Workshop on Multiprocessors for High Performance Parallel Computation

Sponsored by
Carnegie-Mellon University
and
The National Science Foundation*

Edited by
Anita K. Jones, Zary Segall, Chuck Seitz and Andrew Wilson

Seven Springs
Champion, Pennsylvania
June 27-29, 1983

Steering Committee:
B. Chern, A. Jones, Z. Segall, C. Seitz

Local Arrangement Committee:
B. Tomchik, A. Wilson

*this work was supported by the National Science Foundation, Computer Engineering Program under grant number ECS-8215-465.

Scope and Purpose of the Workshop

For three days in late June, 1983, 44 computer scientists and engineers, mainly active researchers in highly parallel computer systems, met at Seven Springs, Pennsylvania. The purpose of our meeting was to assess the status of research in multiprocessor systems that provide high performance, parallel computation, and to determine what future research activities would be most productive. The National Science Foundation and Carnegie-Mellon University sponsored the meeting of this group. This report summarizes the discussion that took place and the conclusions that were reached.

Interest in multiprocessor architectures had grown steadily over the past decade and a half prior to the meeting. Multiprocessor systems appeared to offer many benefits, particularly better absolute performance and better cost/performance than most competing architectures. VLSI technology offered the potential for building substantially larger computers in the form of multiprocessors to solve problems that present computers, as well as predicted future sequential machines, were too small to solve in a timely way. Many architecture designs for multiprocessors exist -- on paper. Only a few machines with 20 or more processors and a high degree of parallelism had actually been built at the time of the workshop. But initial experimental results from these parallel computers were available.

The increasing need for effective parallel computing engines, coupled with recent availability of actual experimental observation meant that it was important at this time, mid 1983, to focus the research in this field. It was appropriate for the designers, experimenters and potential users of parallel computer to take stock and examine the following issues:

- What are the most important experiments to perform in the existing multiprocessor laboratories, and how can the experimenters best communicate the results so others can use them?
- How can one evaluate newly proposed software/hardware architectures and the corresponding parallel algorithms in light of those findings?
- What are the trade-offs in specializing parallel machines to important classes of large-scale applications such as numerical computation, symbolic computation and database.

This workshop was a forum to address these 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 five discussion groups. The mechanism to determine discussion group topics was as follows: a large number of

topics were "nominated". A group was formed to discuss that topic if a critical size of above five 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 important. Each discussion group created a summary of their deliberations; the summaries follow in this document.

Most groups chose to make recommendations of what research should have the highest priority in the near future. The recommendations are summarized in the Conclusion of this document.

1. The Operating Systems Working Group

Ivor Durham, CMU (Group leader and author)
Faye Briggs, Rice
Ingrid Bucher, Los Alamos National Laboratory
Robert Jump, Rice
David Shaw, Columbia,
Pradeep Sindhu, CMU
Alan Smith, University of California at Berkeley

The original charter of this group was to look at four areas: Parallel I/O architecture, interconnection structures, building systems from many components, and operating systems. After some meandering discussion and difficulty in defining what was to be discussed we focused on identifying operating system issues and briefly explored the question of what could be done to facilitate the effective exploration of new systems.

1.1 Issues

The following sections describe briefly the areas of concern in developing new operating systems for high performance multiprocessors.

1.1.1 Operating System Goals

Our preliminary discussion revealed that there are many views of what software in a system constitutes an *operating system* in contrast to other system software. It is essential that the specific goals for the operating system be stated clearly and explicitly before any attempt is made at a detailed design.

The principal goal for any operating system is to make the system programmable. That is the operating system must provide a programming interface (environment). The programming environment must be as straightforward as possible; complexity in such an environment discourages programmers from using the system to the greatest effect. Even perceived complexity caused by new and different notions, such as objects and capabilities, can have a serious impact on programmers who are very familiar with more traditional environments. In multiprocessors, there is inherently more complexity than with a single processor system because the programmer is now required to attend to communication and synchronization among multiple processes.

The second major function of an operating system is to manage the resources of the system. This is particularly important in any system that supports concurrent processes which may vie for use of various resources: memory, I/O devices, etc.

For a high-performance system the issue arises of whether the system should support more than one task at a time. Experience suggests that supporting multiple users need not be excessively expensive, particularly in a system that must already support multiple processes and shared resources. The advantage to supporting multiple users is that multiple applications may be developed simultaneously, but the system may be dedicated to a particular application for a period of time without excessive operational overhead.

Separating operating system mechanisms from the policies they are used to implement has proved effective in several systems. The implementation of policies (such as scheduling) may be handled by software at the application rather than operating system level. This provides considerable flexibility, particularly with respect to tuning and experimentation. Hence, a goal for an operating system is to provide mechanisms that support the policies that may be needed for various classes of application to be run on the system.

An operating system needs to provide instrumentation mechanisms--in combination with the hardware and firmware--to aid in monitoring, debugging, and tuning applications. Specific goals must be chosen for the flexibility and cost of such mechanisms.

A multiprocessor has the potential for being very reliable because of the redundancy provided by the hardware. However, achieving this potential is the subject of considerable current effort. In applications that require massive amounts of computation, the cost of a crash can be quite high given the large amounts of computation that may be lost. Reliability must be approached along several dimensions in an operating system. The operating system needs to be able to adapt easily to changes in the hardware configuration. The designers must choose whether to provide this adaptability dynamically or semi-statically--only when the system is loaded, for example. A dynamic approach is likely to be more complex. The operating system ought to be *fault-tolerant* according to some measure. For example, it ought to be able to survive transient hardware errors and mask as many of them as possible to prolong the processing time available to the application programs. On the other hand, the *availability* of the system may be of more importance. That is, it may be more important to be able to crash the system and restore full service in a few minutes than to provide longer periods of service with degraded functionality or performance. The goals for the operating system must include a reliability policy to guide the designers' trade-offs with respect to achieving a balance between fault-tolerance and availability.

1.1.2 Complexity and Number of Processors

The complexity (power) and the number of components in a system influence the design of a system along a number dimensions. The modularity and structure of the system are influenced by the need, or lack thereof, to have the entire functionality of the system available to all of the processors in the system. Trade-offs must be made between the space required to replicate services and the cost of communicating requests between client and server processors for particular services.

The (in)homogeneity of the system components will influence the design of the programmer interface. When components are inhomogeneous, the task of presenting a uniform interface is potentially more complicated. Resource management is complicated by the variety of costs associated with different units of a particular resource. For example, in a system such as Cm* there is a hierarchy of costs associated with accessing different memories in the system depending on the relative location of the source and destination of the particular reference. These costs must be acknowledged in management policies and in the tuning mechanisms that help the programmer to optimize the costs for a particular application.

The functionality of a multiprocessor operating system does not differ significantly from that of a system that supports multiple concurrent processes. The additional functionality to support communication and synchronization is independent of the number of processors in the system. The variety of mechanisms will depend on the power of the individual system components and the frequency of their use. For example, communicating through shared memory using busy-wait locks may be preferred over a relatively expensive message mechanism.

As the number of components grows, so does the number of the components that are likely to be inoperative at any moment in time. To make effective use of the system, the operating system must be sufficiently flexible to accommodate and mask the loss of particular components whenever feasible. That is, the system should present to the programmer or user a logical machine configuration in which identical components may be substituted by the operating system without affecting the application programs. For example, the operating system may have to adjust the memory mapping to avoid using a particular memory unit that has performed unreliably. Of course, this may be the domain only of the memory management sub-system not the core of the operating system. If changes in the availability of particular components can be accommodated dynamically, the overall period in which the system can perform its duties is extended. In particular the designers must trade between two reliability extremes: *fault-tolerance* to provide some level of perhaps degraded service after some unit has failed and *availability* which is the proportion of time that the system is able to perform its duties properly. In some cases it may be more appropriate to have the system crash and be reloaded rapidly than to have its operation slowed down by (say) 20% because of the loss of some memory.

1.1.3 Common Operating System Abstractions

Is there a common set of abstractions that constitute a minimal operating system and is this set dependent on a specific architecture? If not, could various implementations of the set of abstractions then be shared across various multiprocessor systems, thereby distributing the development effort? The abstractions need to be characterized in terms of functionality, performance, cost, and reliability. One possible set of abstractions is: Memory, Processors, Static Software Units (e.g. Modules, Task Forces), Dynamic Software Units (e.g. processes), Synchronization (e.g. Monitors), Communication (e.g. Mailboxes, I/O), and Monitoring/Debugging probes.

1.1.4 Abstraction versus Transparency

How much of the raw hardware should be visible to the programmer or user? An alternative view of the same problem is how much does it cost to hide the machine details from the programmer and how much benefit is to be obtained by doing so. Experience has shown that the desirable level of abstraction or transparency may depend on the particular stage of development of an application. In the early stages in which the functionality of the application is being developed, an abstract machine is sufficient. In the later stages when performance is more important to the programmer, a much more transparent view of the machine is needed so that unnecessary overheads may be avoided and performance optimized. The principal argument here is whether the programmer wants or needs to see details of the machine that are not relevant to the immediate task at hand.

1.1.5 Potentially Expensive versus Desirable Features

What features of an operating system are desirable, but may impact the overall system performance to the extent that customers needing every last ounce of performance would not want them? Several candidate areas here are *fault-tolerance*, *programmability*, and *dynamic task structure*.

Fault-tolerance requires some of the system resources be used for redundancy rather than for the application. Some components of a fault-tolerant design need not be expensive. For example, the flexibility to accommodate to a variety of hardware configurations does not incur a dynamic cost if the configuration is constructed when the system is loaded. Dynamic changes in the configuration are more difficult to accommodate, but the costs may be incurred only when such a change occurs. Other measures such as checkpointing or redundant secondary storage access incur dynamic costs that must be borne by the application.

Programmability may incur an overhead that may not be acceptable. One of the principal tools that supports modern programming methodology is a *type* mechanism to enforce the integrity of different

classes of data. Objects of a specific class (type) may be manipulated only by the functions defined for that class. In addition to enforcing object integrity the typing mechanism can also support other operations such as garbage collection (where objects need to be tagged to indicate whether they can be accessed or not). The enforcement of type constraints must be efficient to avoid an unacceptable dynamic overhead.

Supporting dynamic task structures may require mechanisms to resolve constraints on the components of the task. For example the Medusa operating system provides a notion of *co-scheduling* which attempts to schedule all of the active processes in a particular task at the same time. If the number of processes changes dynamically, the resource constraints required to satisfy policies such as co-scheduling must be re-evaluated, thus incurring some dynamic overhead.

Question: Can some guidelines be established for trading raw performance for desirable system attributes. What arguments would satisfy customers who need to squeeze every last application cycle out of their machine?

1.1.6 Operating System Structure

How is the operating system structure influenced by the hardware architecture? In systems that support the Client and Server process models, much of the operating system work can be handled in the same was as ordinary application programs, leaving the basic operating system to provide the mechanisms to support Client/Server interactions. The principal issue for the designer here is how to make the various services available to all processes in the system that may need those services. How many server processes are required for a particular service? Such performance questions can best be answered by monitoring the system behavior and tuning it. Provided that the services are designed with the potential for replication in mind, no extra work is required to handle changes in load and performance requirements. It is the job of the operating system to make the variations in how a particular service is provided transparent to the application programmer. The days of the monolithic operating system should soon be past, at least for multiprocessors.

1.1.7 Name Space (Addressing) Architecture

What is the structure of the space of names available to a programmer for identifying data. Should that structure be a linear array of virtual addresses, a graph structure of objects, or a combination of both? Should the programmer see the world from within individual processes or the task force as a whole. The chosen view determines whether there is limited sharing (objects owned only the by task are visible to a process), controlled sharing (any object for which the process can obtain an

unforgeable name may be accessed), unlimited sharing (all processes can access potentially all memory locations), or no direct sharing (information is exchanged more formally through mailboxes).

1.2 Design Principles

Several guiding principles were identified for the design of multiprocessor operating systems. While the list is not necessarily comprehensive, the principles deserve serious consideration:

- The system must be designed as the integration of hardware, firmware, and software components. Therefore the three classes of component must be developed simultaneously. In particular it is highly undesirable that the hardware be designed without due consideration for how it is to be programmed.

- Policies and mechanisms should be separated. The operating system should provide the mechanism necessary to implement the range of policies envisioned for system or application-level software. (This approach also supports a world composed of client and server processes, in which the servers implement the management policies for the resources they make available to their clients. This approach is particularly convenient in multiple-process systems.)

- Functionality should be separable from performance. That is a programmer should be able to develop software initially without full consideration of its ultimate performance, using only a subset of the system resources during development (2 rather than 64 processors, for example). Tuning tools must be provided outside the application to help realize the full performance potential of the application. These tools must help with the fundamental problems of *placement* of code and data and *assignment* of processes to processors. This principle simplifies software development and encourages flexibility in the design of applications (allowing for variability in the available resources, for example).

- The system functions must be logically uniform. The following are examples of logical uniformity:

    - Asynchronous-Function Invocation: A client should not need to know how and where a server is implemented to make use of a particular service.

    - Synchronous-Function Invocation: An application should not be able to distinguish functions implemented in software from functions implemented in hardware, firmware, or software except by their performance.

    - Data Communication Functions: Basic I/O primitives should be device independent, also allowing the device to be another program. For example pipes and I/O streams may be substituted for each other in Unix[tm] allowing programs to be connected together to form more complex programs. (This is ideal for a multiprocessor environment.)

1.3 Potential Experiments for Existing Test-Beds

The principal area in which experiments may be performed on existing test-bed systems is the design of effective interfaces between the system and either programmers or end users. Current work described to the workshop is already beginning this task. In particular the monitoring, debugging, and tuning aspects of software development need to be integrated into a convenient and uniform interface. Interfaces for multiprocessor systems need to be more sophisticated than for uniprocessor systems in that a programmer or user is interested in a collection of concurrent activities. This is an area in which graphics may play an increasingly important role.

1.4 For the Future

Probably the most important problem facing the designers of a new multiprocessor architecture and system is how to achieve the ideal integration across hardware, firmware, and software. All too often the design of the software is delayed until most of the hardware design is complete or even the implementation is completed before software can be tested. In discussing this problem, the group identified some potential benefits that might be obtained from the establishment of an *Experimental-Multiprocessor Laboratory*. The notion is similar to that of the broad test-beds proposed elsewhere during the workshop. Particular goals that are important to the successful use of such a laboratory include:

- The laboratory should provide an environment in which researchers can develop complete new multiprocessor systems from architectural concepts to new application algorithms, but without necessarily having to construct a physical feasibility demonstration, at least initially.

- The laboratory should provide a small, "canonical" multiprocessor system with software for constructing experiments in systems and evaluating them. This includes the ability to build emulators for new architectures that are sufficiently powerful to support the development of software for the new system. The provision of a particular multiprocessor sand-box would have several advantages:

    - A common, shared set of software tools that support a variety of research efforts. Monitoring, data collection, and debugging tools are of prime importance.

    - A shared support effort to reduce the support overhead required when building one's own demonstration machine.

    - A complete system that may be replicated at some future time depending on the demand for a facility closer to particular groups of researchers.

    - Emulator components may be shared across research efforts to enable the rapid construction of future emulators. For example emulators for different interconnection structures could be developed and used as off-the-shelf emulator components.

  - Using one multiprocessor system (however imperfect) to develop ones own ideas can help one to appreciate the difficulties and issues in actually using a multiprocessor.

- The laboratory system should be easily accessible to research communities interested in multiprocessor research. In particular, a convenient network is very important. If the demand is sufficiently high the physical resources of the laboratory could be replicated to form "regional" laboratories.

- The initial laboratory should be used not only to develop a common working environment for multiprocessor researchers, but also to act as a depository for accumulated wisdom, experience, experimental results and so forth. The staff of the laboratory would then be equipped to provide a consulting service to application developers.

The principal problem with such laboratory proposals seems to be human rather than technological: Researches prefer to build their own systems. Much greater control can be exercised over a system in one's own environment and one is not at the mercy of "ill-considered" decisions on the part of the maintainers of a distant facility. The goals described above may help to alleviate this problem, but the problem must be considered when any new multiprocessor research involving the construction of new machines is proposed.

2. The Performance Working Group

George Almasi, IBM
Gerald Estrin, UCLA (Group leader and author)
Glenn Ricart, U of Maryland
Zary Segall, CMU
Richard Snodgrass, U of N. Carolina
Dalibor Vrsalovic, CMU
Andrew Wilson, CMU

2.1 Introduction

The need for a hundredfold increase in performance is the driving force behind the current accelerated expenditure of material and intellectual resources to realize "next" generation computers. Some of the innovative proposed multiprocessor architectures are certainly laudatory; however a lack of clarity concerning performance objectives and performance predictions is evident. In particular, the impact of that lack on the design and development cycle can be disastrous. The Performance Working Group saw the need to introduce systematic methods and automation in order to face the central issue:

> How can we shorten the design and development cycle for new multiprocessor system architectures . . . assuming that performance goals are established during a requirements analysis phase with possible modification during design and development?

Given the above goal, three fundamental aspects were identified:

- Prediction of behavior during system design.
- Evaluation of behavior during system development.
- Measurement of behavior during system operation.

2.2 Issues in Multiprocessor Performance Evaluation

The Working Group then raised a series of questions relating to each aspect of performance evaluation. These issues formed the framework for subsequent recommendations made by the group.

2.2.1 Prediction of Behavior During Design

How can we adequately model multiprocessor systems and parallel computation so that we can predict performance of any proposed system . . . and either reject the proposed system or continue deeper into the costly design and development process?

How can we characterize parallel processor workloads?

2.2.2 Evaluation During Development

How can we create a development testbed for parallel processor systems?

What meaningful set of tools will provide effective automated support for evaluation of parallel processor systems and subsystems during realization of a proposed architecture?

2.2.3 Measurement During Operation

The final test of prediction during design, and evaluation during development, is observation of behavior of the operating realized system.

Is it feasible to formulate natural and synthetic benchmarks which are truly representative of computational loads to be handled by multiprocessor systems?

Is it possible, at reasonable cost, to observe the behavior of highly parallel systems?

2.3 Recommendations of the Performance Working Group

2.3.1 Prediction During Design

We recommend support of research seeking improved models of multiprocessor systems and improved models of parallel computation so that it becomes more reasonable to predict performance and also to deal with validation and verification issues.

We recommend support of research seeking improved methods to characterize workloads for multiprocessor systems.

We recommend support of experiments to determine how far we are from being able to predict performance of complex systems executing highly parallel computation.

2.3.2 Evaluation During Development

We recommend design and development of multiprocessor system testbeds to be used during development by universities, research institutions and industry.

We recommend development of a methodology for design of experiments to be used in such evaluation environments.

We recommend formation of a task force whose charge is to determine how to satisfy the national need for multiprocessor system testbeds.

We recommend support of research into methods for evaluation of testbeds.

2.3.3 Measurement During Operation

We recommend support of research seeking methods for specifying benchmarks in a system-independent manner.

We recommend support of research into observability of multiprocessor systems and into analysis of their behavior.

We recommend development of a knowledge base about benchmark programs and an associated query system. We recommend formation of an interest group to conduct network dialogs and develop critical annotation of the benchmarks.

3. The Models of Parallel Computation Working Group

Jim Browne, UT Austin
George Hetrick, DEC
Mal Kalos, NYU
Simon Kasif, U. Maryland
Joe Mohan, CMU
Dan Ostapko, IBM
Larry Rudolph, CMU
Leah Siegel, Purdue (Group leader and author)

3.1 Fundamental Issues

We view the problem of mapping a problem to an architecture as consisting of (at least) two levels: a high level model based on what might be called a "virtual machine" and a lower level model which deals with specific details of an architecture. Our initial concern is with the high level model. Specific issues include:

1. What constitutes an appropriate high level model of parallel computation? Is there a single model or are there several viable models? Among possibilities are the paracomputer (shared memory) model, message passing models, and data driven models. It is not immediately clear if any one model subsumes the others.

2. Language provides one possible means of expressing a high level model. One way of focusing on parallel computation is to assume the capabilities of a conventional serial language and examine the constructs needed to support parallelism. Candidates include provisions for:

   - Making data appear in the address space of multiple processors. In the most general sense, this may include both shared data structures and message passing.
   - Process control, through synchronization or other mechanisms.
   - Invocation/termination of processes.
   - Creation/destruction of processes. This is distinct from invocation/termination, and implies a notion of ownership of the process to be created or destroyed.

   We are considering the modeling of processes whose execution can be overlapped in time. However this does not preclude execution on a single processor.

3. It is not clear that the above four capabilities can adequately specify SIMD processing. For example, there is no way to describe the selective activation/deactivation of individual processors within a single process.

A number of attributes which will bear on the efficiency of an implementation are not

captured by the general model. Examples are the differences arising from message passing vs. shared memory communications and use of local (by some measure) vs. long-distance communications. This suggests that there may appropriately be an intermediate level model which does not deal with all of the detail of the specific low level model, but which does capture those gross properties of an architecture which pertain to efficiency. At what level of abstraction can the model still provide useful information?

4. Is there a distinction between models and languages? If so, what is the nature of this distinction?

3.2 What research needs to be done?

1. The essential research involves establishing what the appropriate model(s) are. Towards that end, specific models should be defined and compared with respect to inclusion (can one model be expressed by another?), expressiveness, and naturalness.

2. What models do capture properties related to efficiency?

3. Many questions pertain to deciding what is properly modeled at what level. Examples:

   - Consider a tree communication structure. Is that a feature of the highest level or of a lower level model?
   - At what level is the SIMD/MIMD distinction most appropriately modeled?
   - At what level should notions such as data encapsulation and ownership be modeled?

4. A variety of significant applications and algorithms should be expressed in terms of languages and/or models of computation. This will allow evaluation of the utility of the language with respect to expressing parallel formulations of significant problems.

5. Use the analyses of 4 to define the parallel abstract architectures which are suitable targets for translation (compilation) of languages. Study how these machines may be implemented on hardware realizable architectures.

6. Evaluate the relative effectiveness of general purpose and problem specific models of parallel computations.

3.3 Experiments on existing systems

In light of the issues addressed in 3.1 and the questions answered in 3.2, write algorithms in the high level languages for existing and proposed machines. Evaluate the correspondence between the models and the actual programs.

3.4 Summary

The issue of models is clearly an important one, and equally clearly one in which there exists a broad spectrum of views. We are each drawn to the model which most closely approximates our own work. Substantial research is needed to consider the full range of possible models against a wide variety of applications.

4. The Large-Scale Computations' Characteristics Group

B. Chern, NSF
A. Despain, University of California at Berkeley
L. Forgy, CMU
R.M. Lea, University of California at Berkeley
G.J. Lipovski, University of Texas,
A. McAulay, Texas Instruments
B. Rau, Elexsi
R. Warren, Digital
H.J. Siegel, Purdue
H.S. Stone, University of Massachusetts, (Group leader and author)

The committee attempted to characterize the nature of large-scale computations in order to discover what kinds of computer architectures are suited to these computations. Our findings, in general, are that there are significant differences among the various kinds of large-scale computations. It may be quite reasonable to develop specialized architectures that are well-suited to particular classes of problems. It is also rather unlikely that a single general-purpose architecture will suffice for all large-scale problems. To simplify the characterization of computations, the committee focused on three major areas, which in turn are subdivided into specialized areas. The areas reported here are the following:

1. Numeric
    a. Mesh problems
    b. Nonregular (particle-in-cell and similar problems)

Symbolic

    a. Expert System, Knowledge-base System
    b. Combinatorial Search (theorem prover, etc.)
    c. Interactive AI
        i. Vision
        ii. Speech Recognition and Speech Understanding
        iii. Robotics

2. Data Base

The motivation for partitioning programs as given above is that the numeric programs appear to have qualitatively different sets of requirements than do the symbolic programs. Perhaps the

characteristics can be exploited in parallel machines to increase computation speed, and if so, numeric and symbolic problems may lead to different parallel architectures. But in recent years, as large-scale programs have grown more sophisticated, many such programs have taken on characteristics that lie in two or more of the partitions identified above. For example, most expert systems contain a data-base subsystem, and thus are not purely symbolic nor purely data base in character.

Given the possible mixture of different types of computations in a large-scale program, it may well be desirable to use specialized processors for each distinct computation type. For example, a machine for expert systems might well have data-base processors for the data-base component and logic or inference-oriented architectures for symbolic computations. Therefore, we have separated the world of large-scale problems into numeric, symbolic, and data base, and we acknowledge that specific programs many contain any two or all three types of programs.

4.1 Numeric

Table 1 shows the characteristic of numeric programs as identified by the committee. Repetition in these programs is normally associated with nested loops. The table suggests that the computations have a low complexity, which means that computation time grows as a small polynomial in the size of the input data. This suggests that the problems are large because there are many input data, and therefore, these problems require a high-bandwidth I/O architecture. If the input data size is very large, then it is likely that small portions of the input data will be bought into local memories for computational purposes as they are needed, and therefore, there will be additional I/O operations required for data movement between local and auxiliary memory, over and above the extensive I/O required for initial input and final output. This additional I/O exacerbates the I/O bottleneck. We note that some large-scale numerical codes may have less need for high-speed I/O than others, but except for processors that are very specialized to those particular problems, high-speed numerical processors will be structured to support a very high I/O bandwidth.

Table 1 shows that the numerical codes appear to be relatively easy to analyze. The computational bottleneck is usually readily identifiable, and in the case of mesh calculations, the bottleneck is an inner loop that operates on data in a predictable fashion. These characteristics have led to the early implementation of pipeline and array processors, because these architectures were believed to be capable of exploiting the behavior of large-scale mesh-oriented calculations. But program analysis and studies of actual implementations suggest that large-scale numerical computations have a sufficiently large percentage of data dependencies to reduce the effectiveness of SIMD architectures

to the point where they become unattractive. New approaches might generalize the SIMD architecture to enhance its capability to support data-dependent numerical computations including the nonmesh calculations. Or the approaches might abandon the SIMD approach to look toward MIMD architectures for dealing with large-scale numerical algorithms. Table 1 indicates that the latter approach may well be feasible because the numerical program appears to be partitionable, and amenable to implementation on MIMD architectures.

The nonmesh calculations described in Table 1 actually refer to computations that may be derived from mesh representations, but for one reason or another the mesh representation does not lead to lock-step parallelism or to highly predicatable, highly repetitive computations. One important representative of this class of problems is the particle-in-cell (PIC) Algorithm that appears to be ill-suited to SIMD machines because of the data dependencies within the inner loop. Table 1 shows the data dependency to be the primary characteristic that distinguishes this class from the mesh problems. Also included in the nonmesh class is the class of sparse matrix operations, mainly because the locations of the nonzero elements are not readily predictable in many cases. The primary way to gain efficiency is to be able to focus on the nonzero elements as much as possible, and this has proven to be difficult to do for nearly all architectures studied to date.

4.2 Symbolic Computations

Table 2 summarizes the characteristics of two typical kinds of symbolic operations, the expert system and the combinatorial searcher such as a theorem prover. The committee believes that the programs for the two classes of systems may actually have many characteristics in common. The differences lie in the execution of the programs. The combinatorial searchers search a decision tree; sometimes exhaustive case-by-case examination is required. The algorithms that make up this class generally cannot rely on tricks or shortcuts to reduce total computational complexity to very slowly growing functions of problem size, although heuristic approaches have apparently been helpful. On the contrary, expert systems apparently do successfully reduce the potentially large amount of computation to something more manageable because they rely on expert knowledge to follow the more promising paths in the decision tree. In comparing the two types of programs, note that Table 2 shows the complexity of the theorem provers and other combinatorial algorithms to be greater than that of the expert system. There are no published data on which the committee can rely for this opinion, but this result reflects the collective intuition of the committee. Because of the greater need for backtracking in the purely combinatorial algorithms, the supporting architecture should be biased toward making fast context swaps. This capability will be useful as well for expert systems,but will probably not be as critical here as for the support of backtracking in combinatorial algorithms.

Both types of problems are highly data dependent. Undoubtedly, a small portion of code might constitute the inner loop of the computation in either case, but data accessed by that code changes frequently in time and is rather data dependent. The committee at first described this condition as "poor predictability," but the wording subsequently changed to "unknown predictability." While it is certainly true that the data dependencies prevent the programmer from knowing precisely what paths will be traversed before the code is actually run, it may well be possible to make good predictions of the future from knowledge of the recent past. Hence, there may be architectures, much like cache memories, that predict future behavior as a function of past behavior and take advantage of such predictions to enhance performance.

Both types of programs shown in Table 2 are likely to have a lower ratio of I/O activity in symbolic computation. Memory management may have a considerable impact on I/O structure, but its characteristics for symbolic programs is still not well understood. Unlike numeric programs, the bottlenecks in symbolic programs are not easily identified. Although some subprograms may be executed repeatedly, and these could be identified in advance, the specific data used by those routines is not easily predicted, and therefore it is very difficult to bring data in advance to the computations that require the data.

There appear to be many opportunities for partitioning symbolic programs into smaller modules that could be executed in parallel. The ability to run parallel partitions does not in itself guarantee high performance because a multiprocessor could well spend the bulk of its computational activity on redundant computations. Hence, although the ease of partitioning makes the multiprocessor architecture an attractive candidate for symbolic programs, the necessity for limiting the amount of redundant computation suggests that it may not be an easy task to obtain large speed-ups this way.

Another major difference between Tables 1 and 2 is that the numeric codes require a very fast floating-point engine, whereas the arithmetic for symbolic programs tend to be heavily biased toward arithmetic comparisons. Also, the symbolic computations, particularly the programs that do many backtracks, perform a significant number of context switches, which are less likely to occur in the large-scale numeric codes.

Now consider Table 3, which shows three important types of interactive Artificial Intelligence programs. The programs represented by this table have additional properties beyond those given in Table 2. Architectures for the programs in Table 3 might well contain specialized processors for the unique aspects of the programs identified in the Table 3 plus other processing capability directed to the needs identified in Table 2. Note that all three types of programs described by Table 3 perform considerable computation while engaged in sensing and controlling real-time activities.

4.3 Very Large Data Base

Table 4 indicates the primary characteristics of data-base programs. Note that data-base characteristics are present throughout the symbolic computations described in the previous section. Consequently, the data-base aspects of such computations might be partitioned from the symbolic computations, thereby permitting a specialized data-base processor to work in conjunction with a symbolic processor.

4.4 Summary

To re-iterate, we feel that there are major differences between the various sorts of large-scale computations. Therefore, developing and using specialized processors for each of the following computation type seems to be desirable.

- **Numeric types.** Large-scale numerical algorithms could feasibly run on MIMD architectures. Numerical programs appear to be partitionable and amenable to implementation on MIMD architectures.

- **Symbolic types.** Symbolic programs could be partitioned into smaller modules that could be executed in parallel in a multiprocessor. Unfortunately, this partitioning may result in redundant computations that would slow the final processing time down.

- **AI types.** Interactive artificial intelligence programs could run on architectures that contain specialized processors for their unique aspects and also have processors for their symbolic computations.

- **Data base types.** Partitioning the data base aspects of a symbolic program would allow a specialized data base processor to work in conjunction with a symbolic processor.

It is unlikely that a single general-purpose processor would suffice for all these large scale computations.

TABLE 1

Characteristics of Numeric Programs

	Mesh	Nonmesh
Structure	highly iterative	highly iterative, with substantial data dependency
I/O-to-compute ratio	low, O(N)	low, O(N), less than mesh
Computaional Complexity	$O(N^2)$ or less	$O(N^2)$ or less
Predictability	high	high
Bottlenecks	known in advance	known in advance
Arithmetic	heavy	heavy
Data Size	very large	very large
Partitionability	opportunities, with schedulable I/O	opportunities, but I/O is irregular

TABLE 2

Characteristics of Symbolic Programs

	Expert System	Theorem Prover (Combinational Search)
Structure	data dependent, decision tree traversal	data dependent, combinational search, backtracking
Basic step	pattern matching, possible combinational search; rules tend to reduce search	pattern match, change data, schedule next step, choose one
I/O-to-compute ratio	possibly very low	possibly very low
Computaional Complexity	possibly very high	possibly very high, higher than for expert system
Predictability	unknown	unknown
Bottlenecks	not known in advance	not known in advance
Arithmetic	comparison intensive	comparison intensive
Data Size	moderate, working memory unknown	moderate, working memory unknown
Partitionability	opportunities exist	opportunities exist
Memory management	very complex	very complex

TABLE 3
Characteristics of Vision, Speech and Robotics Programs

	Vision	Speech Understanding	Robotics
Structure	expert + real-time + image processor (numeric) + feature extractor pattern recognition	expert + real-time + speech processor (numeric) + feature expractor pattern recognition	expert + real-time + numeric processor vision system + feedback control (sensors and transducers) + size and weight constraints
Other Properties	data intensive per data point, many production rules	heavy computation	

TABLE 4
Characteristics of Very Large Data-Base Programs

Data Size	very large
Computation type	touches only a small part of the data, variable amount per item accessed
Predictability	data-dependent accesses could have poor locality
Partitionable	opportunities exist
Input/Output	potentially heavy
Other	Index creation, maintenance

5. The Fine Granularity Working Group

Howard Brauer, IBM
George Cox, Intel
Lanny Forgy, CMU
Bert Halstead, MIT (Group leader and author)
Carl Hewitt, MIT
Anita Jones, CMU
H.T. Kung, CMU
Chuck Seitz, Caltech
Sal Stolfo, Columbia

5.1 Fine Granularity Defined

The group had some difficulty arriving at a suitable definition of "fine granularity." Such a definition was vital, since the group's charter was to consider issues related to fine granularity machines. Among the proposed definitions were

- Ephemerality of tasks. A fine-granularity machine is characterized by tasks that only execute a few instructions during their lifetime.
- Frequent communication/interaction between tasks.
- Parallelizing innermost loops. "Fine granularity" is an approach to computation in which concurrency is sought between executions of the bodies of innermost loops, rather than outermost loops (this would be "coarse granularity").
- Large communication/computation ratio. Granularity is fine when the amount of communication is relatively large, compared to the amount of computation.
- "Tight coupling." This definition attempts to avoid the issue by defining one buzzword in terms of another!
- Large degree of parallelism. Finer granularity programs tend to have a larger number of concurrent tasks.
- Small physical processing node size.
- Short code size for program modules.
- Few bits of state per node, or per program module.

A majority of the working group, though not the chairman, preferred the definition of fine granularity as "small physical processing node size." One conclusion from this exercise is that the term "fine granularity" should be used with caution in the literature, as it tends to mean many different things to different people.

The working group reached two related conclusions: fine granularity is a relative term (thus the group did not attempt to define a precise threshold for "fine granularity"), and the granularity observed may vary according to one's point of view. The table below was produced during the group's meeting, illustrating three levels of detail (application programming level, system software level, and architecture/hardware level), and showing that the same machine might appear to have different scales of granularity at these different levels (key: CRAY = Cray-1, DADO = Stolfo's DADO machine, HEP = Denelcor HEP-100, 432 = Intel 432).

Granularity:	Fine	Coarse	None
Application level		432 HEP	CRAY DADO
System software level	DADO	432 HEP	CRAY
Architecture and hardware level	HEP CRAY DADO	432	

The location of many of these machines on the chart is somewhat arbitrary (depending, in particular, on which definition of "granularity" is used); however, some interesting patterns are evident. The Cray-1 appears as a sequential machine (as far as correctness is concerned, not performance) to application and system level software, but its hardware uses pipelining in a way that can be called fine-grained, according to several of the above proposed definitions. In the case of the DADO machine, the application language is Prolog, where no parallelism is evident. But system software and the hardware are both cognizant of a large amount of possible parallelism.

5.2 Issues in Fine Grain Computing

The fundamental issues relating to computing using physically small processing nodes were identified as follows:

- Physical communication costs, as well as the desire for more parallelism, favor smaller and more numerous processing nodes, rather than a small number of large nodes.
- Working set considerations favor larger processing nodes.
- Smaller nodes will lead to greater amounts of communication.
- Massive communication is hard to deal with, so it would be best to shield the user from it.

Research that ought to be done in this area includes

- Determine the node size that produces the best balance between the considerations that favor larger vs. smaller nodes. This optimum size can be expected to vary according to the application, but if the optimum size for several representative uses were known, better conjectures could be formed about the optimum size for other uses.

- Design and implement languages to hide the small physical node size from users. It was generally agreed among the group that, although implementing suitably efficient languages may be a difficult task, the utility of fine-grained machines would be much enhanced by good languages that conceal the communication demands imposed by small node size.

- "Standard," or benchmark, applications are needed for calibration. Development of relevant fine-grained machines would be aided by the availability of a set of applications generally agreed upon as interesting.

- Fine-grained machines should be built. It is important to build software first, and analyze the expected performance where possible, to avoid committing resources to unproductive designs. On the other hand, the nature of fine-grained machines is such that thorough simulation with interesting application data sets will usually be computationally infeasible.

- "Hardware testbeds" that attempt to model the hardware of proposed fine-grained machines at a low level would probably be fairly specialized to a particular proposed architecture, and take longer to build than the proposed fine-grained machines themselves, and are probably not worthwhile.

- Software testbeds, where algorithms and languages may be bread-boarded at a higher level, may be useful.

5.3 Current Research Capabilities

Actions that can be taken today are to (1) learn more from existing fine-grained machines, and (2) develop languages and applications on existing testbeds. The group identified the following existing or soon-to-exist fine-grain machines:

- DADO (Columbia).

- Cosmic Cube (Caltech). This consists of 64 8086/8087 processors.

- Systolic array testbed (Naval Ocean Systems Laboratory). This uses an 8-by-8 array of 8086/8087 processors, but takes 100 microseconds to simulate one step of a systolic array!

- 31-element Mosaic tree (Caltech). This will exist soon.

The only existing testbed for language and application experimentation on a sufficiently large scale is the Cm* machine at CMU. Several MIMD machines proposed or under construction were

mentioned as additional possibilities. In the chairman's opinion, a desirable testbed should have at least 20-30 processors, and should be able to support a shared-memory model of computation for maximum flexibility.

6. Conclusions

Although group discussion topics of the workshop on Multiprocessors for High Performance Parallel Computation were varied, several groups produced similar recommendations. Attendees of the conference were representative of a broad cross-section of the research community. Consequently, the conclusions reached by the group can be regarded as a strong statement by the research community, in general, and not the opinion of a special interest group.

The major theme that appeared in several group recommendations was

the need for for actual experimentation on multiprocessors, and

communication of experimental results and concommitant insights in a form useful to researchers other than the original experimenters.

The overall recommendations can be grouped and summarized as follows:

1. **Build.** Build a variety of multiprocessors. Only experience using actual multiprocessor systems will lead to a scientifically credible understanding of multiprocessors as a vehicle for high performance computation. At this point in research the multiprocessor remain the leading contender for providing very high performance parallel computation in the future. Two strategies were proposed.

    - Develop a national Experimental Multiprocessor Laboratory that is
        - conducive to experimentation with new architectural concepts,
        - well-instrumented,
        - equipped with support for parallel models and languages,
        - easily accessible to multiple research communities, and
        - supportive of a variety of application developers.
    - Develop multiple experimental laboratories at different research sites, each developing different architectural concepts or focussed on different applications such as: general purpose, numeric, symbolic, and database.

    In either case, it is desireable to reduce both the amount of effort and the elapsed time required to suitably design, construct and evaluate prototype systems.

2. **Experiment.** Develop a wide variety of algorithms and full applications, instrument and measure their behavior in all dimensions. This requires building not only actual multiprocessor systems, but the substantial support environment that allows programming, instrumenting and measuring the programs. This requires further development of

    - languages for expressing parallelism,

- methodologies for experimentation with parallel applications,
- methodologies for the design and implementation of parallel processor testbeds,
- methods for measurement that is conducive to comparison across multiple systems.

In addition, experimentation requires the investment of considerable resources to build the necessary testbed support software that makes experimentation tractable.

3. **Evaluate.** Evaluation of experiments requires the development of better models and measurement techniques. These include
   - models of parallel computation,
   - models of multiprocessor systems,
   - characterization of workload,
   - techniques to enhance our ability to observe behavior within testbeds, and
   - system-independent benchmark algorithms.
4. **Communicate.** A current weakness in reseach in parallel computatinon to date is that researchers starting from different premises have had great, usually insurmountable difficulty, in comparing their results with results of others. Two recommendations were made that would improve communication within the research community:
   - Develop a knowledge base about benchmark programs, their analysis and actual experimentation results for each. The database would include an associated query system. An interest group should be formed to conduct network dialogs, augment the database and develop common, critical annotation of the benchmarks.
   - If a national laboratory were formed, it should be used not only to develop a common working environment for multiprocessor researchers, but also to act as a depository for accumulated wisdom, experience, and experimental results. The staff of the laboratory would then be equipped to provide a consulting service to application developers. A laboratory supported system could be replicated at multiple research sites so that researchers at other sites could perform experimentation, yet not incur the heavy cost of the development of their own experiment supporting environment.

A number of multiprocessor architecture implementations actually existed at the time of the workshop, but very few had the substantive additional support required for performing and instrumenting extended experiments as proposed above. More multiprocessor systems are needed. Experimentation with existing system must continue.

Table of Contents