execution of a subtask. The partitioning and the subtask epilogue functions must therefore be able to generate subtasks of different sizes as a function of the target architecture. Luckily, our experiences indicate that most subtasks operate on regular data structures, such as subtrees or submatrices, whose sizes define the granularity. Being able to generate subtasks with different grain sizes therefore does not, in general, pose problems. Moreover, very rarely will subtask modules depend on the granularity. We therefore conclude that multi-satellite star programs instantiated for different target architectures will differ mainly in the run-time libraries that are used.

Performance Efficient Programming for Share Memory Parallel Processors

Bob Thomas
BBN Advanced Computers Inc.

Position Paper Prepared for
CMU Workshop on Performance Efficient Parallel Programming
September 8-10, 1986

My interest is large scale shared memory parallel processors. In particular, machines which use multistage switching networks to implement the path from processors to memory. The BBN Butterfly Parallel Processor and the IBM RP3 are examples of machines in this class. Because these machines have many processors and many memories, they provide with substantial processor and memory bandwidth to the programmer.

Apart from the normal concerns of writing performance efficient programs for uniprocessors, there are two issues that must be addressed when programming shared memory machines in this class: memory management and processor management.

The goal for performance efficient memory management is to make full or nearly full use of the memory bandwidth provided by the hardware. This means avoiding situations where many processors reference a particular memory at the same time. When this happens program execution is slowed since processors must proceed serially as they access the "hot" memory. Hardware can assist some here: for example, combining memory references within the switch and interleaving memories such that consecutive memory addresses are in different memories can help. While interleaving memories at the hardware level may help, it is interesting to note that it may also prevent programmers who, understanding the dynamics of the reference patterns to their application data, want to allocate the data to memories in a way that ensures processors referencing the data usually access different memories. I believe systems should support both interleaved and non-interleaved memory, and allow the programmer (compiler?) to use the type of memory appropriate for various situations.

The goal of performance efficient processor management is to fully utilize the processor bandwidth. Approaches which are dynamic, or at least semi-static, in the way they assign computational tasks to processors are attractive. They have a natural load leveling effect; as processors become free, they simply take the "next" task. Furthermore, such dynamic and semi-static approaches facilitate the development of programs that are independent of the number of processors. Such programs have performance and reliability advantages; if the current hardware configuration is not powerful enough for a program, the program can be run, unmodified, on a larger more powerful configuation; if a processor or memory is broken, the program can be run on a reduced configuration. Dynamic and semi-static approaches work best in a shared memory environment, where each processor has efficient access to all application data.

To work well, these approaches require a low cost means of allocating computational tasks processors. A fully dynamic scheme would perform task allocation each time a processor becomes free. A semi-static scheme would perform task allocation less frequently for each processor, but often enough to ensure that processors seldom sit idle while other processors have many tasks. If one thinks in terms of task allocation bandwidth (BWa), which is the number of tasks a system can allocate per unit time, and task execution bandwidth (BWe), which is the number of tasks a system can process per unit time, dynamic allocation is appropriate where BWa $\ll$ BWe, and semi-static allocation is appropriate where BWa and BWe are comparable.

We have developed a dynamic processor allocation mechanism for the Butterfly Parallel Processor which has been used to support performance efficient programs for a wide range of application domains.

5 September 1986

The Importance of the Accurate Performance Modeling of Parallel Systems

Position paper by D.Vrsalovic
Computer Science Department CARNEGIE-MELLON UNIVERSITY

1. General

Now when parallel systems are reality we have to do the same procedure we did for all the previous generations of computer systems. We have to learn how to program them after the fact that hardware is already built, and all the efforts to be put in parallel programming, are once more underestimated. Most of the researchers see the parallel systems as a vehicle to achieve three main goals:

- Building the computer systems with much higher performance than those we have today;
- Achieving the inherent fault tolerance via the use of multiplied resources;
- Having the perspective of the lower production costs due to the effects of learning process on higher volumes of identical system components.

While widely accepted as the viable implementation for the last two goals mentioned above, parallel systems are often challenged as the solution for the high performance systems. This comes from the fact that present efforts in parallel programming are performance-wise often ending with catastrophic results (ie. real/potential performance ratio is still very low in most of the cases).

In order to overcome this situation much better performance models of the parallel systems should be built. Having such models we will be able to advance our knowledge, and to build the computer aided tools for parallel programming.

2. Performance Models of the Parallel Systems

Models available today could be classified into two major groups:

- Stochastic models;
- Deterministic models.

While having strong fundamentals in the stochastic mathematic, former models are giving the statistical average behavior of the system. They are usually very robust(ie. insensitive) for the fine grain parallelism. Due to the fact that most of the analytical results are obtained by the use of the Markovian chains, stochastic models are also insensitive to the effects of synchronization.

Mathematical foundation for deterministic modeling is yet to be developed. This is the reason why deterministic models are today only able to cover fine grain effects, while being to complicated for complex system design. It is also obvious that there will be always the data dependent effects in a parallel system, which are inherently stochastic.

It seems that future research should lead towards closing the gap between high level, and low level system modeling. One of the promising approaches lays in combining both kinds of models to cover all the aspects of parallel systems design.

3. Conclusions

Intensive parallel programming can give us the experience necessary to develop better models of parallel systems behavior. Only then when we will have such models, parallel systems design will stop to be the "Art of the few", and become "The Science of us all".

Speedup Versus Efficiency in Parallel Systems

John Zahorjan and Edward D. Lazowska[1]
Department of Computer Science
University of Washington

Exploiting parallelism is an increasingly common approach to improving the performance of computer systems. In terms of hardware, this typically means providing multiple, simultaneously active processors. In terms of software, this typically means structuring a program as a set of largely independent subtasks.

In the sequential world the performance of a system usually can be adequately characterized in terms of the instruction rate of the single processor and the execution time requirement of the software on a processor of unit rate (which we refer to as its *service demand*). In the parallel world things are considerably more complex. In the hardware domain we must be concerned not only with the instruction rate of a processor, but also with factors such as the number of processors. In the software domain we must be concerned not only with service demands, but also with factors such as the structure of the software.

In evaluating a parallel system two performance measures of particular interest are *speedup* and *efficiency*. Speedup is defined for each number of processors n as the ratio of the elapsed time when executing a program on a single processor (the single processor *execution time*) to the execution time when n processors are available. Efficiency is defined as the average utilization of the n allocated processors.

Ignoring I/O processing time, the efficiency of a single processor system is 100%. If efficiency remains at this level as more processors are added we have *linear* speedup. This is the ideal case, as improvements in speedup can be obtained at no cost in efficiency. Linear speedup is not achievable in general, because of contention for shared resources, the time required to communicate between processors and between processes, and the inability to structure the software so that an arbitrary number of processors can be kept usefully busy.

Although the idea of speeding up computations through parallelism has existed for more than a century, general purpose systems based on multiple (five or more) processors have only recently become common (e.g., commercial machines by Sequent, Encore, Alliant, and BBN, and limited-edition machines such as IBM's RP3 and DEC's Firefly). The existence of such systems has stimulated widespread research activity: algorithms work concerned with parallel solutions in many problem domains, compiler work concerned with parallelizing code, architecture work concerned with how best to interconnect processors, etc. Obviously, results in these areas play a critical role in improving the speedup and efficiency properties of parallel systems.

In this work we take a more abstract view. Rather than studying specific implementations and implementation problems, we study the *tradeoff* between speedup and efficiency that is *inherent* to a software system. Further, we do not do this in the context of a specific software structure; instead, we derive relationships that can be very broadly applied. We are interested both in fundamental issues concerning the properties of this tradeoff, and in practical issues that might arise in considering specific software systems. Among the fundamental issues that we address are:

[1]This work performed jointly with Derek L. Eager, Department of Computational Science, University of Saskatchewan.

- To what extent is the speedup-efficiency tradeoff determined by the *average parallelism* of a software system, as contrasted to other, more detailed characterizations?
- How "bad" can speedup and efficiency *simultaneously* become?
- What is the nature of the "knee" of the execution time – efficiency profile, where the benefit (increase in speedup) per unit cost (decrease in efficiency) is maximized? For example, what guarantees can be made regarding speedup and efficiency values at the knee?

Among the practical issues related to specific software systems that we consider are:

- To achieve a given speedup, what efficiency penalty must be paid?
- What is the speedup advantage that will result when increasing the number of processors by some factor?
- What is the efficiency penalty that will result from this change?
- What number of processors yields the knee of the execution time – efficiency profile?

Our objective is to address these questions by obtaining bounds on performance – bounds expressed in terms of the average parallelism measure of software structure. It should be clear that, given complete information regarding a specific software structure, precise answers (rather than bounds) could be obtained for many of these questions. There are two reasons, though, why bounds expressed in terms of one or a small number of parameters may be more desirable than precise solutions that require complete information:

- It is unlikely in practice that complete information will be available. For most software systems, parallelism will depend to some extent on the (unknown or varying) data that would be supplied as inputs. The volume of information required will in many cases be prohibitive.
- It often is the case that bounds yield more insight than exact answers utilizing complete information.

Our pursuit of this approach is in the spirit of Amdahl's law, which states that if a fraction f of a computation is inherently sequential, then the speedup $S(n)$ is bounded above by $\frac{1}{f + \frac{1-f}{n}}$.

(Precisely, f is defined to be the ratio of the service demand of sequential parts of the computation to the service demand of the entire computation.) This is a simple upper bound on speedup that is expressed in terms of a single-parameter characterization of the software (f) and a single-parameter characterization of the hardware (n). It provides considerably more insight than more detailed alternatives, such as a table displaying exact speedup values computed for a number of specific software structures running on a number of specific hardware structures. Amdahl's law illustrates the flavor of the results that we seek.