Workshop on
Performance Efficient Parallel Programming

Sponsored by

Carnegie-Mellon University

and

The National Science Foundation*

Edited by Zary Segall and Larry Snyder

Seven Springs

Champion, Pennsylvania

September 8 - 10, 1986

Steering Committee:

J. Browne, B. Chern, R. Grafton, Z. Segall, L. Snyder

Local arrangement:

P. Reyner

*This work has been supported by the National Science Foundation under grant #DMC-8603052.

Table of Contents

1 Scope and Purpose of the Workshop 1
2 Working Group on Sources of Performance Degradation 3
2.1 The Taxomony 3
2.2 Concrete Examples 4
2.3 Insights 5
3 Working Group on Mapping Algorithms onto a Given Architecture 7
3.1 Introduction 7
3.2 Problem Statement 7
3.3 Rationale 7
3.4 Promising Approaches 7
3.5 Research Directions 8
4 Working Group on Operating System Characteristics for Performance-efficient Parallel Programming 9
4.1 Statement of Problem 9
4.2 Rationale 9
4.3 Approaches 10
4.4 Research Directions 11
5 Working Group on Idealized Parallel Machines 13
5.1 The Problem 13
5.2 Idealized Parallel Machines 15
5.3 Topics for Further Research 15
6 Working Group on Parallel Programming Environments 17
6.1 Problem Overview 17
6.2 Promising/Important Areas of Research 17
6.3 Recommended Approaches 19
7 Majority Report -- Working Group on Language Design Issues in Performance-Efficient Parallel Processing 23
7.1 Introduction 23
7.2 Research Questions and Directions 23
7.2.1 Language Constructs 23
7.2.2 Performance 25
7.2.3 High-Level Issues 25
7.2.4 Miscellaneous 26
7.3 Conclusions 26
8 Minority Report – Working Group on Language Design Issues in Performance-Efficient Parallel Processing
8.1 Prologue 27
8.2 Introduction 27
8.3 Why a parallel programming language now? 27
8.4 Required Characteristics 28
8.4.1 Provides good abstractions. 28
8.4.2 Provides a coherent set of abstractions. 28
8.4.3 Machine independence for a reasonable range of hardware 28
8.4.4 Miscellaneous qualities 29
8.5 Conclusions 29
9 Working Group on Performance 31
9.1 Problem Statement 31
9.2 Rationale 32
9.3 Research Directions & Promising Approaches 34
10 Conclusions 35
11 List of Participants 37

1 Scope and Purpose of the Workshop

On September 8-10, 1986, 44 computer scientists and engineers, mainly active in the area of performance efficient programming of parallel systems, met at Seven Springs, Pennsylvania. The purpose of this meeting was to assess the status of research in the area of parallel programming technology and to determine what future research directions would be most productive. The National Science Foundation and Carnegie-Mellon University sponsored the meeting. This report summarizes the discussions that took place and the conclusions that were reached.

Interest in parallel architecture has grown steadily over the past decade and a half. Parallel processing has long been heralded as a method to build computers executing an extraordinary number of instructions per unit of time. 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, in this workshop we emphasized the parallel programming technology required to apply parallel solutions to problems with the objective of improving execution speed. In preparation for the meeting, the following research issues were singled out:

- 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.

The workshop was a forum to address those issues.

Prior to the meeting, attendees submitted position statements; these were distributed at the meeting to act as a catalyst for discussion and exchange of opinion. A number of informal presentations were made, most based on position papers. In addition, the researchers separated into seven discussion groups: 1) sources of performance degradation, 2) mapping algorithms onto a given architecture, 3) operating system characteristics for performance-efficient parallel programming, 4) idealized parallel machines, 5) parallel programming environments, 6) language-design issues in performance-efficient parallel processing, and 7) performance. The mechanism to determine discussion group topics was as follows: a large number of topics were "nominated" by the participants. A group was formed to discuss that topic if a critical size (> 4) people could be formed. Individuals were members of only one group. No attempt was made to "balance" the group topics to cover the wide spectrum of issues that might be discussed. The choice of group topics thus represents a statement about what this group of researchers felt was most important. Each discussion group created a summary of its deliberations; the summaries follow in this document. These summaries were written at the workshop while the discussion was fresh. The time was very short, however, and so even though the editors have reviewed the material, the authors should be excused if these summaries are less well-polished than their normal technical papers. If a group could not come with one agreed-upon summary, the group was encouraged to produce a majority report and a minority report.

Most groups chose to make recommendations of what research should have the highest priority for that area in the near future. The recommendations are summarized in the conclusion section of this document.

2 Working Group on Sources of Performance Degradation

Ed Lazowska, U. of Wash.
David Nicol, NASA
Dave Rodgers, Sequent
Tom Sterling, Harris
John Zahorjan, U. of Wash.

The subject of this working group is at the center of the workshop. In order to achieve performance-efficient parallel programming, it is necessary first to identify the sources of performance degradation; then, research attention can be devoted to those areas that offer solutions: performance analysis, programming environments, operating systems, etc.

We begin this report by presenting a fairly abstract taxonomy of the sources of degradation. We then present a set of concrete examples to illustrate the effectiveness of the taxonomy. Finally, we present some insights concerning ways to address these sources of degradation.

2.1 The Taxomony

Our performance objective is to maximize

$$\frac{\text{results}}{\text{resources x time}}$$

In attempting to achieve this, many possible sources of performance degradation must be considered. The following tree structure is our taxonomy:

Losses

- Losses manifested by processor busy time:
  - *Redundant computation* Example: late termination.
  - *Overhead* Examples: synchronization overhead; context switching overhead; communication overhead; operating system overhead.
  - *Algorithm* Example: selection of an inappropriate algorithm with long running time.
- Losses manifested by processor idle time
  - *No existing work to be done* Examples: not enough threads to be executed, because of synchronization in the algorithm or a small problem size; threads with mismatched computational demands so that "true" concurrency is minimal.
  - *Existing work is not accessible* Examples: latency, contention, imbalanced allocation of work to processors.

2.2 Concrete Examples

We posed the following question to the Sequent, Harris, and Warp members of our working group: "When a user comes to you and says, 'I paid you $300,000 for this machine and my code doesn't run as fast as I expected,' what are the most common sources of the problem?" Here are the replies, approximately rank-ordered:

- Sequent
  - A problem decomposition that puts most of the work in one thread (e.g., the optimizing phase of a concurrent compiler), so that no real concurrency can be realized. (This is an instance of "mismatched threads" listed under "no existing work" in the taxonomy).
  - Memory thrashing due to a poor choice of operating system parameters, (illustrating "latency" under "work not accessible").
  - Excessive I/O which is not overlapped with computation ("latency").
  - A synchronous software structure, such as might arise from a very large granularity or a producer-consumer relationship with a tiny number of buffers ("not enough threads" under "no existing work").
- Harris
  - Synchronization overhead ("overhead").
  - Contention for shared variables, including counting semaphores, task queues, the "problem heap" ("contention" under "existing work not accessible").
  - Starvation due to a small problem size ("no existing work").
- Warp
  - Excessive I/O that is not overlapped. Note that Warp users are relatively sophisticated compared to Sequent users, so the first two Sequent problems are not encountered by Warp; this many change when they export the machine.
  - Data dependencies in loops ("not enough threads" under "no work").

One could extend this list with examples from vector processors and dataflow machines. The purpose is to illustrate that "concrete" problems described in "user-oriented" terms do indeed map into the taxonomy in the previous section.

Several other points are worth noting. Primarily, various "aspects" or "problems" of parallel programming have manifestations at several leaves of the taxonomy. For example, the act of synchronization can cause overhead, can result in loss of concurrency ("not enough threads"), and can be a source of contention (on

counting semaphores). As another example, the "mapping problem" manifests itself as excessive overhead, as mismatched threads ("no existing work"), and as latency and imbalance ("existing work not available").

2.3 Insights

Several conclusions can be drawn from the preceding remarks and from our discussion.

(1) We believe that reducing performance degradation is a process of iteration among three phases: determining that losses in fact exist, determining the cause of those losses, and making modifications to reduce the losses.

(2) We believe that there are complementary analytic and empirical approaches to each of these three phases. Neither analysis nor experimentation should be the only tool that one uses in addressing the problem of reducing performance degradation.

(3) We believe that models and measurements are fundamental to the process we have alluded to. Models of various sorts (algebraic, simulation, stochastic) are useful in deciding that things could in fact be better (i.e., in determining that losses exist), and in cost-effectively assessing the effect of possible modifications. Measurements are necessary to understand the way parallel programs behave, to parameterize models, and to empirically assess results. Simplicity should be the goal here: there is an extensive history in the area of sequential computation of over-estimating the amount of detail necessary to achieve an understanding of what's going on; it would be nice to avoid repeating that history.

(4) Research in the areas of all the working groups is necessary to address the objective of reducing performance degradation. In performance analysis, we need to determine what factors are appropriate to measure, what types of models are appropriate to what tasks, etc. In programming environments, we need to support the entire cycle described in (1), in addition to other aspects of parallel programming. In the area of virtual machines, we need to determine a "small covering set" of virtual machine abstractions that can efficiently map algorithms onto architectures. And so on, down the list.

(5) We note that interest will not always be restricted to a uniprogramming environment. Again using sequential computing as an analogy, it was originally the objective to run a single job as quickly as possible. Various factors led to multiprogramming, and to a new set of performance problems. These are perhaps not

"programming" problems, but in a sense they are, because they are at least "performance" problems. We should not be short-sighted concerning them.

3 Working Group on Mapping Algorithms onto a Given Architecture

Fran Berman, U. of Calif., San Diego
John Feo, Lawrence Livermore
Jeanne Ferrante, IBM
Charles Holland, ONR
Leah Jamieson, Purdue
Dave Mizell, USC
H.J. Siegel, Purdue

3.1 Introduction

The discussion on the mapping problem was lively and fruitful. In the course of our discussions it became clear that the mapping problem is actually a family of problems which describe different facets of implementing an algorithm on an architecture. What determines each mapping problem instance is the level of abstraction of the algorithm and architecture representation, and the specification of cost and performance measures. The following document attempts to collect our thoughts in this area.

3.2 Problem Statement

The mapping problem is the process of implementing a computational task on a given target architecture in order to maximize some performance metric.

3.3 Rationale

The mapping problem is fundamental to parallel computing. Solving it is essential to making parallel machines cost-effective and achieving their performance potential. It makes programming for a high-level virtual machine possible, and thus makes parallel processing accessible to a wider user community. It makes possible the development of software tools that automatically determine the mapping, as well as techniques to help programmers specify the mapping. A mapping methodology applicable to different architectures would increase the portability of different algorithms.

3.4 Promising Approaches

Current work on the mapping problem can be classified according to the representations which form the bases for the mapping transformations. Graph based approaches perform transformations on some aspects of the algorithm and architecture represented as graphs. Examples include the work of Hennessey and Sarkar, Fishburn and Finkel, and Berman et al. Linear algebra based approaches represent the graph and/or its data

dependencies by a matrix, then transform the graph by performing matrix operations. Included in this category is the work of Moldovan et al. and Ramakrishnan et al. Language based approaches transform one form of program text into another form, where the target form textually incorporates information about the architecture. This is exemplified by the work of Kuck et al. Characteristic based approaches represent the algorithm in terms of a set of characteristics which determines the transformations. An example of this approach is the work of Jamieson and Siegel.

3.5 Research Directions

The list of research directions can begin with the four representation-based approaches identified in the previous section:

- graph based approaches
- linear algebra based approaches
- language based approaches
- characteristic based approaches.

Additional research directions include:

- Experimentation: We need empirical data to support/refute conjectures, and to provide experience and knowledge.
- Evaluation of mappings: What is an optimal mapping? What are appropriate performance/cost measures?
- Automating mappings: What parts of a mapping can be done efficiently automatically (e.g., task decomposition, resource assignment, data allocation)?
- Automatic translations: Can automatic translations make good use of the underlying parallel architecture?
- Retargeting mappings: How do we retarget the implementation generated by a mapping for one machine into an implementation on another machine?
- Meta mappings: Is it feasible to transform a mapping for one machine into a mapping for another machine?
- Models: What are the appropriate models of algorithms and architectures? Are they hierarchical? We speculate that the mapping process occurs at several levels of program abstraction. At each step, mapping decisions are made based on cost constraints defined by a corresponding succession of increasingly detailed models of the given architecture.

4 Working Group on Operating System Characteristics for Performance-efficient Parallel Programming

David Black, CMU
Jordan Brower, U. of Wash.
Raphael Finkel, U. of Wisc.
Herb Schwetman, MCC
Michael Stumm, Stanford

4.1 Statement of Problem

Our task is to suggest ways in which the operating system level of the software can assist in promoting efficient execution of parallel programs. In particular, we will suggest features that the operating system should and should not provide to its clients.

4.2 Rationale

We believe that every parallel programming system will have an operating system. The efficiency of the parallel program is directly affected by this operating system. Furthermore, the facilities provided by the operating system can form a common base of efficiently implemented functions that make it easier to produce correct and fast programs.

An operating system has two major components: the kernel and the utility routines. The kernel is that part which is resident on, or at least executed by, every processor. It contains critical routines to manage resources, particularly the implementation code that carries out policies that may be specified outside the kernel. The kernel should probably be protected (at least from overwriting). Reasons to include a function in the kernel included (1) security or integrity, (2) efficiency, (3) convenience, (4) consistency, and (5) reliability and error recovery. Utility routines can be subprograms linked with ordinary tasks or special-purpose tasks built by systems programmers. We will not discuss these routines here.

Our discussion concerns "general purpose" parallel architectures, not special-purpose, restricted ones. Such a machine is self-contained (not requiring an external host, except perhaps for initial loading), is not dedicated to a single application or language, and has general computational abilities. We do not address such architectures as the Connection Machine, Warp, or FPS array processors. We are also not attempting to propose operating system functions for SIMD machines.

4.3 Approaches

We strongly advocate that the operating system be built as a multiple-user facility. Even if it is used in simple-user mode for production, it will be used in multi-user mode for program development. Furthermore, issues of security apply even in single-user systems. Although the interface between application and kernel should be as fast as possible, security must be maintained.

There are fundamental differences between shared and distributed memory architectures. Nonetheless, the kernel utilities should provide both for message-passing and for shared-memory semantics.

The kernel should support two different computational abstractions: tasks and processes. A task is an address space along with a certain amount of context, such as capabilities (open files, Inter-process Communication (IPC) privileges). It is a basic unit of allocation. Tasks can overlap their address spaces with each other. A process (also called a light-weight process) is a thread of execution, that is, a program counter, a run-time stack pointer, and a set of private registers. Each process belongs to exactly one task. When a task is created, it typically is formed with one process. The task can continue to exist even after all its processes have terminated. Processes within a task potentially share all the address space of the task (limited by programming language rule, if desired) and may executed simultaneously.

Representative operations on tasks include creation, termination, swapping in and out, modification of the address space size, migration from one region of physical store to another (or possibly onto a different machine in a distributed-memory environment), state queries, and protection changes. Representative operations on processes include create, suspend, continue, terminate, and schedule. Group operations that affect all the processes in a task should be available.

Scheduling is performed on a per-process basis. Real-time schedules may be outside the scope of the general-purpose facilities we are concentrating on. Information about scheduling needs to be available to help guide programs. In particular, hints about the amount of available parallelism can be helpful.

We believe that virtual memory is necessary and can be used to assist IPC. The kernel should support memory hierarchies directly (backing store, main store, cache) and provide facilities to allow programs to deal with nonhomogeneous memory (local/global) to gain efficiency.

Inter-process communication is needed. Its semantics could take a wide variety of forms; there is little consensus on this issue. However, it seems that broadcast (group communication) is important, that unreliable media must be supported, and that the typical case should be extremely fast. A remote-procedure call semantics can be both efficient and usable.

4.4 Research Directions

The overall approach to all of these areas must eventually be a system-building approach; operating system behavior and interaction with user programs is difficult to move, and almost always requires a working implementation to validate any modeling work.

We feel that the following areas deserve research attention.

(1) Scheduling. Parallel programs present fundamentally different scheduling requirements for efficient execution than do sequential programs. Locking and synchronization behavior is a major contributing factor, but there are certainly others. Research is need to determine the appropriate scheduling policies for efficient execution of parallel programs. Promising research directions include

- Processor affinity for lightweight processes
- Simultaneous (or gang) scheduling of the cooperating components of a parallel application.
- Policy-mechanism separation to allow greater user input into scheduling policy.

(2) Exception Handling. Virtually all work on exception handling assumes and is fundamentally dependent upon a sequential execution model. Parallel programs present new problems due to the presence of threads of execution other than the thread that causes the exception. Basic groundwork is needed to come up with a reasonable exception handling model for parallel applications.

(3) Increased Information Bandwidth between user and operating system. Parallel programs need more input on kernel policy decisions (scheduling in particular) and correspondingly impose greater demands on kernel services. Also needed are new developments to increase the inner-kernel communication bandwidth, such as a shared page or pages, to reduce the cost of the required cooperation between parallel programs and the operating system, thus contributing to program efficiency. Such mechanisms deserve investigation.

(4) Hardware support. The efficiency of some operating system facilities could be improved by hardware

support. Examples include: user accessible, high resolution clocks, both real-time and cpu-time; support for tracing at higher levels than instructions (e.g., IPC events); translation lookaside buffer connecting to parallel cpu's; assist process management, e.g. more register sets. These topics should be investigated on an overall cost-benefit basis, i.e. what is the cost to the system for the increased efficiency.

(5) Monitoring and Debugging. The fundamental nature of these activities will require support from the operating system kernel (hooks if you like). As research into parallel monitoring and debugging proceeds, the requirements for these kernel hooks will become clear.

(6) Synchronization primitives. Due to the interaction of synchronization with scheduling (a kernel function) an operating system kernel for parallel processing must support synchronization primitives. Both a simple quick lock (such as test and set) and a more complicated lock (such as a kernel semaphore with an associated process queue) are needed for efficient operations, including the avoidance of excess busy-waiting. Implementation and further investigation of alternatives in this area are needed.

(7) Group Operations. Parallel programs are of necessity larger than the process or task that the kernel recognizes as the fundamental units of computation. As a result, operations must be efficiently supported on groups of tasks and processes. Further work is needed to delineate requirements for such operations and define appropriate semantics.

(8) Memory Structures. Recent hardware developments in the parallel processing arena have exposed memory hierarchies to software management and returned to shared memory structures with nonhomogeneous access times (e.g., RP3, Butterfly). This clearly impacts scheduling and memory management; more research is needed to adapt and develop operating systems to efficiently support parallel programs on these new architectures and machines.

5 Working Group on Idealized Parallel Machines

Larry Snyder, U. of Wash.
Larry Rudolph, Hebrew U.
Terrence Pratt, U. of Virginia
Heinz Muhlenbein, GMD
David Culler, MIT
Larry Carter, IBM

A large number of parallel algorithms, parallel machines and parallel languages have been created in recent years, and perhaps the most noticeable feature of this work is its diversity: algorithms are based on very different assumptions making them hard to compare; machines exhibit radically different architectures with few reliable figures of merit to compare them by; and it is often problematical now to generate performant object code for a given language-machine pair even though programs must be portable. This diversity has complicated our use of parallelism.

A similar diversity would also complicate the sequential domain were it not that the von Neumann machine serves as a standard by which to reduce complexity. Here is how: the von Neumann machine, not the literal device he defined, but rather an idealization of it distilled through the years to a few salient features including a program counter, a random access memory, etc., defines for the architects the facilities that a physical machine should have, and it defines the facilities language designers can expect in their target machines; it also implies performance characteristics by which one can determine to a first approximation the performance of an algorithm. The idealization saves the architect from considering the specifics of the languages running on the machine, though he may. The idealization saves the language designer from knowing all the details of the architecture and enables him to have portability by limiting himself to these features. It is thus a consensus describing what architects will build and what language designers will build upon.

5.1 The Problem

To a large extent the problems of diversity in parallel computation could be reduced if there were one or more idealized parallel machines.

> Problem. Identify one or more idealized machines to serve as a basis of communication between language designers and architects and to serve as a basis for evaluating the practical performance of parallel algorithms.

We use the term idealized machine to mean a (small) set of salient features that are significant in determining the performance of a class of algorithms. These are the features architects must implement well and the features language designers can depend upon. Similar concepts have been identified by a number of researchers (Pratt, Rudolph, Segall, Snyder, and Browne): a virtual machine, an implementation machine, a type architecture, and a computational model.

It is assumed that there will be a small number of idealized machines corresponding to a presumed small set of parallel computation approaches; if the number of idealized machines is not small, it probably means that we have not identified the right salient features. Notice that any physical machine will implement the features of each idealized machine with a different degree of efficiency. The "implementation" includes hardware, firmware, operating system and perhaps compiler optimizations. Generally, the idealized machine that a physical machine most efficiently realizes can be thought to be the family to which that physical machine belongs. Likewise, different idealized machines will serve as a platform for each parallel language with differing efficiency. Algorithmic paradigms will generally exhibit different performance on different idealized machines. It is intended that these differences be good predictors of practical efficiency.

The idealized machine is selectively incomplete in order to focus on critical features of the parallel processesing phenomenon which it is describing. The features not mentioned are those whose characteristics are either implied by the critical ones or only influence performance modestly. Notice that this fact that the idealization is not a full specification means that it is not a "layer of abstraction." Nor is it a "model of computation," which has a formal definition.

In order to illustrate the concept, the working group developed a list of idealized parallel machines. Although this list was compiled from more than two dozen candidates, we emphasize that it is only an initial estimate. We have not tried to be complete, so there may be other parallel processing phenomena not included. We have found it most difficult to unify concepts, so there may be redundancy in the list. We have found it difficult to separate salient from irrelevant features, so the idealized machines may be best described by a different set of features. Nevertheless, we present our candidate list.

5.2 Idealized Parallel Machines

- *Local Memory Machine.* Multiphase, topologically-oriented local memory multiprocessors; programs exhibit a series of phases, each with a specific topological structure that persists for the duration of a phase; the number of processes can change with each phase.
- *Shared Memory, Asynchronous.* Global memory multiprocessors with $f(p) \geq log(p)$ memory reference delays; pipelined memory reference is possible.
- *Shared Memory, Synchronous.* Global memory multiprocessor, with $f(p) \geq log(p)$ reference delays; programs execute in lock step.
- *Data Flow.* Dynamic instruction scheduling based on the availability of operands.
- *Computational Memory.* Small computational elements in a massive array of memory structured in some nonlinear way; the machine is capable of associative search.
- *VLSI Machine.* A grid containing a layout of small processing elements and communication paths each having comparable significance.

Notice that idealized machines are not each described by the same set of characteristics. This is because they encapsulate different types of parallel processing, which cause different resources or organizations to be significant. It would not be appropriate to list the properties found in the above list and then to expect each entry in the product space to describe an interesting, or useful, or even sensible idealized machine.

Additionally, note that one does not program in an idealized machine; rather one programs in an abstract language model defined in terms of an idealized machine. The programmer keeps the idealized machine in mind as he programs in order to assess the potential efficiency of his program. Moreover, one does not try to exactly implement an idealized machine with a physical machine and operating system; the idealized machine is a medium of communication and need not literally exist.

5.3 Topics for Further Research

The challenge is to refine the above list so that it may serve as a basis of algorithm development, language development and machine construction. As specific suggestions, the committee recognizes the following:

(1) Verify that the idealized machines listed are independent and different with respect to performance efficiency. It should be possible to find properties that distinguish each pair of entries.

(2) Establish the completeness of the list, or at least support its completeness by identifying parallel computation phenomena and determining whether they are represented by one or more entries on the list.

(3) Assess languages in terms of how well they can be hosted by each idealized machine.

(4) Assess physical architectures (and their operating systems) in terms of how well they host algorithms defined in terms of the facilities of each idealized machine.

(5) Discover general intertranslatability between the idealized machines.

6 Working Group on Parallel Programming Environments

Jan Cuny, U. of Mass., Amherst
Bob Grafton, NSF
George Hetrick, DEC
David Notkin, U. of Wash.
Tom Reinhardt, MIT
Karsten Schwan, Ohio State
Zary Segall, CMU
Bob Thomas, BBN

6.1 Problem Overview

Parallel programming environments, needed to facilitate even routine programming, will be of particular importance in the interactive development of performance efficient parallel programs. In some cases, it will be possible to incorporate aspects of existing sequential environments; in others, new tools and paradigms will have to be developed. The need for these new techniques arises from four factors:

(1) Temporal Complexity - parallel computations appear to be more complex than sequential computations because of the potentially large amount of simultaneous activity.

(2) Plurality of Models - there are many models of parallel computation both at the language and the architectural levels.

(3) Lack of Common Knowledge and Experience - we do not have the backlog of experience with parallel systems that has enabled us to develop appropriate environments for sequential programs.

(4) Irreproducibility - due to asynchroneity, it is often impossible to replicate specific executions of a parallel system.

Whether these factors represent fundamental or merely qualitative differences between sequential and concurrent domains, they significantly affect the viability of our current parallel programming environments and the foreseeable directions of our research.

6.2 Promising/Important Areas of Research

We have enumerated an (undoubtably incomplete) list of areas in which parallel programming environments require technology other than that currently available in sequential environments. Most of these areas have analogies in the sequential domain; however, our current understanding of parallel programming does not yet allow us to solve the parallel version of these problems. Table 1 relates each of these areas to the four factors listed in the introduction; checks represent factors that currently affect the area quantitatively, if not qualitatively.

(1) Mapping/restructuring. The transformation of high-level parallel programs to their eventual implementations is not yet as straightforward as for sequential systems. Similarly, restructuring programs to increase parallelism is still an art. Getting the various mappings involved in these transformations "right" is not yet the responsibility of the environment.

(2) Performance Prediction. Performance efficient parallel programs are the goal of most parallel programming efforts. The many language and architectural models, combined with our relative inexperience in constructing parallel programs makes it essential for us to focus on predicting the performance of a program from its initial conception.

(3) Representation. Different program models and their constituents are best described by different representations. For instance, target I/O parts may be tabularly associated with host I/O streams, while processor connections may be best defined by relations. Because of the plurality and complexity of models, the need for appropriate representations is exacerbated in the parallel domain.

(4) Debugging for Correctness. In the face of massive parallelism and asynchrony, it is often difficult to create parallel programs that solve intended tasks. Bugs can occur in any of the levels of mappings, as well as in the complexities of interprocess interactions.

(5) Debugging for Performance. Again, our goal is to construct efficient parallel programs. Until we can predict (and avoid) "hotspots" and bottlenecks more competently, we must rely on tools to help us improve performance.

(6) Managing information/views. The many data and threads of control lead to a large environmental repository. Condensed and restricted views of this repository are needed to help users benefit from the information and to ease the development of tools that manage only parts of the repository. Views of both static and dynamic information are required.

(7) Semantic Support. Many steps in constructing parallel programs can be done (semi-) automatically. For instance, many classes of interconnection structures can be constructed with only a little guidance by the user. Semantic support helps relieve the user of details not conceptually part of parallel programming.

(8) Specialization. General-purpose parallel programming environments are beyond our current abilities. Instead, environments specialized, for instance, for a particular class of algorithms or program model demand our attention. Software structures that allow a range of specialized environments to be constructed at reduced cost are required and feasible.

(9) Intervention. The ability to intervene in program execution is common in sequential environments (e.g., breakpoints and the setting of values within a debugger). The extent to which intervention is feasible or even desirable is open to much debate (and, in fact, generated more dissension than any other in our meetings).

(10) Expertise. As we become more expert in solving problems related to parallel processing, many (now difficult) tasks can be left to the environment. Today's expertise may be tomorrow's semantic or syntactic support.

(11) Operating System/runtime support. Many environmental tools, for instance program observation tools, require specific support from the parallel architecture. The exact nature of the support is still to be determined.

6.3 Recommended Approaches

The obvious direction for research is to pursue answers to the above questions. In many cases, however, the complete answers will have to wait for progress in related areas. Tools to support mapping and methodologies, for example, will have to await a better understanding of those processes. It is premature to expect complete solutions to the problems of any of the above areas and it is even more premature to expect to produce sophisticated, general purpose, integrated parallel programming environments. Instead, we believe that current research should focus on specific issues and we suggest the following four approaches.

(1) Construction of complete but narrow, parallel programming environments. This is needed (i) as a way to facilitate the use of parallel computers by current users, and (ii) as a way to develop suitable methods, techniques and paradigms for building more general parallel programming environments. Obvious dimensions for narrowing include the class of problems to be solved and the architecture to which the environment is targeted.

(2) Development of flexible test-beds. This would allow for low cost experimentation with proposed techniques and fast prototyping of specific tools.

(3) Creation of appropriate infrastructure for the construction of parallel programming environments. Constructing a parallel programming environment directly on top of a conventional operating system is difficult because the abstractions and tools provided by operating systems do not map easily to the abstractions and tools required by parallel environments. For example, interconnection structures are represented more appropriately as relations than as files and the mapping between these representations is complex. A level of infrastructure, represented by suitable abstractions, tools, and integration methods eases the construction of programming environments. The construction of such an "environmental architecture" would be particularly useful in the parallel domain as we experiment with environments for a variety of application classes and architectural models.

(4) Development of methods for comparison and evaluation of tools and environments. The three previous research approaches must be accompanied by the development of methods that support the comparison and evaluation of tools, mechanisms and complete parallel programming environments. Additionally, evaluation of sequential environments and tools must proceed with the goal of transferring as much technology between the two classes of environments as possible.

	Temporal Complexity	Plurality	Lack of Experience	Irreproducibility
Mapping/Restructuring		√		
Performance Prediction	√	√	√	
Representation		√		
Debugging for Correctness	√			√
Debugging for Performance		√		
Managing Information/Views	√			
Semantic support	√	√		
Specialization		√ + cost		
Intervention*				
Expertise	√		√	
Operating System/ Runtime Support*	√			

* These areas are also influenced by the feasibility of implementing existing sequential techniques in the parallel domain.

Table 1: The impact of parallel/sequential distinctions on programming environments

7 Majority Report -- Working Group on Language Design Issues in Performance-Efficient Parallel Processing

Beth A. Bottos, CMU
Lanny Forgy, CMU
Ted Lehr, CMU
Jerry Leichter, Yale
Vijay Saraswat, CMU

7.1 Introduction

Our working group, unfortunately, could not even agree to disagree. With backgrounds and experiences covering production systems, logic programming, imperative languages, "alternative memory models" (Linda), and low-level machine languages, our range of experience left us with divergent views of not only the current state of the art in parallel programming languages, but also of the future of parallel programming. As a result, our group split into two segments, each with different views of what is important for the immediate future of parallel programming languages. This report represents the majority view, and includes the views of Beth Bottos, Lanny Forgy, Ted Lehr, and Vijay Saraswat. While Jerry Leichter expressed agreement with the views put forward here (with the exception noted in the text), he felt that recommendations as to the specific form of a general parallel processing language were possible and necessary, and, as a result, produced his own minority view report.

7.2 Research Questions and Directions

In our discussions of language constructs for performance-efficient parallel programming, we found that we had many more questions than we had answers. Many of our questions required answers from areas other than programming languages. For example, we did not feel that the parallel virtual machine models with which we were familiar presented the type of virtual machine interface we would like to provide to a parallel language programmer. We also found instances in which our research questions seemed to bring up questions in the domain of parallel operating systems.

7.2.1 Language Constructs

Though we felt that we were not able to present an outline for a parallel language at this time, we were able to identify certain types of constructs that seem to be necessary in the next generation of performance-efficient parallel languages.

We felt that language constructs that are sufficiently low-level so as not to mask unnecessarily the machine architecture are required. (Jerry Leichter dissents here; "because of the form the statement finally took"). For example, synchronization is a common activity in parallel programs. On shared memory machines, a language primitive such as "fetch-and-add" abstracts away from the read-modify-write cycle of the memory, but still allows the user a low enough level of control for efficient manipulation of shared data. While such low-level constructs may not be the way to go in the distant future, since research in parallel virtual machine models may point to more useful higher-level abstractions, in the near future, this low level is needed to give us the performance efficiency we desire.

Overlapping with the desire for low-level constructs is the desire for explicit control of certain types of resources of the parallel machine. For example, in large search problems, we may want to spawn a large number of "tasks" that each search a section of the search space. Only one of these tasks will find the goal, and at that point we want to be able to call off all the other searches. This leads to the problem of resource recall. We also may want the user to be able to control the mappings of virtual resources to physical resources. Since the user-level parallel program will be controlling resources traditionally controlled by operating systems, such as virtual process allocation and termination, the management of resource sharing among multiple users on a parallel machine will be much more difficult than it is on a sequential one. Questions relating to the management of resources shared among several applications need to be addressed in parallel operating systems research.

The subject of light-weight "threads of computation" has come up several times in the course of the workshop presentations. We feel that the existence of truly light-weight "threads" can have a profound impact on the granularity of exploitable parallelism in future parallel programs. This is a current area of work in parallel operating systems. Research should be done in the language area to determine how best to use these "threads".

Given that we can add all these constructs to a language, we will still need to develop compiler techniques to optimize the explicit parallel constructs such as "par-do".

7.2.2 Performance

In order to understand why our parallel programs behave in the ways they do, we must develop a better understanding of connections between algorithm, program and explicit implementation. This is an area not completely understood in the sequential domain: a sequential program implementing a particular sequential algorithm may behave in a manner that analysis of the algorithm did not indicate. Such problems will probably be compounded in the parallel domain.

A related issue to the one of connections between algorithms and programs is the question: "what are appropriate static 'goodness metrics' for these parallel programs?". Concepts such as depth of loop nesting give us handles on the complexity of sequential programs, but, given that we are not sure what types of data and control abstractions a parallel language should be providing, it is less clear how to interpret the frequencies of use of the abstractions that are there.

Relating to the areas of program instrumentation and parallel programming environments, we feel that research is required in the area of measurement tools for dynamic behavior of parallel programs. Many questions remain about instrumentation of programs and presentation of data obtained from such instrumentation.

7.2.3 High-Level Issues

We feel that more work on models of parallel computation is needed. Current work, such as the Linda work, shows us that novel ways of viewing memory may aid in the construction of parallel programs. Are there any other interesting ways of looking at memory? Are there interesting ways of viewing parallel control, other than "par-dos" and "forks"?

The issue of determinism in parallel languages needs to be explored. With a parallel program, unlike a sequential one, issues of inter-task timing become important, and it may be difficult to force a parallel program to exhibit repeatable behavior. What language constructs are needed to ensure such repeatability?

Research is required to determine what role, if any, probabilistic programming might play in real-world programming. Might it help us in thinking about parallel algorithms and programs? Perhaps a "coin-flip" primitive might be useful in future languages, both sequential and parallel.

7.2.4 Miscellaneous

We feel that the development of a body of example parallel programs will help in many of the above listed endeavors. If we had a large body of non-trivial parallel programs, we could (hopefully) identify commonly-used idioms in parallel programming: an example is the parallel prefix idiom of which Larry Rudolph spoke at the workshop. Once such idioms are identified, it should be possible to figure out which ones might appropriately be built into a parallel language. Those that are not primitive to a language might require some sort of linguistic support for the user in maintaining his own library of abstractions.

7.3 Conclusions

We found that we had more questions than we had answers. In the space of the short time we had, we were able to enumerate some of the questions we felt should be answered before a useful language for performance-efficient parallel programming could be designed. Since many of the questions are in areas other than language design, we eagerly look forward to results from some of the current research in parallel machine modeling and parallel operating systems. In the meantime, we find that parallel language implementation consists mainly of adding parallel patches to sequential languages and forcing ourselves to think of parallel programming as piecing-together sequential programs. Work with languages such as PROLOG indicates that there are systems in which parallelism might be implicit, and it might be possible to build performance-efficient parallel systems without a user requiring a language that is a parallel language, per-se. We hope investigations into the above questions will provide some guidance as to which ways of thinking about and exploiting parallelism will prove useful.

8 Minority Report -- Working Group on Language Design Issues in Performance-Efficient Parallel Processing[1]

Jerrold Leichter, Yale

8.1 Prologue

Working Group 5 was chartered with examining parallel programming languages. It is the contention of this Minority Report that there is a sufficient consensus, and a sufficient need, to make designing a parallel programming language with some degree of universality a task whose time has arrived.

8.2 Introduction

In "The American Side of the Development of Algol", Alan Perlis discusses the state of computing in the late 1950's. There is scarcely a word that could not be applied to the state of parallel computing today. The "expos[ure of] opportunities that most of the computing world ha[s] not anticipated and [is] not prepared to exploit", the widespread belief "that only machine assembly language could serve as a useful algorithmic base language" - modified by acceptance of high-level languages for the sequential program portions, coupled with very low-level, completely explicit control over the parallel portions; and, particularly, the profusion of machines and languages "more like each other than anything else": Phrases that could have been taken from a talk on the current state of parallel programming.

Certainly, ALGOL did not achieve the goal of becoming the single, universal programming language. But in the process of producing that "failure", the committee of 13 visionaries created the foundation upon which virtually all programming languages have since been built.

8.3 Why a parallel programming language now?

There are at least five answers to this question: We need a medium for EXPRESSING parallel algorithms; we need a medium for THINKING ABOUT parallel algorithms; we need a medium for PUBLISHING parallel algorithms; we need medium to deal with parallelism in a MACHINE-INDEPENDENT way; and we need a mechanism to produce practical, TRANSPORTABLE parallel programs.

---

[1] This is an abbreviated report. A copy of the full Minority Report is available from the author.

8.4 Required Characteristics

While it is clear what a sequential machine is, a parallel machine is defined by negation: Any computer NOT based on a single instruction stream operating on a single set of data is per force parallel. The class of possible "parallel machines" is thus so large that it is unlikely that a single language, or even a single KIND of language, will be suitable for all of them.

Nevertheless, the required characteristics can be thought of as goals for the designers of a "parallel ALGOL".

8.4.1 Provides good abstractions.

In particular, the abstractions of the language must match what the user wants to think about, they must be mappable to the machine, and they must allow the user to provide low-level information in a natural way.

The third item should be read to say that the previous two cannot be considered separately - it makes no sense to provide one set of abstractions that the user finds natural, and another that maps well to the machine. A simple example: In many numerical algorithms, the user wants to think about vectors anyway - for a vector machine, let him express himself in vectors, not loops. The result is better for both user and compiler.

8.4.2 Provides a coherent set of abstractions.

The language must be amenable to efficient optimized compilation; it must allow the user explicit control of the parallelism; and it must allow the user to develop intuition about the costs of his choices.

The last two items follow from the need for efficient compilation. Even in the sequential case, present compiler technology is limited in its ability to produce efficient code from high-level language programs. In the parallel case, the situation is clearly much worse - the sequential portion of the program must be optimized as before, but now there are a whole new range of variables to consider.

8.4.3 Machine independence for a reasonable range of hardware

It is important to understand what we mean by machine independence. It is NOT our claim that it is practical to produce a language such that a single, unmodified program, written with no concern for or knowledge of the target architecture, can be compiled into efficient code on, say, both an Intel Hypercube and a four-processor Cray. We believe producing such a language and compiler is well beyond the state of the art. Rather, all we require is that it be possible to write, in the same language, a program that will run efficiently on the Hypercube, and also one that will run efficiently on the Cray.

8.4.4 Miscellaneous qualities

The language should avoid a complete break with past experience; it should provide for incremental change where possible; and finally, it should be able to run in a multi-user environment.

8.5 Conclusions

We have made much of the similarities between the current status of parallel programming languages, and the status of sequential languages in the late 1950's. History has shown it was not possible even in 1958 to produce a language that would become accepted as "the" standard programming language. Certainly, no one could hope to produce such a thing for parallel programming today. Nevertheless, there is much to be said for making an effort in exactly this direction, even knowing that it will fall short - both in acceptance and in the basic qualities of the language itself.

We will not go so far as to suggest that the ACM repeat its actions of 1957 and call for the formation of a committee to design a new language. We are interested rather in encouraging the development, within the parallel programming community, of a movement toward agreement on a common parallel programming language.

9 Working Group on Performance

Ingrid Bucher, Los Alamos
Ronald Larsen, U. of Maryland
Joanne Martin, IBM
Dalibor Vrsalovic, CMU

9.1 Problem Statement

There is a large body of literature focusing on performance evaluation of sequential processors. Analytic models and simulation techniques have been developed for those systems, concentrating primarily on issues such as capacity planning, system utilization, and throughput analysis. Unfortunately, the methods developed for sequential machines do not transfer naturally to the analysis of performance on today's complex vector and parallel systems. The performance issues are, in many cases, more concentrated on speed and turnaround time for individual applications than on the system issues that can be handled by the known (usually statistical) methods.

Measures that provide well-bounded estimates of sequential machine performance have proved inadequate to characterize the performance of multiple processor systems. Using these measures, a given architecture can have a performance range of two orders of magnitude depending on the application that is being executed. Moreover, the top of this range is an upper bound to the attainable execution speed of the system, but typically it will be significantly less than its advertised peak execution rate. Such unattainable peak rates lead to unrealistic expectations and should be avoided in any serious discussions of system performance.

Simplistic measures such as MIPS and MFLOPS are not only inaccurate but also misleading when used in the wrong context. MIPS on a vector processor will often decrease as the rate of computation increases. Furthermore, a good metric for the rate of computation is not clearly defined. MFLOPS is commonly used, but is inappropriate for applications that contain relatively few floating point operations and in any case cannot we generalized beyond a specific application's performance.

What is needed is a basis on which to build emerging performance evaluation methods for the plethora of complex architectural systems that is currently available. Basic measurable parameters need to be defined

and measured in both architecture and application domains before usable models can be developed. Using these parameters, classes of applications and architectures need to be defined to order the performance evaluation process. Although lessons can be learned from past performance analysis techniques, new techniques must be developed that recognize the joint contributions of hardware and software systems and thus, should focus on the correlation of the classes of architectures and applications as they become defined.

9.2 Rationale

Performance information is critical at each stage in the life cycle of a computer system, although its intent and utility will vary. During the design phase of a system, primary focus will be on the virtual architecture and secondary focus will be on the workload for which the system is being designed. The development phase will place top priority on the target system -- the software that will serve as a bridge between the application software and the system architecture. Secondary emphasis will be on the virtual architecture and the workload.

Once a system has been developed and moves into an operational phase, the workload will be the main emphasis because it is the most accessible variable. If performance measurements uncover bottlenecks at this stage, the applications software is the exposed and changeable system component. Finally, during an enhancement phase, the three components of the target system, the workload and the virtual architecture would all be considered for evaluation and modification.

In this context, the following table describes the role of performance evaluation in the prediction, detection, and avoidance of bottlenecks throughout the life cycle. The control variables specify where the main emphasis will be at each stage; however, it should be remembered that in all cases the interaction of these components will dictate the total performance of the system.

Performance Evaluation as a Life Cycle Issue

				Measure goal	
Prediction					
	Char.	Set goals	Refine goals	Achievement	Refine goals
	Mod.	Establish feasibility	Isolate potential bottlenecks	Characterize operating range	Establish feasibility
	Exp.	Set baseline	Calibrate model	Calibrate model	Calibrate model
Detection					
	Char.	Limit design space	Recognize bottlenecks	Identify workload/ architecture mismatch	Understand design limitations
	Mod.	Prioritize design options	Isolate observed bottlenecks	Estimate extent of limitations	Evaluate enhancement alternatives
	Exp.	Validate concepts	Verify diagnosis	Verify limitations	Validate enhance-ment
Avoidance					
	Char.	Balance gross design parameters	Refine gross design parameters	Classify poor performers	Correlate bottleneck to design parameter
	Mod.	Find hot spots	Balance fine design parameters	Identify optimal configuration	Rebalance fine design parameters
	Exp.	Calibrate model	Calibrate model to design	Validate reconfiguration	Recalibrate model to design

9.3 Research Directions & Promising Approaches

We see three important areas in which research will be needed in the near future:

(1) Development of new metrics to measure performance of parallel processing systems. In view of the inadequacy of the existing performance metrics to deal with the new degrees of freedom of parallel processing systems, there is a clear need to develop new concepts in this area. It will be necessary to develop interrelated performance metrics for both computer system (including hardware and software), and the applications to be run on that system. It is conceivable that the definition of such metrics in several layers will be useful in characterizing both systems and applications at different levels of detail.

(2) Development of models to capture characteristic factors of parallel computer performance. Like metrics for parallel computing, modeling will have to encompass in a coherent way both the computer system and the applications turning on it.

Enormous problems exist in mapping applications onto a model. Research is needed to show how to correlate real parameters of the workload with parameters of a model. The modeling of a system may be slightly more familiar. However, it might be advisable to build models in layers to characterize the design of the parallel computer system at various levels of detail. The development of new mathematical tools to deal with these models may be desirable in the long run.

(3) Measurement and evaluation tools and techniques. Data collection techniques associated with parallel computations are inherently more complex and difficult than those for sequential processes. They should include hardware and software devices. In addition, software tools will be needed to evaluate and represent the collected data in a useful form (e.g., graphical interfaces, database systems with specialized query languages, etc.). Actual measurements will serve two purposes: to isolate and improve models of parallel computer performance as well as to measure the performance of actual systems. It will be important to minimize, or at least assess, the disturbance caused by the monitoring systems on the system to be monitored.

10 Conclusions

Although the group discussion topics were distinct, a close relationship can be drawn between these topics. In fact one of the working groups seemed to be able to "prove" that each of the nominated topics includes all other topics.

With the exception of the working group on languages, each group was able to come with one, largely agreed report. The working group on parallel language decided to have a majority report and a minority report.

Attendees of this workshop were representatives of a broad cross-section of the research community. Consequently, the conclusions reached by the working groups can be regarded as a strong statement by the research community, in general, and not the opinion of a special interest group in particular.

The overall recommendations can be grouped and summarized as follows:

1) **Develop the technological infrastructure for performance efficient parallel programming.** This includes, but is not limited to, languages, parallel programming environments and parallel operating system primitives supporting performance efficient parallel programming, as well as techniques for performance prediction and evaluation. Specific recommendations are:

- Develop compiler techniques to optimize parallel constructs.
- Explore the issue of determinism in programming languages.
- Identify commonly used parallel program idioms and abstractions.
- Consider the definition of standard constructs for parallel languages.
- Construct complete specialized parallel programming environments to acquire knowledge necessary in building general purpose parallel programming environments.
- Develop practical framework and tools to reduce the cost of the construction of parallel programming environments.
- Identify metrics for comparison and evaluation of programming environments.
- Develop metrics to measure the performance of parallel processing systems including characterization of application workload, idealized machines, and physical architecture.
- Develop formal and heuristic models to capture the characteristic features of parallel computer performance.
- Develop performance measurement and evaluation tools and techniques including: language

instrumentation, operating system instrumentation, data collection and interpretation as well as, the suitable graphical interfaces for the presentation of the interpreted measured data.

2) **Study the sources of parallel programs' performance degradation and the methodology for performance degradation prevention, detection and avoidance.** Specific recommendations include:

- Encourage full blown parallel processing case-studies following the cycle of determination of performance degradation, causes, and methods for their reduction. The observation is that research tools have to be provided by the topics studied by the other working groups. The benefit of this research is fundamental knowledge about the sources of performance degradation.
- Assess the concept of idealized machine in the context of mapping of algorithms into a specified architecture, languages and physical machines.
- Explore ways of reducing performance losses in the operating system though development of performance efficient scheduling, exception handling, memory management, synchronization primitives and monitoring, and debugging aids.
- Study vertical migration of the OS mechanisms and policies into the application program on one side and into the hardware in the other side.

3) **Develop and formalize the concept of the idealized machines as the central practical model for the process of mapping of applications into performance efficient parallel programming.** Specific recommendations include:

- Identify appropriate performance and cost measures to evaluate and contrast mappings of algorithms into a specific architecture.
- Develop techniques and supporting tools for efficient automatic mappings and for mappings transformations.
- Explore the distinctions between static and dynamic mappings.
- Provide sufficiently precise description of the concept of idealized machine.
- Specify in detail the identified instances of idealized machines.
- Evaluate the list of instances of idealized machines for completeness, redundancy and intertranslatability.

11 List of Participants

CMU/NSF 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
sequent!dave@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