4-8-1994

Performance Evaluation Tools for Interconnection Network Design

Anna Kolinska
Portland State University

THESIS APPROVAL

The abstract and thesis of Anna Kolinska for the Master of Science in Electrical and Computer Engineering were presented April 8, 1994, and accepted by the thesis committee and the department.

COMMITTEE APPROVALS:

[redacted]
Michael A. Driscoll, Chair

[redacted]
W. Robert Daasch

[redacted]
Jingke Li
Representative of the Office of Graduate Studies

DEPARTMENT APPROVAL:

[redacted]
Rolf Schaumann, Chair
Department of Electrical Engineering

****

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by [redacted] on 8 June 1994

ABSTRACT

An abstract of the thesis of Anna Kolinska for the Master of Science in Electrical and Computer Engineering presented April 08, 1994.

Title: Performance Evaluation Tools for Interconnection Network Design.

A methodology is proposed for designing performance optimized computer systems. The methodology uses software tools created for performance monitoring and evaluation of parallel programs, replacing actual hardware with a simulator modeling the hardware under development. We claim that a software environment can help hardware designers to make decisions on the architectural design level. A simulator executes real programs and provides access to performance monitors from user's code. The performance monitoring system collects data traces when running the simulator and the performance analysis module extracts performance data of interest, that are later displayed with visualization tools. Key features of our methodology are "plug and play" simulation and modeling hardware/software interaction during the process of hardware design. The ability to use different simulators gives the user flexibility to configure the system for the required functionality, accuracy and simulation performance. Evaluation of hardware performance based on results obtained by modeling hardware/software interaction is crucial for designing performance optimized computer systems.

We have developed a software system, based on our design methodology, for performance evaluation of multicomputer interconnection networks. The system, called the Parsim Common Environment (PCE), consists of an instrumented network simulator that executes assembly language instructions, and performance analysis and visualization modules. Using PCE we have

investigated a specific network design example. The system helped us spot performance problems, explain why they happened and find the ways to solve them. The obtained results agreed with observations presented in the literature, hence validating our design methodology and the correctness of the software performance evaluation system for hardware designs.

Using software tools a designer can easily check different design options and evaluate the obtained performance results without the overhead of building expensive prototypes. With our system, data analysis that required 10 man-hours to complete manually took just a couple of seconds on a Sparc-4 workstation. Without experimentation with the simulator and the performance evaluation environment one might build an expensive hardware prototype, expecting improved performance, and then be disappointed with poorer results than expected. Our tools help designers spot and solve performance problems at early stages of the hardware design process.

PERFORMANCE EVALUATION TOOLS FOR INTERCONNECTION NETWORK DESIGN

by

ANNA KOLINSKA

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

Portland State University
1994

ACKNOWLEDGEMENTS

I would like to express my heartfelt thanks to Dr. Mike Driscoll, my advisor and friend, for his patience, continual guidance and intellectual support. His help and encouragement directly contributed to my successful graduate work at Portland State University. I am grateful for the tremendous amount of time that he spent guiding me in my studies and research, for his invaluable ideas and for revising and proofreading my thesis. The experience of studying and working with him is precious and unforgettable.

I would also like to thank Dr. Robert Daasch and Dr. Jingke Li for serving on my committee and for their constructive comments and suggestions about the thesis.

I owe my grateful thanks to faculty and staff of Department of Electrical Engineering for their kindness and helpful attitude.

Finally, my special thanks goes to my family. To my husband, for his friendship, encouragement and support during my years at PSU. For his patience when trying to understand the intricacies of my graduate work. To my brother and his wife, without whom it would not be possible for me to study at Portland State University. I am grateful to them for giving me a great start, spiritual and financial support, and all other help when I needed it most. I would like to give my special thanks and gratitude to my parents for all their love and commitment throughout the years. I am thankful to them for teaching me the most important values: love and appreciation of people and science.

Portland, Oregon
April 1994

Anna Kolinska

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

In this thesis we propose a methodology for designing performance optimized computer systems. Our approach is based on the software tools used for performance monitoring and evaluation of parallel programs. We add performance monitors to a simulator modeling a hardware design. We can collect data traces by running performance instrumented application programs. With analysis and visualization tools we can examine data traces with respect to different architectural choices, and tune the hardware design accordingly.

Based on our methodology we have created a prototype performance evaluation system intended for hardware designers developing multicomputer interconnection networks. To prove that performance problems in hardware designs can be solved using software development tools, we have used our system to solve a specific network design problem. We have performed a set of experiments to investigate how many PE channels should connect a processor to a router in each node in a mesh network to obtain the best performance. Our assumption was that the neighboring nodes are connected by one network channel, which is a common practice in building multicomputer systems [1,2]. Because each node in a mesh network has two to four neighboring nodes, we expected that adding up to four PE channels would significantly improve performance of the network.

After examining the problem, it turned out that our intuition was misleading. The number of PE channels is not the most significant factor determining network performance. The most important factor is the efficiency of algorithms used to generate communication patterns in the network. Our evaluation tools helped us to set up performance experiments and analyze their results, leading to a solution of the design problem. Hence we have validated our design

methodology.

A generic software system using performance evaluation methods for designing computer systems is intended to help engineers to make decisions at the architectural design level. A key component of such a system is a simulator. It is used to model hardware under development. As with the real computer system, application programs are used as the input to the simulator. A designer interested in system behavior during program execution needs to obtain performance data from the simulator. To provide a way to look into the behavior of a given system, a simulator needs to be instrumented with performance monitors. The information should be accessible on the application level to enable collecting data traces of interest for further analysis. To use the performance monitoring capabilities of a simulator, application programs need to contain performance monitoring code. That code would pick up the values of specific simulator probes during program execution and decide how to compute and output performance data traces. The code development stage can be supported by a performance instrumentation system for application programs. Using such a system, insertion of performance monitoring code into the programs can be automated, thus relieving the designer from the tedium of manually customizing the code for every performance experiment. Data traces collected during the simulation run need to be analyzed to obtain the performance information. With a performance analysis system the designer can filter the performance data from collected traces and apply specific functions to them. The final, analyzed set of data may be presented using visualization tools.

Parsim Common Environment (PCE) is an implementation of such a generic performance evaluation software system intended for hardware designers developing multicomputer interconnection networks. It contains a network simulator, performance instrumentation system, and performance analysis and visualization tools. Using PCE it is easy to measure the performance of a given network configuration while executing one or more programs and observe generated traffic patterns. With the instrumentation system, the user is given the flexibility of defining

events and measurement points to be monitored. The visualization tools let the designer thoroughly analyze and understand the collected data traces, and make conscious decisions about the hardware to be built before more substantial commitments are made. Our methodology is not restricted to multicomputer network design. With our approach one can solve many performance problems in computer architecture design.

MOTIVATION FOR A NEW HARDWARE DESIGN METHODOLOGY

A lot of effort has been put into the development of tools that allow tracing execution of parallel programs and monitoring utilization of processors, for further improvement of program performance. However, programs cannot perform any better than the underlying hardware. Hence performance optimization has to be tackled in both the areas of hardware and software design. This is especially true for parallel computers. Performance of a computer can be enhanced at five stages [1]:

1. machine design,
2. algorithm design,
3. data structuring,
4. compiling stage,
5. fine tuning.

The machine design stage is the most important part in the process of enhancing computer performance. The goal is to optimize the architecture and operating system to yield high resource utilization and maximum performance. Once all design decisions are made and the machine is built, it is no longer possible to alter the behavior of hardware. This stage is very laborious and many tests and prototypes are required to decide on optimal hardware configuration.

Many scientific computer systems have been developed in the last three decades. Designers have been taking advantage of changes in electronic technology. They have worked on making both CPU's and computer peripherals faster. Clock cycle time of the processors and memory access time have been shortened, and cacheing and pipelining techniques have been introduced, to name a few of the enhancements. However, the advances in uniprocessor systems are nearing their limits. New improvements do not change the performance significantly, they only raise costs. Concurrent processing has become a step beyond the technological limits that have been reached by uniprocessor systems.

It is very hard to estimate the performance of a parallel design. With uniprocessor systems the design process was incremental. New solutions were based on the previous designs, and it was relatively easy to make good predictions of the performance of a newly developed system. When improving a uniprocessor system, many design choices are implied when analyzing similar machines. One can estimate the performance gain in the modified design. The situation is much different with the parallel machines. Every design is unique and therefore estimates cannot be made through comparison with similar architectures, because they do not exist. Good predictions of the performance of innovative architectures are impossible, since the changes in the designs are revolutionary, and do not constitute mere improvements of existing solutions.

However, engineers are not totally helpless in the process of hardware development. Predictions of behavior and performance can be made based on the results of simulations of the hardware. Even though developing a simulator is quite expensive — it takes a lot of time to model hardware in the right way, it is usually worth the effort [3]. Generally, simulators are very flexible. Design parameters can be easily varied, leading to good solutions. One disadvantage of simulators is that they provide only a good approximation (at best) of the hardware behavior. Simulation results may prove to be inaccurate for machine as constructed. Other disadvantages are that simulators are usually very slow and that their results are hard to interpret.

The next four stages of performance improvement mentioned above are in the area of software. At the stages of algorithm design and data structuring, programs have to be matched to the target hardware. At the compiling stage application code should be optimized not only for concurrency and vectorization, but also for scalar operations. The fine tuning stage closes the loop of software development. At this stage program performance is monitored and analyzed. Based on the obtained results programmers can verify their programs, and redesign them starting over from the algorithm or data structuring stage.

In a parallel computer, a single CPU has been replaced by many processors, each of which could execute different sets of instructions in parallel. CPUs have to cooperate to get the task done. To use parallel machines effectively, appropriate software tools have to be provided. Operating systems for new machines have been created along with specifications of the parallel languages and compilers. Later advances brought integrated environments for software development like parallel debuggers, profilers, and performance instrumentation and visualization tools, all of which help programmers immensely in development of software.

DEVELOPING HIGH PERFORMANCE SYSTEMS

To illustrate the differences in the process of enhancing the performance of parallel computer systems which are experienced by different specialists, consider the cases of software and hardware engineers.

A software engineer working on parallel applications has been aided by integrated environments for program development and performance tuning. An example of such an integrated system is The Pablo Performance Analysis Environment developed at the University of Illinois [4]. Each of the design stages is supported with appropriate tools. First, debuggers help to develop logically correct programs. They play an important role helping designers on the algorithm design stage. Once written, code can be run and analyzed to detect loops and other potential parallelism in the program structure. This data structuring stage is supported by compiler

preprocessors that insert monitoring probes into application programs. Using such probes, information about execution times of various regions of code can be obtained, as well as the number of occurrences of specific events. Further vectorization of the program is achieved at the compiling stage. The designer cannot greatly impact the performance of this stage, but depends on the functionality of the compiler.

At the fine tuning stage the designer is aided by various run-time monitors, showing utilization of processors during execution of the program. Moreover, post-processing performance analysis modules can analyze collected data traces. Performance visualization tools let users view the collected data as graphs and pictures easily understandable by humans. All of these tools help users develop programs that are optimized with respect to execution time and also utilization of processors in the multiprocessor system. Through several iterations the process converges to the optimal solution. Different versions of programs can be easily compared and the best one can be chosen as the final product. The integrated environments for parallel programming make the job of software engineers much easier and less stressful than it used to be when the only available tools were compilers. Nowadays, moving through the design loop is fast and not very troublesome.

On the other hand, looking at the job of a hardware designer who works in the area of parallel computing, it does not seem as easy as that of the software engineers. It is true that there are a number of systems developed for monitoring the performance of hardware. These are usually hybrid systems — a mixture of hardware and software monitors. Such systems are valuable in understanding and interpreting the behavior of parallel systems. However, they play an important role only at the end of the design process. Therefore, any revisions of the design are very expensive, and actually engineers can only guess what parameters to change to obtain a better performance of their system. Comparisons between designs may be performed only after several prototypes have been created. As described, the hardware development process is very

time consuming and expensive. Usually only a couple of options are checked. Therefore it is highly possible that the obtained architecture is not optimal. And since software cannot perform any better than the hardware, optimization on the hardware level is crucial.

Simulators make hardware development a little easier, because they help predict the impact of parameter changes on design behavior. Once a simulator is available many tests can be performed before choosing the final, and hopefully optimal, configuration. However, a simulator itself is not a sufficient tool to make the hardware engineer effective. Simulator results are large sets of data traces that are very hard to interpret. Any data analyzers that have been developed are application dependent and cannot be ported to other simulators. Generally, the only tools available for hardware development are low level circuit simulators. On the architectural design level engineers are still forced to investigate different configurations of hardware without the help of good tools.

We believe that the same set of tools that help software engineers to develop performance optimized code will also be useful for hardware designers to create performance optimized computers. An integrated performance evaluation system with the simulator used in place of a computer will help hardware engineers to spot performance problems in the developed systems and speed up the design process.

THESIS OVERVIEW

In this chapter we introduced a methodology for performance evaluation of hardware designs. We presented an overview of experiments performed to solve a specific design problem using software performance evaluation tools of our creation. The tools are intended for hardware designers developing multicomputer interconnection networks.

We also presented the steps leading to performance optimized systems and underlined why achieving good hardware performance is crucial. We described the tools that support

engineers in the process of optimizing parallel software and also that of parallel hardware. We showed how the methodology that we used for development of parallel hardware fills the need for designing performance optimized systems.

In the remainder of this thesis we validate our claim that the software tools can be used for performance evaluation of hardware designs and can help develop performance optimized computer systems by using the tools to solve an example problem and evaluating how helpful they were in the design process. First we present background needed to understand the design problem.

Chapter II describes software and hardware performance optimization techniques. We present several existing systems for performance evaluation of parallel software and hardware.

In Chapter III we concentrate on designing parallel hardware, focusing on multicomputer interconnection networks. We introduce the nomenclature that will be used to discuss the system that we created for monitoring performance of interconnection networks. Definitions of performance of parallel computer systems are also provided.

Chapter IV describes in detail the methodology for designing performance optimized computer systems. We present the PARSIM Common Environment (PCE) — a software performance evaluation system intended to support engineers in hardware development process of multicomputer interconnection networks.

An example network design problem, its analysis and solution using PCE are shown in Chapter V.

Chapter VI concludes the thesis with the discussion of the design process using PCE. We discuss the limitations of the system and future work to improve the efficiency of hardware design process.

CHAPTER II

PERFORMANCE EVALUATION SYSTEMS

Performance and cost are the two most important factors that determine if a new computer is a marketing success. These factors heavily impact each other, and most designers analyze them together. The cost/performance ratio is the basic point of interest for both customers and designers.

However, while designing commercial systems the upper cost boundary has to be set beforehand, and cannot be exceeded. This guarantees that the product that is intended for the predefined range of customers will be within the parameters of the targeted portion of the market. Also, new performance options are not implemented if they raise the cost/performance ratio of the final product. Thus, we will focus on performance of parallel systems. In this chapter we will describe existing systems for performance evaluation of parallel software and hardware.

Performance depends on the degree of utilization of the parallelism in hardware as well as in software. Figure 1 shows the steps leading to the overall performance of the concurrent system. Both areas are very important: hardware, because programs cannot run any faster than hardware lets them to, and software, because even the highest computational power may be wasted if the software is inefficient.

Software Performance Optimization.

Software performance has become the area of interest for the growing population of users of parallel systems [6]. Historically, not many performance monitoring systems were available, because vendors considered performance tools a luxury. The tools were expensive to develop,

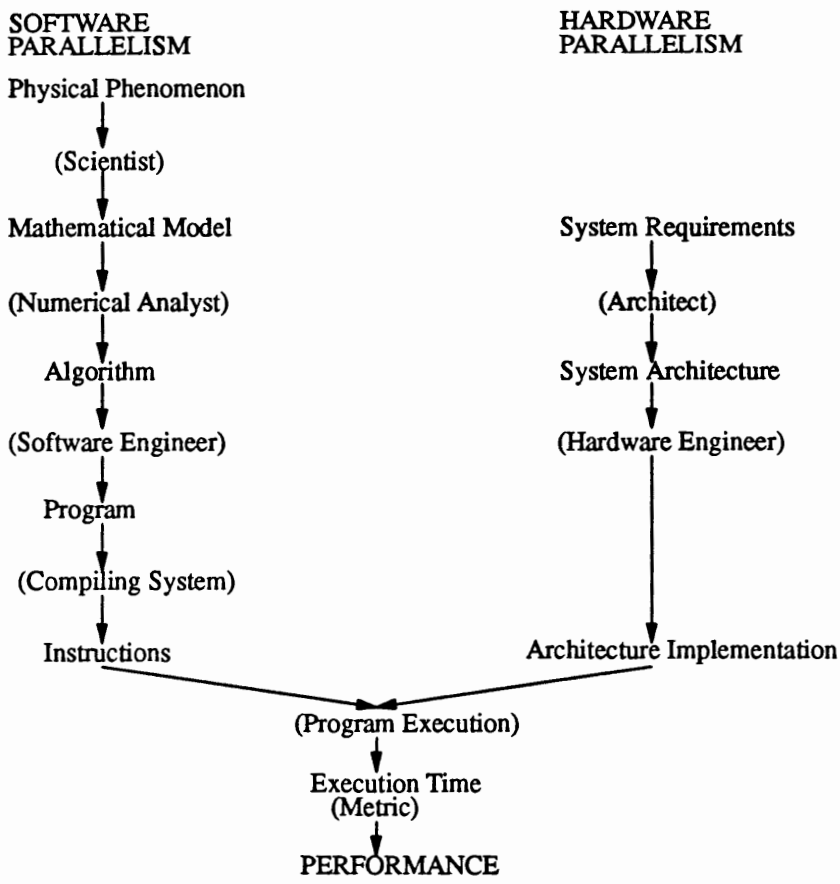


Figure 1. The steps leading to performance [5].

and not needed to sell parallel machines. Good tools appeared only if the machine succeeded. Today, when many diverse machines are becoming available, competition on the market is much stronger. Performance tools are needed for marketing, and are necessary for a machine to become a success [6]. This is why a lot of effort is dedicated to building environments for software development and optimization.

Traditionally, users of uniprocessor systems were aided by debuggers, to spot the logical errors in their code, and profilers, to analyze the performance of specified parts of the program. These tools were usually sufficient to satisfy the requirements of software engineers.

Parallel software development, on the other hand, is more difficult. None of the tools used for uniprocessor systems could help software designers. Parallel debuggers and performance

monitoring systems have to be much more complex to effectively aid parallel programmers. Software engineers need to understand details of the architecture and behavior of parallel machine, to be able to interpret the results and improve program performance.

A generic performance evaluation system consists of a performance instrumented computer and a performance analysis and visualization system. Such a performance instrumented computer is shown in Figure 2.

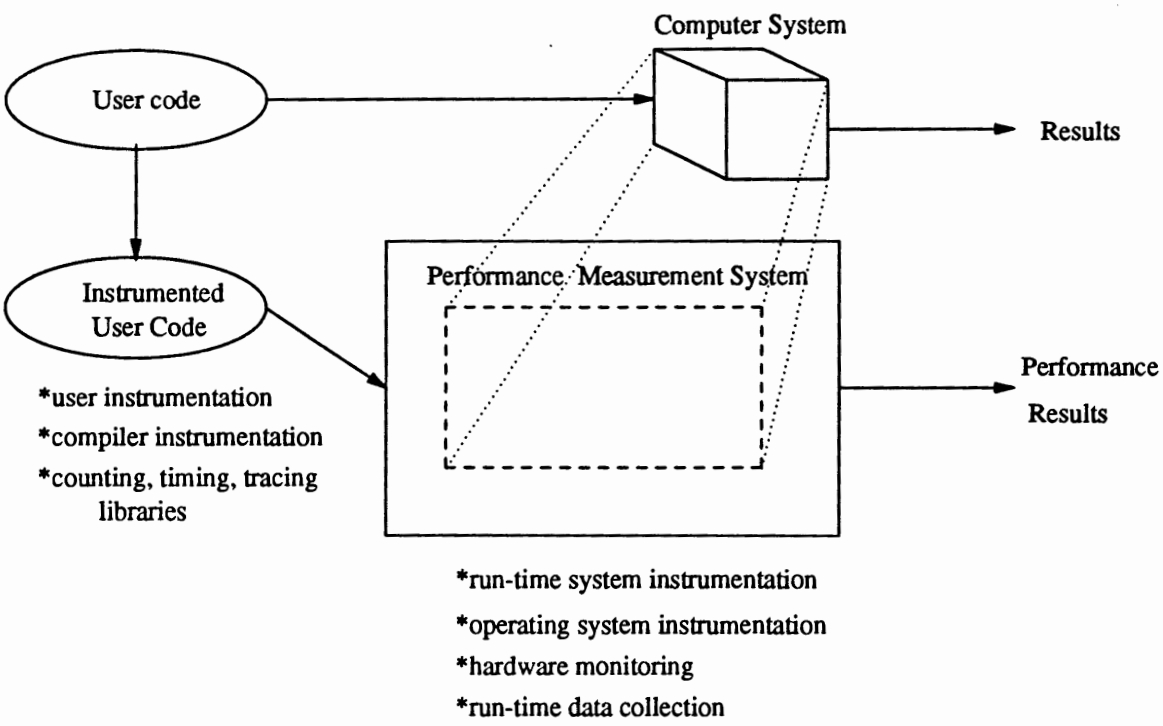


Figure 2. Performance instrumented computer [6].

The measurement system is usually hardware based or a hybrid (software and hardware), and provides information on processor utilization, time stamps, link utilization, message length, type, send and receive events, and more. Instrumented code defines events for observation, what action needs to be taken upon their detection and what data traces are to be collected.

Results obtained from the program run are further analyzed and presented with performance analysis and visualization tools. Such tools use a set of advanced graphical displays to

present gathered data. Examples of such displays are: two-dimensional line and scatter plots, two-dimensional surface and contour plots, three-dimensional line plots, analog and digital dials and meters, PIE charts, event graphs and execution graphs [6]. The new techniques for data presentation, still very immature, are data animation and sonification.

Integrated environments for performance evaluation help users to determine the performance of a given program and to improve it. To do so, a set of experiments has to be performed.

A typical performance experiment consists of three phases [6]:

- specification,
- instrumentation and data collection,
- data reduction and presentation.

In the specification phase the user needs to decide what kind of information is significant to understand the program behavior, and how it relates to overall performance. Once the problem is stated, code needs to be instrumented. The data collection phase poses problems because of the huge volume of data generated during the experiment. One needs to carefully estimate the volume of data to be generated, to fit it successfully within the available resources (disk space).

To enhance understanding, the collected data traces need to be further reduced, and viewed using available displays. A wide variety of presentation techniques lets the user get an insight into locating performance problems and leads to proper conclusions.

> *"...visualization has become a powerful, almost indispensable mechanism for the scientific user. Interactive visualization combines computer graphics and imaging with user interfaces to aid understanding of complex computation. As a tool, visualization provides a time-efficient method for testing and debugging ideas as well as a gauge for performance."* [7]

Example Performance Evaluation Environments.

The IPSC/2 supercomputer is aided by a set of performance tools [8]. HYPERMON is a hardware-assisted monitoring system. A modified GNU C compiler is used for software instrumentation with command line options enabling generation of instrumented code. The data analysis and visualization system contains a user interface, a set of generic data analysis filters, a set of stainers (filter-display interfaces), and a set of display views.

The Pablo Performance Analysis Environment [4, 6] is a toolkit consisting of performance data instrumentation tool and a performance data analysis and presentation tool. Instrumentation allows for graphical event specification, source code instrumentation and portable data capture. The presentation tool consists of graphical programming modules, data transformation modules, self-defining data format, dynamic graphics and sonification.

Pablo is a portable system. It is intended for use with any monitoring system that supplies the data format used by Pablo. Currently, the data capture library supports the Intel iPSC/2, iPSC860, Thinking Machines CM-5, Intel Paragon and uniprocessor Unix systems [6]. Many other performance evaluation tools have been described in [6, 7, 9, 10] but the two described above are typical.

Hardware Performance Optimization.

> *"Despite continued technical advances, parallel system design remains ad hoc, an art form practiced by small cadre of experienced, highly valued designers. No known, general purpose methods can predict the performance of a proposed system design."* [8]

It is true that parallel hardware performance issues are important to a small number of scientists and designers developing concurrent systems. The quote might lead one to think that the performance of concurrent systems is only a matter of luck. Is the situation really so dramatic?

There are several approaches to estimate the performance of a parallel machine [11]:

- conduct a mathematical analysis which yields explicit performance expressions,
- conduct a mathematical analysis which yields an algorithmic or numerical evaluation procedure,
- write and run a simulation,
- build the system and then measure its performance.

All of the above methods are widely used, and designers base their decisions on the obtained results.

However, all the methods are far from perfect. The systems to be modeled are usually very complex, hence mathematical descriptions are not exact, and yield approximations. Performing simulations is usually sufficient, if the design problem is well defined. Simple display methods can be used to present collected data. To explore more sophisticated problems, such as impact of different hardware configurations on system performance, simulation alone is not a good solution. Usually huge volumes of data are generated. Evaluation of collected traces is a very time consuming and difficult process. To make the data analysis easier additional software tools have to be provided. Measuring performance of the real machine has one significant drawback: it is difficult to make changes in the architecture to optimize performance.

According to Bradley [5], proper performance evaluation should lead to the desired understanding of performance using the following steps:

1. Design - What performance are we trying to measure and how do we measure it? Creation of experiments and methodologies for testing a performance hypothesis.
2. Observation - What happened in the experiment? Execution of the experiment, recording measurements, traces.
3. Analysis - What caused the behavior observed during the performance experiment?

4. Synthesis - How can we improve the performance? Providing feedback to hardware and software designers regarding new opportunities for higher performance.

We propose a software-based approach to hardware performance evaluation that satisfies the above criteria. Our methodology is based on existing performance optimization environments for parallel program development. Figure 2 applies to the system we have created, however we have replaced a real parallel machine with a simulator. We have also added some functionality to the user's code in that the user can configure the parallel hardware for exploration at the beginning of a program.

A simulator has built-in monitoring capabilities. We obtain an instrumented program by running a preprocessor that inserts monitoring code into a given application. Performance traces are collected during the simulation run. Later, they are analyzed and displayed using a simple performance visualization system. A comparison between different architectures can be easily made by providing a set of different configuration variables to be altered during the simulation. Simulation is repeated and data can be analyzed, displayed and compared. In Chapter V we will show how to solve a specific design problem using our system.

CHAPTER III

OVERVIEW OF PARALLEL HARDWARE

To understand the workings of our software environment for performance evaluation of interconnection networks, in this chapter we introduce key concepts of parallel systems, interconnection network design, and definitions of performance measures.

The tasks of a parallel program have to be mapped to many processors. Processors of the system, also referred to as nodes, have to perform the tasks simultaneously to gain computational speed. Many software applications, in particular those solving scientific and engineering problems, have inherent parallelism that can be exploited to improve the performance of those applications. A basic concurrent processing system is shown in Figure 3.

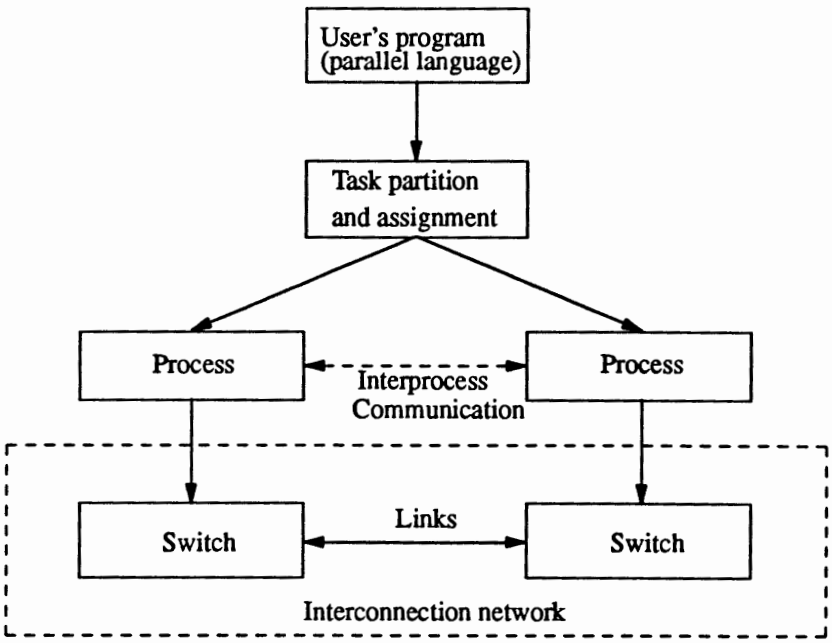


Figure 3. An overview of concurrent processing systems [12].

A user's program is partitioned into several processes that are assigned to individual processors. Interprocess communication is performed through an interconnection network. To exploit program parallelism efficiently, a distributed system must be designed to considerably reduce the communication overhead between the processors [13]. A general model of a parallel hardware system is shown in Figure 4.

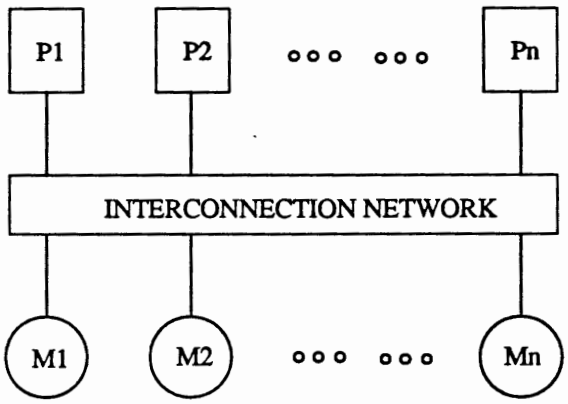


Figure 4. Hardware model of concurrent processing systems [12].

We can divide parallel computer systems into two categories: multiprocessors and multicomputers. The level at which interactions between the processors occur is the main difference between the two architectures.

Multiprocessors.

In a multiprocessor system all the processors must be able to directly share the main memory. All the processors address a common main memory space. From a programmer's perspective, code development for shared memory systems is easy, in fact it is almost the same as for uniprocessor systems. This is due to using common address space by multiprocessors. An example of a commercial shared memory architecture are Flexible Corporation's Flex/32 and Encore Computer's Multimax, introduced during 1980s [12]. Typically, in addition to main memory, each processor in a shared memory architecture uses a cache as a local memory. Although caches speed the execution of the programs, they introduce cache coherency

problems. Maintaining cache coherency for a large number of processors becomes impractical, because of the time needed to access memory elements that are not local to a given processor. Hence, these systems are not easily scalable.

The Stanford Dash multiprocessor [14] is a shared memory architecture that attempts to provide cache coherence without compromising scalability. The Dash system improves scalability by providing directory structures for maintaining cache coherency. These structures relieve the processing nodes from broadcasting every memory request to all processor caches, as the more common snoopy protocols would do.

In Dash the main memory is physically distributed among the nodes (clusters). Each cluster contains a small number of high-performance processors and a portion of the shared memory. Nodes are connected through an interconnection network. Access to the data blocks that reside in the remote memory is provided by passing messages over the network.

Hence the Dash architecture lays somewhere between multiprocessors and multicomputers. It provides the ease of use of the first and the scalability of the second.

Multicomputers.

In message-passing architectures, nodes share data by explicitly passing messages through the network. Processing nodes, consisting of an autonomous processor, local memory and a routing element, are connected via an interconnection network (Figure 5). The way routers are connected to each other is called the network topology.

Multicomputers have been developed primarily to provide scalable systems that will accommodate a significant increase in processors, and will satisfy the performance requirements of large scientific applications, characterized by local data references.

The scaling dilemma is solved because the processor and memory are physically localized in a node, and interprocess communication takes place less frequently than memory accesses.

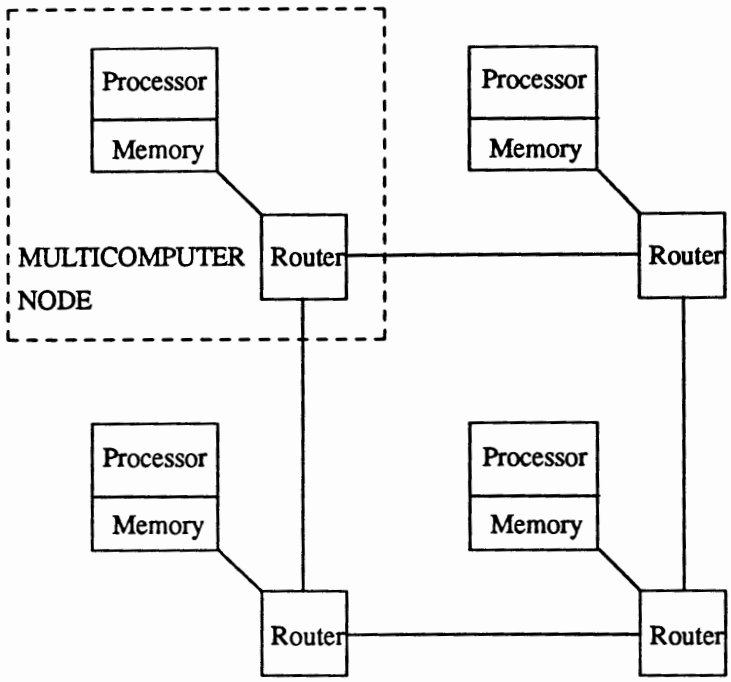


Figure 5. Multicomputer nodes in the mesh topology.

Thus, message passing networks are usually the choice for building massively parallel computers (consisting of thousands of nodes).

Design decisions are greatly influenced by the overall system cost. The goal is to maximize performance while minimizing cost to achieve the best cost/performance ratio. The cost of multicomputers is dominated by memory elements and interconnection network complexity (length of wires and router logic) [15, 16].

Conventionally, parallel computers have been built by replicating workstation-sized units, increasing processors and memory proportionally. In this approach, the cost of the machine is proportional to the number of nodes used. To obtain the best cost/performance ratio, Dally [17] proposes adding more processors to the network, while keeping the amount of memory constant. This way, performance of the machine is dramatically increased with little impact on its cost.

The MIT J-Machine is a multicomputer that is built using this approach. Each node of the network consists of a 32-bit processor, a floating point unit, a communication controller and a 512k bit RAM on a single chip. Having an on chip memory leads to fast communication with memory (reading a row of memory takes just one cycle) [18]. The J-Machine is often referred to as a fine grain machine, because of a little amount of memory per one processing node.

The J-Machine also supports the fine-grain programming model. Fine-grain programs consist of many short tasks as opposed to the coarse-grain approach, where programs consist of a few long tasks. Fine-grain programs usually produce much more communication traffic than coarse-grain programs. Hence, one must be very careful while designing a network to support fine-grain computation model.

Much research has been done to find an efficient network mechanism for communication between processors. The interconnection network plays a central role in determining the overall performance of multicomputer system; all other components depend on its performance. Latency, throughput and utilization are the most common measures characterizing network performance.

Latency is the time from when the first bit of a message leaves the sending node to when the last bit of the message arrives at the receiving node. Latency of a message, T_L, is defined as the sum of the latency due to the network and the latency due to the processing node [15]:

$$T_L = T_{net} + T_{node}$$

Throughput is the rate of message delivery (bits/s) when the network is fully loaded [17]. The goal is to minimize message latency while maximizing throughput. A measure of network utilization is link utilization, defined as fraction of time the links are occupied by messages.

An interconnection network consists of routers and wires. Wires are the physical medium connecting the routers and through which the messages are being sent. Routers are the network components that are responsible for routing the messages from source to destination. They also

allocate network resources and perform flow control of messages in the network. Routers usually contain some amount of memory to buffer messages when needed, and control logic to perform switching in the direction dictated by the routing and flow control algorithms.

Since an interconnection network is a key component in the concurrent system, we will focus on its different design choices and highlight their significance.

INTERCONNECTION NETWORK DESIGN

An interconnection network is characterized by its topology, routing, and flow control. The topology of a network is the way the nodes are arranged and connected to each other. Routing specifies how messages choose a path between source and destination nodes. A flow control strategy allocates channel and buffer resources to a packet while it is traveling through the network. It also resolves the conflicts between packets competing for the same network channels.

Topology of the Network.

There are many ways to connect nodes together. The most popular are shown in Figure 6. Mesh, torus and hypercube are special cases of k-ary n-cubes. Some other topologies used are: chordal ring and cube-connected cycles [16]. The network topology is usually characterized by its diameter, mean internode distance, and bisection width. Diameter of the network is the maximum shortest path between any two nodes [1]. The mean internode distance is the expected number of hops a "typical" message needs to reach its destination [16]. The bisection width of a network is the minimum number of wires that need to be cut to divide the network into two equal halves [15].

The bisection width is a measure of the wire density of the network, and helps to estimate its cost. The diameter of the network directly relates to the message latency. Early message-passing computers, e.g. the Cosmic Cube, iPSC, nCube [1], were connected by a hypercube network to achieve better performance (low network diameter).

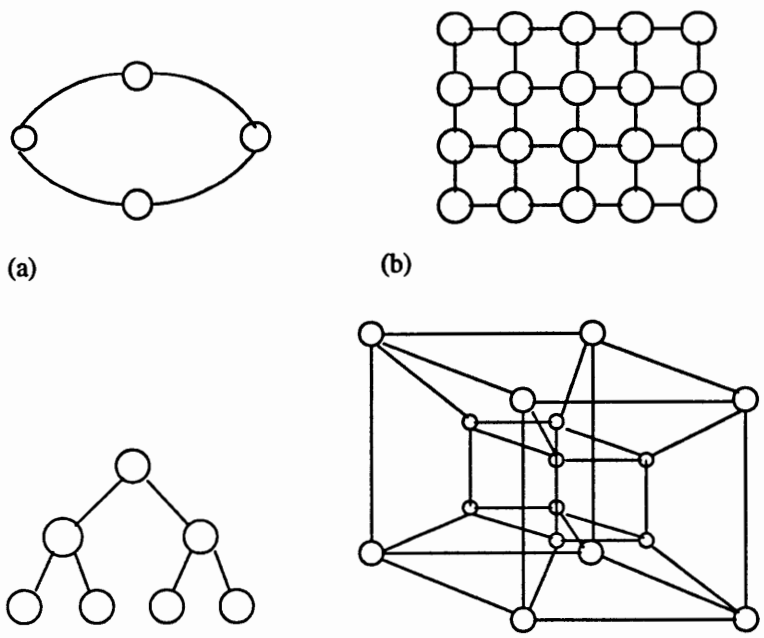


Figure 6. Distributed memory interconnection network topologies: (a) ring; (b) mesh; (c) tree; (d) hypercube [19].

However, the hypercube has proven to be inefficient (expensive) for machines with a large number of processors, due to the increased bisection width. Mesh and torus architectures have become a better alternative, even though they have higher network diameter for the same number of nodes. The latency introduced by additional channels to traverse could be ignored when employing efficient routing techniques like wormhole routing. Meshes are easy to map to the three physical dimensions, making efficient use of available wires. On the contrary, higher dimensional networks like hypercubes, needed additional wires to allow mapping to the 3-D plane.

The benefit gained from the low network diameter is illustrated by a comparison of two topologies of the same bisection width: a hypercube (high dimensional network, low diameter), and a mesh (low dimensional network, high diameter). The mesh has higher bandwidth per channel, since it requires less channels in the network. The distance between nodes is not an

issue when efficient routing mechanisms are used (wormhole, virtual cut-through). Examples of mesh or torus architectures are CM-2 and Intel Paragon [1].

Routing Algorithm.

Most multicomputer networks use the packet-switched transport mechanism, in which each message is divided into fixed-size packets that are routed separately through the network. Because packets are relatively small, they do not require a significant buffering space in each routing component. However, overhead is introduced for reassembling a message at the destination from the packets that may have arrived out of order [16]. Each packet consists of a number of flow control digits, or flits. A flit is the smallest unit of information that a channel can accept or refuse [15]. Only the head flit of a packet contains the routing and sequencing information. The end of a packet is marked in the tail flit of a packet.

Three basic schemes of routing messages are used in the multicomputer systems: store-and-forward, virtual cut-through and wormhole routing. With store-and-forward routing each packet is buffered in the intermediate nodes before it is passed to the destination node. On the other hand, with wormhole routing message flits are passed to the destination node as soon as they arrive at a node, and packets are blocked in place when required resources are unavailable [20]. Virtual cut-through is a method that is a combination of store-and-forward and wormhole routing. When resources are available, it passes message through the intermediate node immediately. When the head of a message is blocked, the packet is buffered in the intermediate node, partially freeing network resources. The message advances through the network once resources become available [21].

Virtual cut-through and store-and-forward are expensive: they require sufficient buffer space in each node to store a packet. Wormhole routing does not require buffer space to hold an entire packet, hence it is a cost effective alternative to virtual cut-through with comparable performance.

Figure 7 shows the latency of a message with store-and-forward routing and wormhole routing for a packet sent from node N_0 to N_2 via node N_1 [15]. With store-and-forward routing the message is entirely transmitted from node N_0 to node N_1, then from N_1 to N_2. On the other hand, with wormhole routing a flit is forwarded to the next node as soon as it arrives at the intermediate node.

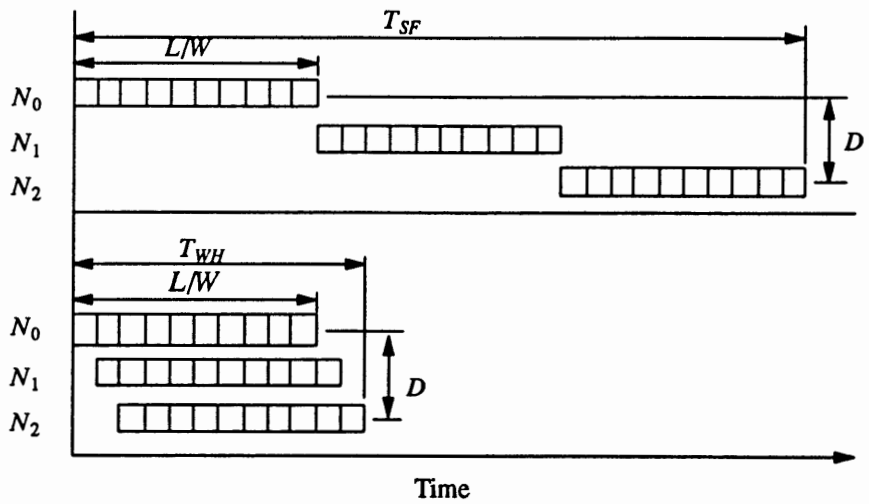


Figure 7. Latency of store-and-forward routing (top) versus wormhole routing (bottom) [15].

With store-and-forward routing, the latency is given by:

$$T_{SF} = T_C \left(\frac{L}{W} \times D \right)$$

and with wormhole routing the latency is given by:

$$T_{WH} = T_C \left(\frac{L}{W} + D \right)$$

where:

T_C - channel transmission time,

L - message length in bits,

W - channel width in bits,

D - number of channels the message must traverse (distance) [18].

Most existing concurrent computers use store-and-forward routing. An example of the computer that uses wormhole routing is the Intel Paragon. Wormhole routing is an efficient mechanism to improve network performance.

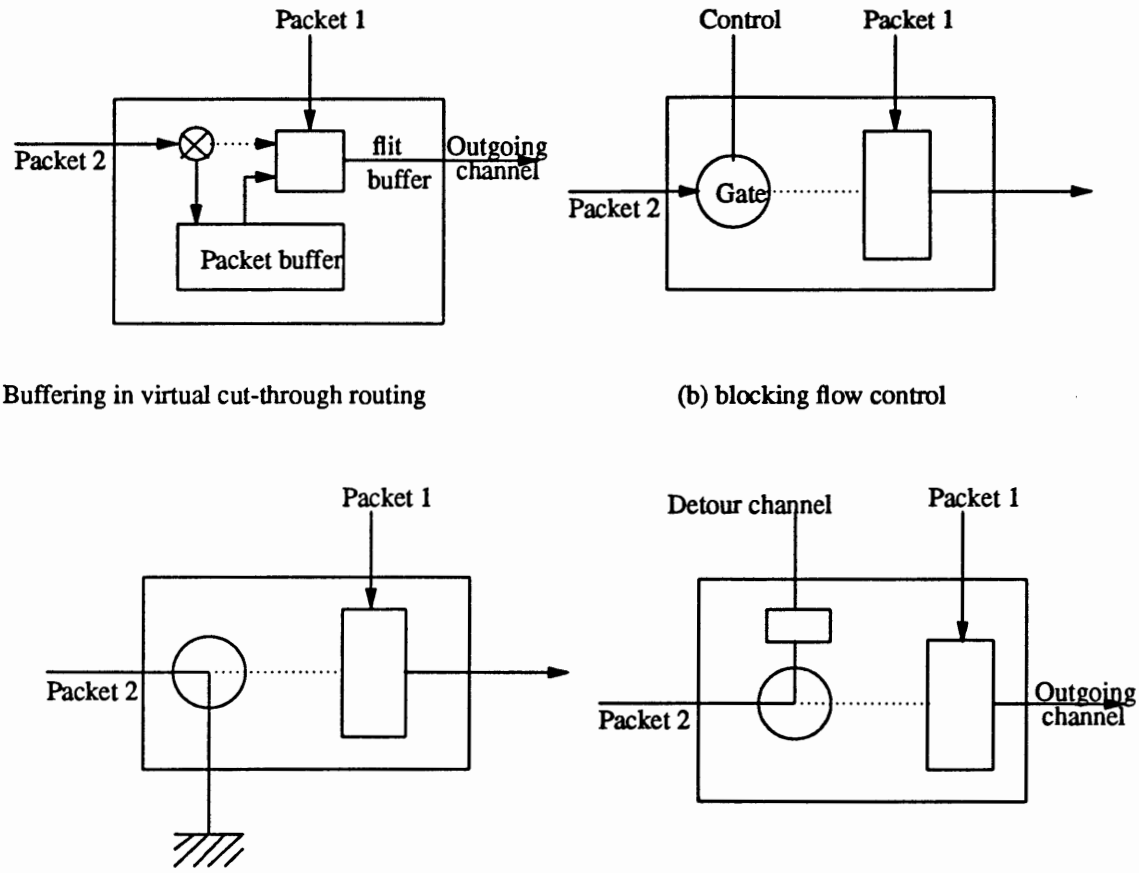
Flow Control.

The flow control protocol of a network determines how resources (buffers and channel bandwidth) are allocated and how packet collisions over resources are resolved [20]. The flow control strategy allocates router buffers and channel bandwidth to flits. The allocation must be done for an entire packet, since it is the smallest unit of information containing the routing information.

Figure 8 shows four ways to resolve a collision between two packets competing over a single channel. Only one message can be allocated to the outgoing channel at a time. The other message may be:

- blocked and buffered in an intermediate node (virtual cut-through, Figure 8a),
- blocked in place (wormhole routing, Figure 8b),
- discarded (retransmission of message is needed, Figure 8c),
- detoured (adaptive routing, Figure 8d).

The messages can take either deterministic routes or adaptive routes. In a deterministic method, the routing path does not depend on network condition, but is completely determined by the source and destination addresses. In adaptive routing, the path may depend on network condition and messages may be misrouted (sent through longer route to avoid the collision and its consequences) to avoid congested regions of the network. For every message to successfully reach the destination a deadlock-free algorithm is required. (A deadlock of a network is a condition when no messages can advance toward its destination because the queues of the message system are full [22].)



(a) Buffering in virtual cut-through routing

(b) blocking flow control

(c) Discard and retransmission

(d) Detour after being blocked

Figure 8. Flow control methods for resolving a collision between two packets requesting the same outgoing channel (packet 1 being allocated the channel and packet 2 being denied) [1].

The most popular deadlock-free deterministic routing algorithm is dimension-order routing, which is used in most existing k-ary n-cube message passing machines [22]. In this method, the message is routed in decreasing order of dimension. Example adaptive routing schemes are planar-adaptive routing [23] and chaotic routing [24].

A mechanism that improves the network performance when collisions are resolved by message blocking is virtual channel flow control [20]. A virtual channel is a logical link between two nodes. It is formed by a buffer in the source node, a physical link between source and destination, and a buffer in the receiving node [1]. Instead of allocating a single deep buffer

for storing blocked messages in the router, several smaller buffers are provided for each physical channel in the network. A physical channel is time-shared by all the virtual channels. Using virtual channels, blocked packets may be passed by other packets, going in different directions. This way the throughput of the network is increased.

Router Design Examples.

The Torus Routing Chip (TRC) [2] performs deadlock-free, cut-through routing in k-ary n-cube interconnection networks, using virtual channels. Each TRC routes packets in two dimensions. Torus Routing Chips are cascadable to construct networks of dimension greater than two [2]. A flit in the TRC is a byte. The TRC has two unidirectional channels (X and Y), each consisting of 8 data lines and 4 control lines. The TRC routes packets using dimension-order routing first in the X direction, then in the Y direction. Each channel is associated with two buffers (one per each virtual channel). One buffer is dedicated to communication with the local node. A 5x5 crossbar switch is used to establish the connection between input and output channels. The Torus Routing Chip has been used in the design of J-Machine Network. The J-Machine is an experimental supercomputer used for research at MIT [18].

The Mesh Routing Chip (MRC) [25] uses dimension-order, wormhole routing. The router has 5 input and 5 output ports (Figure 9). Routers are connected through a pair of unidirectional channels. One pair is dedicated to connection between a node and a router. Each output channel has a buffer associated with it. A 5x5 crossbar switch is used to establish a connection between input and output channels [25]. The Mesh Routing Chip has been used in network design for the Intel Paragon commercial supercomputer.

The background provided in this chapter was intended to introduce the reader to the concepts used in our Performance Evaluation Environment for Multicomputer Interconnection Networks presented in the following chapter.

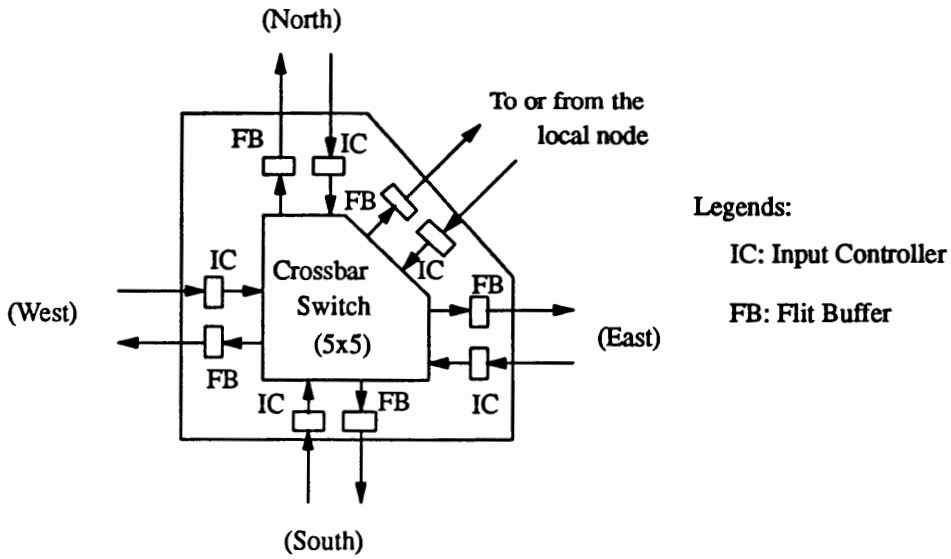


Figure 9. The structure of a Mesh Routing Chip [1].

CHAPTER IV

PARSIM COMMON ENVIRONMENT

Methodology.

The methodology that we propose for designing performance optimized computer systems is based on software tools for performance monitoring and evaluation of parallel programs. We use a software environment to evaluate performance of hardware designs. A heart of such a system is a simulator modeling a hardware design under development. A simulator executes assembly language instructions, and provides access to performance monitors from user's code. Upon detection of events specified by the user appropriate data traces are collected during the simulation run. After the simulation data records are reduced and analyzed using performance analysis module. The analyzed data can then be visualized with a set of graphical displays.

Key features of our methodology are "plug and play" simulation and modeling hardware/software interaction during the process of hardware design. A performance evaluation system can be used to solve many design problems with different simulators, as long as the format of collected data remains the same. A preprocessor must be customized to support language semantics required by a chosen simulator, or a compiler has to be provided. The simulators that can be used in the performance evaluation system must provide the ability to execute code. The ability to use different simulators with the performance evaluation system gives the user flexibility to configure the system for required functionality, accuracy and simulation performance. User code serves as a base for evaluation of hardware. Program performance gives a designer information about the efficiency of modeled hardware. Hardware behavior can be monitored with respect to different types of application programs.

Investigation of interaction between hardware and software on a hardware design stage is not a common approach. Most existing hardware models interface with user on a level of bus transactions, specifying what kind of access is to take place (data/code read/write). Data traces of such kind are hard to obtain from real applications. Therefore, observation of the impact of real software applications on a hardware design is impossible. The simulation approach that we use in our methodology supports program execution on a higher, instruction level instead of the level of bus transactions in a computer system. This kind of simulation approach for performance evaluation is an overlooked area relative to literature.

We have created a prototype performance measurement environment for designing multicomputer interconnection networks. The system, called the Parsim Common Environment (PCE), consists of an instrumented network simulator, a performance instrumentation system for application programs, and performance analysis and visualization modules. A block diagram of PCE is shown in Figure 10. All parts of the system are integrated into a graphical user interface.

To conduct a performance experiment the user first needs to prepare an application program to be executed by the simulator. The program must be annotated with the performance measurement points for specific events to be monitored during the simulation. The preprocessor converts the annotations into performance monitoring code. Then the compiler translates the program into assembly language code. The simulator executes a program and outputs data into a file. Collected data traces are then analyzed with respect to the monitored events and specified functions are performed on the data traces. Analyzed data are presented using visual displays.

Parsim simulator is intended to model small and medium size interconnection networks (up to a thousand nodes). Simulations of larger networks is very time consuming. To predict performance of large networks (massively parallel processors - thousands of nodes) one would need to replace Parsim with a less detailed simulator that would require less computation time.

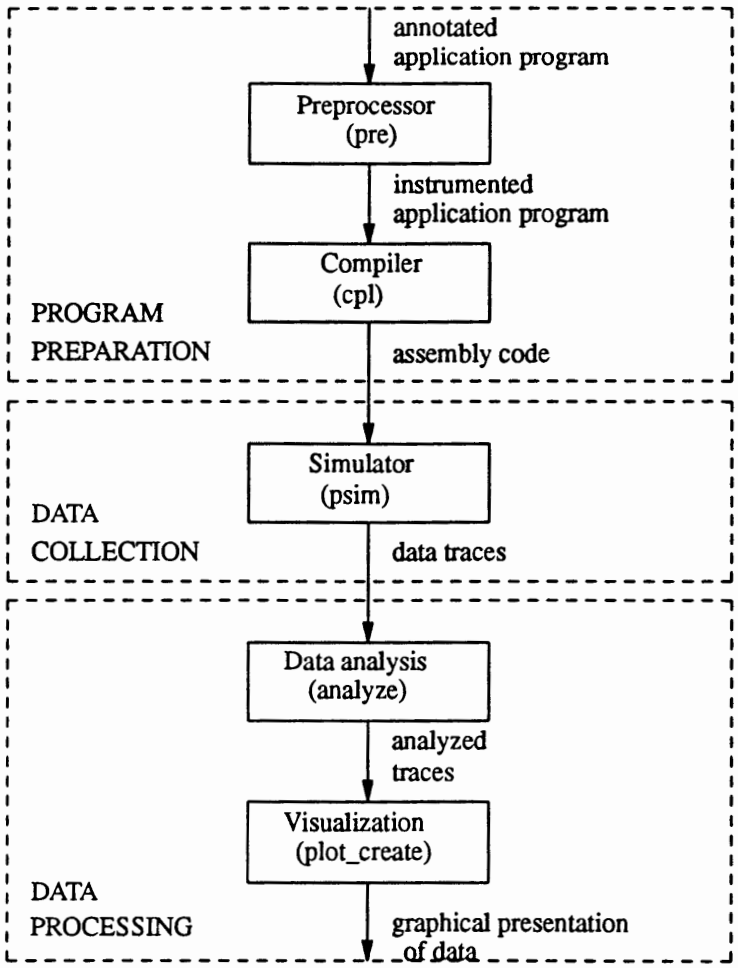


Figure 10. Block diagram of the Parsim Common Environment.

One would also need to be careful when selecting performance data for collection. Since large simulations tend to produce huge volumes of data it is necessary to reduce trace records to a representative set beforehand, so that the storage capacity is not exceeded during the simulation run.

History

PCE is the result of over two years of work by a number of people at Portland State University and is still in progress.

The simulator itself was designed and written by B.J. Porcella using the C programming language and an object-oriented approach. At this point the author of this manuscript joined the project. The initial tests and validation experiments were carried out by Anna Kolinska and Pradeep Rhagavendra. Simple test programs and later on more sophisticated application programs stressing the network communication load have been written in assembly language. The amount of work needed to develop these programs has led to the specification of a high level structured language supporting message passing paradigm, and the development of a compiler. The compiler was designed and written by B.J. Porcella using the C programming language. It made the development of application programs relatively easy.

Subsequently we decided to create a performance monitoring and visualization environment for our simulator. Anna Kolinska designed and implemented the preprocessor instrumenting application programs. She wrote it in the C++ programming language using object-oriented approach. Anna also developed the prototype command line performance analysis and visualization system. This system was initially written as a set of C-shell, awk and sed scripts. The visualization of analyzed data was implemented using the gnuplot utility. In the current performance analysis modules functions written in awk language have been replaced by code written in the C++ programming language.

Anna Kolinska also designed the graphical user interface and supervised Doug Huang, who implemented it using Tcl and Tk programming system for developing windowing applications [26]. Anna supervised work on the PCE interface and tested its functionality, structure and consistency on each stage.

The Parsim Common Environment is still under development. Recently we decided to add new features to the simulator. Work is being done to support torus connections and virtual channels. Preprocessor implementation has been finalized, we do not predict any changes in the near future. In the data analysis module we plan to add more statistical functions to evaluate

collected data traces.

The graphical user interface is the area of intensive work for last several months, and is still in the early stages of development. Basic functionality is available now, but there still remains a lot of work to do to make the interface flexible. We plan to provide a separate analysis menu to access data traces from previous simulations and perform their analysis at a later time. Currently data analysis is tied to the simulation so that collected data have to be immediately analyzed. Recently, we have incorporated a plot window into our environment. It supports a very simple set of display features now, and more functionality will be added in the near future. Using the plot window we can generate two-dimensional displays. In the future we plan to add more sophisticated display methods such as 3-dimensional graphs and performance meters.

PARSIM.

PARSIM is a network simulator for message passing architectures. It has been developed at Portland State University by Porcella et al [27]. It supports the k-ary n-cube family of network topologies. A single node in the mesh network simulated by Parsim is presented in Figure 11. The current router design is based on the Mesh Routing Chip. The network uses the packet-switched transport mechanism. The routing is deterministic: dimension-order, wormhole routing. There are two unidirectional channels in each direction (network channels). There is a buffer associated with each input channel. The buffer is a FIFO queue (first in first out). PE (processor) and a router are connected by a specified number of channels (so called PE channels).

The user may set:

- router delay - a delay associated with network channel buffers;
- PE channel delay - a delay associated with the input buffer from PE to the router;

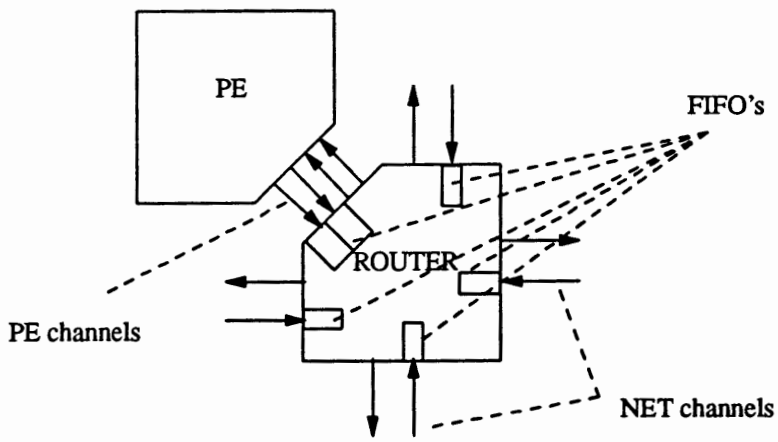


Figure 11. A node in Parsim simulator, for mesh configuration.

- operating system overhead - time that processor needs to issue a message;
- number of channels between a PE and a router;
- size of FIFO buffers;
- message length;
- network configuration.

Input to the simulator is the application program in the PMS format [27] (a low level assembly language supporting the message passing paradigm). In the application program the user configures the network using predefined configuration strings. To relieve the user from the difficulties of programming in a low level language, a structured language has also been defined. The compiler translates files from the structured language into assembly language. The simulator supports a SPMD model of programming: Single Program over Multiple Data streams. In this model each node executes the same program, but with different data sets. A node may be excluded from executing selected parts of the program by conditional statements. For program development a simple debugger is provided.

The simulator contains several monitoring features. At any given time in the program user may access registers keeping performance data.

Such registers are:

- timer - keeps time in each node;
- rec_lat - keeps the latency of the most recently received message at each node;
- nbr_nodes - keeps a number of nodes configured in the network;
- my_node - keeps the identification number of a given node.

Preprocessor.

The preprocessor plays a significant role in Parsim Common Environment. It is capable of inserting performance monitoring code into application programs. Since we estimate the performance of a given network configuration by measuring performance of application programs, preprocessor capabilities need to include collecting performance data for specified regions of code. Such data might be the time needed to execute a given instruction or subroutine, the number of times the subroutine has been called, or the value of specified variable at any given time.

To support such monitoring capabilities we have defined six event primitives that may be used for program instrumentation. The user annotates a program with flags indicating the beginning and end of a given event, and the type of information to be collected. The flags are used to mark a region of code in which performance monitoring will take place. We will refer to the part of code marked by the flags as the "monitored code".

The six available event primitives are [28, 29]:

1. duration - time that is needed to execute monitored code,
2. difference - difference between values of a specified register (variable) at the end and at the beginning of the monitored code,
3. accumulator - the value of a specified register is added to the accumulator each time an instruction changing value of that register is executed within the monitored code,

4. counter - count of the number of times the monitored code is executed, or count of the number of times a specified instruction has been executed, or count of the number of times a specified register has changed its value in the monitored code,
5. value - the value of a given register as a function of time,
6. user defined - actions to be taken upon detection of a given event are defined by the user in the form of a macro. A monitored event may be the execution of a given instruction or the change of a value of a specified register.

The user needs to indicate the start and end of a chosen event in the application program. The preprocessor copies code from the input to the output file inserting monitoring code in the specified parts of the program. The simulator treats annotations as comments if the preprocessor is not executed on the annotated program before starting simulations. Generally monitoring flags are of the following format:

!c <EVENT_TYPE> <START/STOP> <EVENT_NAME> <REGISTER> comment_field

where:

!c - indicates a monitoring flag.

EVENT_TYPE - is a type of event to be monitored, and may be one of the following: DU, DI, AC, CO, VA, MA indicating duration, difference, accumulator, counter, value, or user defined, respectively.

START/STOP - START indicates start of the monitored event, STOP indicates end of the monitored event.

EVENT_NAME - name of the monitored event: a string defined by the user.

REGISTER - register value to be collected during the event.

comment_field - user comments (optional).

The Appendix contains example programs with inserted monitoring flags.

Trace records generated during simulation are of the following format:

<EVENT_ID> <MY_NODE> <REG_VALUE>

where:

EVENT_ID - indicates the type of event and its number (maps to the name of the monitored event) for which the data trace record has been collected;

MY_NODE - which node of the system the data trace belongs to;

REG_VALUE - value of the register that has been monitored.

With duration events, a time stamp is collected at the beginning of the event (when the start flag is detected) and at the end of the event (upon detection of the stop flag). The data analysis module is responsible for extraction of the duration of each event from the data traces (difference between the value of timer at the end and at the start of event).

For difference events a value of a register specified in the REGISTER field is collected every time the program approaches start or stop flag. When timer is specified as a register value to be collected, duration and difference types of event are equivalent. The difference between the values is obtained during the analysis of data traces. Any register value may be collected.

With accumulator events, the user may keep track of the incremental changes of a specified register in the specified code block (marked by the start and stop flags of AC event). Each time the register value changes in the specified region of code it is added to the accumulator. The final value of the accumulator is output at the end of the program. If timer is specified as the register to be accumulated, its value is simply moved to the accumulator, instead of incrementing it. This is because timer register is incremental in nature. One data trace is output from each node at the end of program for accumulator type of event.

Counter event allows the user to keep track of the number of times a specified event has occurred in the region of code marked by the start and stop flags. The events counted may be:

- number of times a specified register has changed its value (if the REGISTER field contains a valid register),
- number of times a specified instruction has been executed (if REGISTER field contains a valid instruction),
- number of times a specified region of code has been executed (if REGISTER field is empty).

The final value of the counter is output at the end of the program.

With user defined events we need to provide descriptions of actions to be taken upon detection of a specific event during program execution. These actions are defined in a macro incorporated at the end of the application program. Macros give a flexible way to create sophisticated performance measures. Detailed information about user defined events is available in [29].

There are two default events available via a command line option from the preprocessor. These are:

- default execution time - execution time of a given program;
- default latency - average message latency in a program.

Data traces for these events are output to the file once in each node at the end of a program. When used, monitoring flags are automatically inserted into the program, and then replaced by corresponding monitoring code.

Creating and Instrumenting Application Programs

An application program has to go through several preparation steps before it can be executed by the simulator as shown in Figure 12. The user is responsible for writing a program and annotating it with performance monitoring flags. The preprocessor inserts monitoring code into the program. Then the program is compiled into the assembly code acceptable by the simulator.

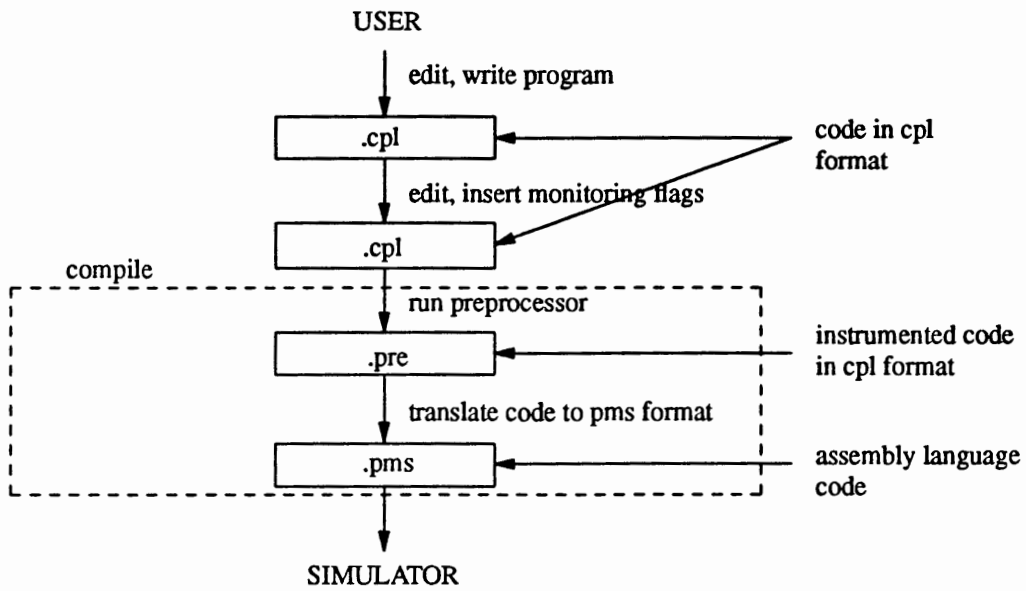


Figure 12. Program preparation for Parsim.

Simulation Environment

The simulation environment is described by the setup file *<program_file>.alter.<setup_file>*. The file contains parameters and the configuration of the network. One of the parameters may be varied to indicate that a set of simulations is to be performed. Data analysis may be made with respect to the varied parameter. If none of the parameters is varied, a single simulation is performed.

Data Analysis.

Data analysis modules extract the data traces for a specified event from a data file created during simulation. The user chooses a function to apply to traces characterizing a given event. The function is applied to the data and the result is stored in the form required by the visualization module. Each event type has a set of functions associated with it. The chosen function and the type of simulation experiment (single simulation or a set of simulations with a varied parameter) determine the number of data files created.

Supported functions are:

- f_{max}, f_{avg}, f_{min}, $f_{accumulated}$:

  The data for a given event are evaluated and maximum, average, minimum or accumulated value of the traces is collected in the output file. If a single simulation is performed, the functions output just one value. If a set of simulations is performed (with varied parameter) the collected data are the function of a varied parameter of a simulation.

- $f_{max}(proc\_num)$, $f_{avg}(proc\_num)$, $f_{min}(proc\_num)$:

  For a single simulation the analyzed data are a function of a processor number in the network. The output of the analysis is collected in a single file. For a set of simulations a set of output files is created, one file for each instance of a varied parameter of simulation. Data collected in each file are a function of a processor number in the network.

- $f_{max}(ev\_num)$, $f_{avg}(ev\_num)$, $f_{min}(ev\_num)$:

  The argument of these functions is the occurrence number of a specified event. For example, if the program executes a given part of program in a loop, we can easily obtain the data specific to each loop iteration by using the above functions (with the assumption that we set a monitoring probe inside a loop). If we perform a set of simulations with a varied parameter, a set of data files is generated. For a single simulation experiment we obtain one output file.

- *value*(*time*)

  The function is available for events of type value and for user defined events. The data gathered in the output file are just the recorded values of a given register as a function of simulation time. A separate data file is created for every processor in the network. For a single simulation experiment we obtain as many files as there are nodes in the network. If a set of simulations with varied parameter is performed, we obtain a set of output files for every variable of simulation.

Detailed description of data analysis stage is provided in [30].

Visualization of Performance Data.

Data traces obtained from the analysis stage are gathered in a directory (./PLOT), which is a subdirectory of the simulation directory (the one in which the application program is stored). The experiments that generated the files are listed in the file: ./PLOT/PLOT_INFO. The file name and the PLOT_INFO file give the user all the information about the contents of the other files. We do not force any specific display configuration for a given event or function chosen. Instead, the user is given the complete flexibility to configure displays generated during simulation and analysis deciding what files or file sets are shown together in one plot window. The user selects data files to be displayed from a file browser of graphical user interface.

Graphical User Interface.

All parts of the Parsim Common Environment are integrated in the graphical user interface. The main window of PCE is shown in Figure 13.

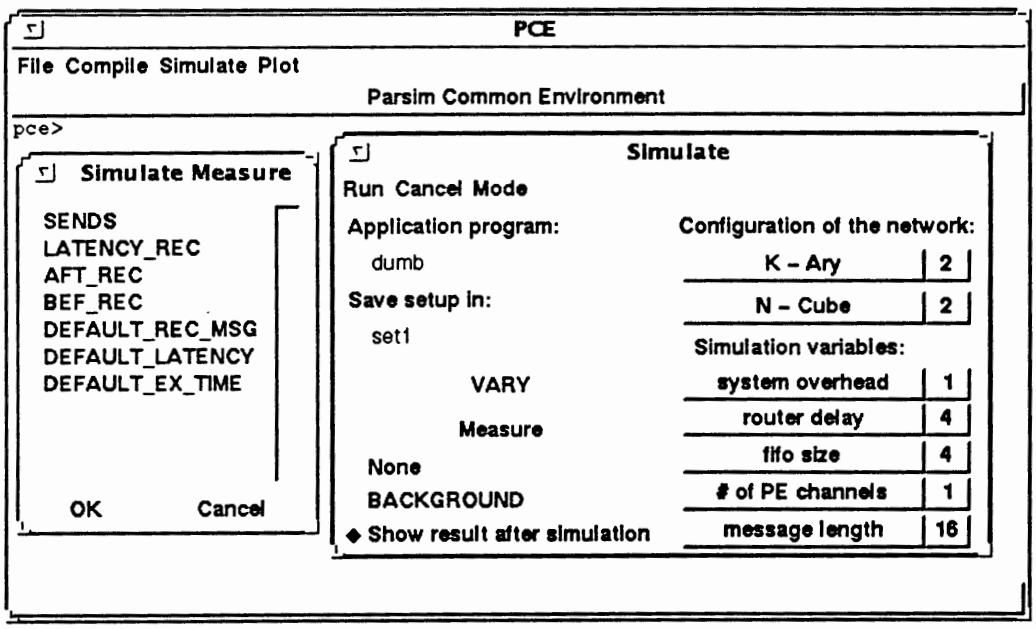


Figure 13. Graphical User Interface for PCE.

In PCE an editor is available for code development and instrumentation. The user can invoke the editor from the *File* menu option of PCE (Figure 13). The *Compile* option performs insertion of monitoring code and translation to the simulator input format. However, the user does not need to separately compile programs before running the simulator, because compilation is always performed at the start of the simulation. The compile option is available for debugging purposes, and is useful for code instrumentation. After compilation the user may view an instrumented program (with the extension .pre) to check its correctness.

The *Simulate* window (opened in Figure 13) allows the user to specify the topology and design parameters of the network. The user needs to set the design variables of the network and its configuration in the *Simulate* window. The variables set in the interface are stored in the setup file **alter**, that is read by the scripts performing the simulations. Values given in the application file are overwritten by the values specified in the user interface. A *VARY* option gives the flexibility to specify a set of experiments to be performed. A *BACKGROUND* option is provided for simulations of large networks, which are usually very time consuming.

The user may specify which event traces to analyze from the *Measure* menu in the simulation window (*Simulate Measure* window in Figure 13). Analysis will be performed right after the simulation run finishes. The event browser from the *Simulate Measure* window shows events that can be analyzed. The names directly correspond to the event names given by the user to monitoring flags in the application program.

The data files that are obtained after finishing the analysis stage are ready for visualization. Menu option *PLOT* opens a file browser, from which user can select files to display. An example *PLOT* window with simple results of simulation is shown in Figure 14. Our system provides two-dimensional displays of performance data. The graphical window can be written into a specified postscript file.

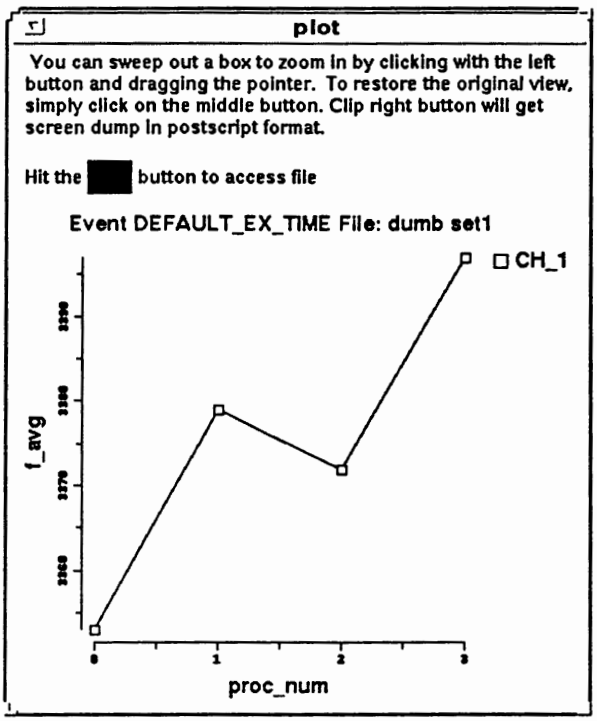


Figure 14. Plotting results with PCE.

Data Storage — Directory Structure

Every simulation experiment or set of experiments generates a directory structure in which data traces are kept. An example directory structure is shown in Figure 15. The names of files the user specifies in the simulation window are used to generate the name of the directory. If a single experiment is performed, and the application file name is *dumb*, and the name of a configured setup for simulation is called *set1*, then the directory containing data traces for this simulation will be called: *dumb.ONE.set1*. Similarly, when *VARY* option is chosen, the directory will be named: *dumb.VAR.set1*. We will refer to these directories as the trace directories.

On the same level as the trace directory, there is an instrumented application file, *dumb.pre*, a list of events monitored during simulation, *dumb.event_info*, a configuration setup file generated by the simulation window, *dumb.alter.set1*, and an assembly language input file to the simulator, *dumb.pms*.

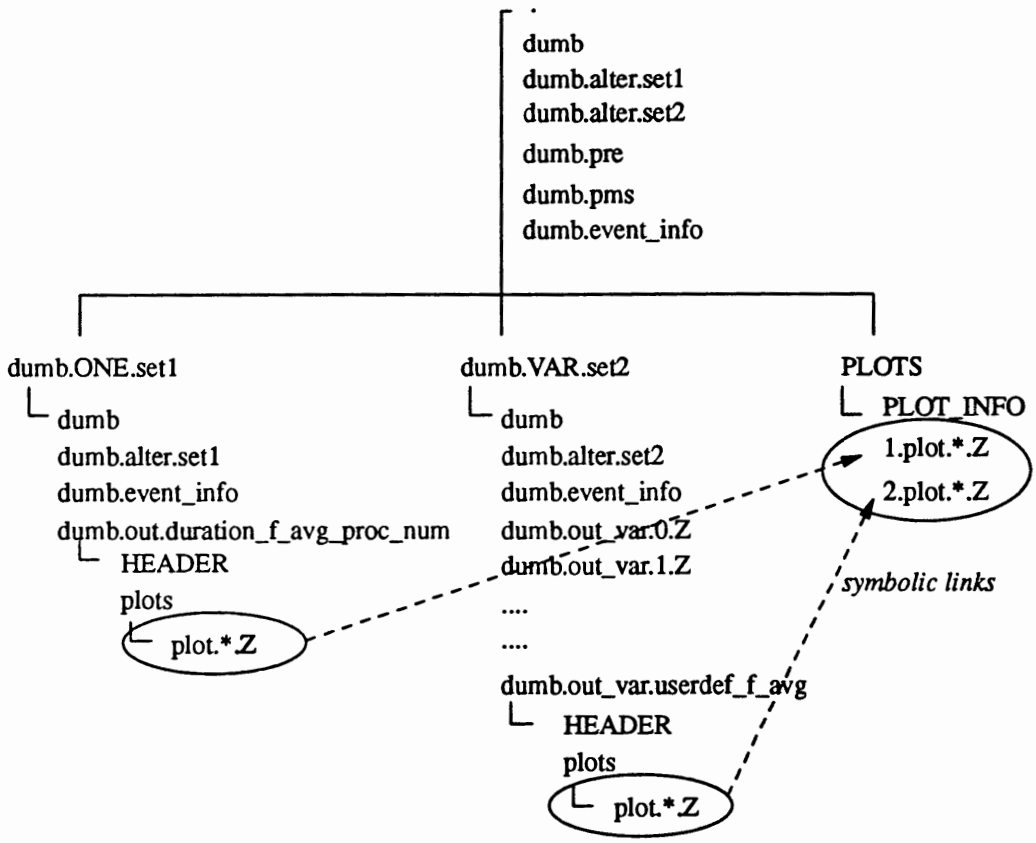


Figure 15. Directory structure generated by PCE.

Underneath the trace directory there are data traces collected during simulation and subdirectories for each event analysis. Each subdirectory contains a file *HEADER* with information about the data analysis performed, and a *plot* subdirectory containing analyzed data traces. All data traces are stored in compressed form to save disk space. Since the data files prepared for visualization are not easily accessible (are stored deep in the directory structure), we provide an additional subdirectory *PLOT* underneath the trace directory. The *PLOT* directory contains links to all the analyzed data traces. It also contains the file *PLOT_INFO* that describes the experiments in which data files have been generated. The directory *PLOT* is accessed from the PCE.

COMMENTS ON THE STRUCTURE OF PERFORMANCE EVALUATION TOOLS

PCE is a prototype performance evaluation tool for hardware design. Although it provides only basic functionality (visualization displays are still in the development stage), it has already helped us solve a number of problems. An example problem for network design is discussed in Chapter V.

A generic performance evaluation tool for designing hardware consists of a part that is strictly dependent on the design and a part that is independent. Simulator and performance instrumentation depend on a design, and need to be developed for a hardware design of interest. Therefore, our simulator may be replaced by any simulator of specific hardware. However, such a simulator must provide performance data of interest. We solved the problem of accessibility of performance data by providing additional registers to keep the required information. Application programs need to be instrumented with performance monitoring code. Depending on the language semantics the measurement flags need to be translated to a specific format. Also events of interest may differ depending on the design problem. To provide flexibility of defining the actions to be taken upon detection of a given event, we provided a user defined event option in our instrumentation tool.

Data analysis and visualization are usually independent on the design problem. To provide data analysis and visualization tools we may either develop our own tools or use tools that are already available. The last option sounds very promising, since it does not involve any programming overhead. However, to be able to use existing tools, we must provide data traces that will have the format acceptable by a chosen tool. Hence a simulator must be able to generate such a data format. The user may also decide to provide data analysis tools and reuse only visualization tools. This approach still gives the flexibility of having various display options available, while performing required analysis, that may not be available in existing tools. Figure 16 shows the parts of performance evaluation environment, dividing it to the design dependent and

design independent part.

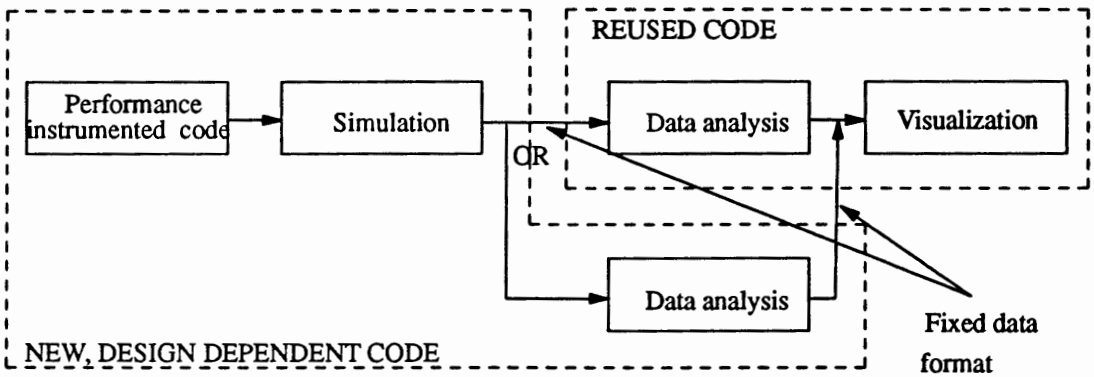


Figure 16. Performance evaluation environment, opportunities to reuse existing code.

The problem that may discourage a software engineer from developing performance evaluation environment, may be the fact that there exists no standard format for data traces for the tools that are available [31]. Hence, once the visualization environment is chosen, it cannot be replaced by any other tool without modifying all other components of the system. A thorough examination of software packages needs to be done before selecting a specific tool.

We have developed all the parts of performance evaluation environment primarily to have a uniform tool for development of interconnection networks. Also, we underestimated the amount of work involved in development of visualization tools. That is why for future work we plan to develop a system that could use sophisticated visualization methods, reusing already existing software packages.

CHAPTER V

NETWORK DESIGN EXAMPLE

Introduction.

In a popular model of an interconnection network, each routing element contains a set of channels for communication between nodes, as well as for processor-router communication within a node. Usually, channels are unidirectional: there is one channel ingoing to the router, and one outgoing from the router in each direction.

There are two sources of delay in the network:

- router delay - a message is blocked in the router waiting for the switching element to establish appropriate connection between input and output channels;
- network contention - a message is blocked in the router due to a message transfer on the output channel.

Each input channel to the router has a FIFO buffer associated with it. When blocked, a message is fully or partially stored in the FIFO buffer depending on the message and buffer sizes.

We refer to the channels between a processing element and a router within a node as PE channels. The channels between neighboring nodes are called net channels. The process of sending messages by the PEs is also referred to as generating messages. The simulation model of a multicomputer node is presented in Figure 17.

When a processing element generates a message, it proceeds through the PE channel and is temporarily stored in the PE buffer waiting for the connection with the output channel. A processor cannot generate a new message until the previous message leaves the PE buffer and proceeds towards its destination.

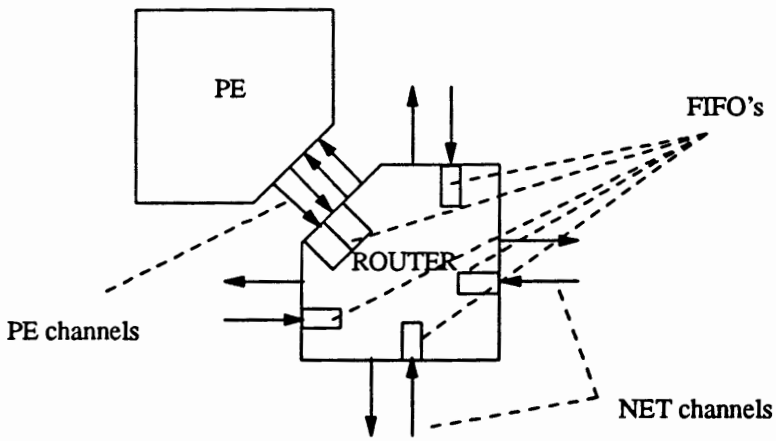


Figure 17. The model of a multicomputer node.

Hence, it seems reasonable to provide multiple PE channels to better utilize the interconnection network. While one message is blocked in the PE channel because the output channel in dimension X is busy, for example, another message can be generated and successfully proceed in dimension Y. This way messages would be able to reach their destinations earlier, and the overall execution time of a program would be shortened, thus improving system performance. Having as many PE channels connecting the router and processor as there are dimensions in the network makes intuitive sense.

In the following example we will examine the behavior of a mesh-connected network for different numbers of PE channels. We will observe the message traffic, trying to find the number of PE channels that gives the best performance of our application programs.

We have selected application programs that generate specific message traffic in the network. First we examine performance for a simple low contention case. Then we stress the network with a heavy message traffic, examining performance for high contention conditions.

Application Programs.

We have investigated the problem with two sets of application programs:

1. A simple program, in which one node sends a message to several destinations (multicast). In this example the network is almost empty and there is no contention when the network with one PE channel is used. For multiple PE channels low contention is created in the network. This shows the performance of the network under low traffic conditions for different numbers of PE channels.
2. An application program solving the N-body problem. In this example a lot of traffic is generated in the network in a short period of time, causing significant contention. This shows the performance of the network for heavy traffic, and contention for different numbers of PE channels.

Network Configuration.

Experiments have been performed for different sizes of the interconnection network configured as a mesh. Parameters of the network were set as follows:

system overhead = 1 (time the processor needs to generate a message);

FIFO buffer = 4 (size of FIFO buffers);

router delay = 4 (time needed for the router to establish connection between input and output channels);

message length = 16

We have chosen a very small system overhead time to expose the time that messages spend in the network. Messages are short, as in the fine-grain programming model. All buffers in the routers are of the same size to reduce router cost. Router delay is set to the minimum time required for the chosen size of FIFO buffers. A FIFO can keep flits of only one message at any

given time and one unit of time is needed to advance a message flit by one position in the FIFO.

NETWORK PERFORMANCE WITH LOW CONTENTION

For this set of experiments we analyze a 16 node mesh network (4x4) as shown in Figure 18, with varying number of PE channels.

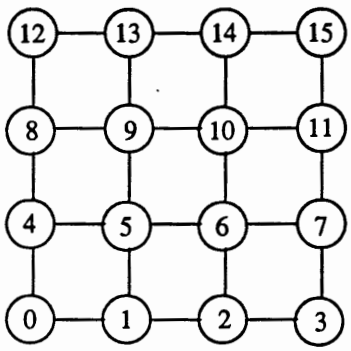


Figure 18. 16 node Mesh configuration (4x4).

In the experiments, specified nodes send several messages, and the destination nodes receive them. We analyze the timing of each message by showing the times when the head of a message and the tail of a message reach certain points in their routes toward the destination. The word *message* refers to a single packet.

The receive time of a message and message latency are two basic performance metrics that we extract from data traces collected during the simulation of the application programs. The receive time of a message is defined as the time at which the tail flit of the message reaches its destination. Message latency is defined as the time that it takes for a message to get from its source to its destination. In other words, it is the time calculated from the moment when the first flit of a message is generated in the source PE until the last flit is consumed by the destination PE. Both performance measures depend on several factors:

- message length,

- distance between source and destination,
- delay of router buffers (PE and network),
- operating system overhead,
- network contention.

Network contention is a significant factor in the timing of the messages. If there is no contention in the network, the following equation is valid [32]:

$$latency = os\_overhead + number\_of\_FIFO's\_to\_wait \times router\_delay + message\_length$$

$$number\_of\_FIFO's\_to\_wait = distance + 1$$

Table I shows the analytical calculation of message latency for specified distances between the source and the destination.

TABLE I

ANALYTICAL CALCULATION OF MESSAGE LATENCY
FOR NO CONTENTION NETWORK

Distance	Router Delay	OS Overhead	Message Length	Latency
1	4	1	16	25
2	4	1	16	29
3	4	1	16	33

Message pattern 1 presented below shows that the experimental results match the expected analytical results (Table II).

Sending Messages.

Message Pattern 1

Consider the detailed timing for messages generated by node 0 to nodes 1, 2 and 3 as shown in Table II. In Table II the following symbols have been used:

MSG*i* - message sent to node *i*,

H - head flit of a message,

T - tail flit of a message,

I - time when the message was generated,

Rec. time - receive time of a message.

Node i - time spent in the buffer of node i, caused by router delay (time spent in the FIFO because of blocking is not indicated),

Ch. $i - j$ - time when the network channel between node i and j was busy because of message transfer.

TABLE II

TIMING FOR 3 MESSAGE MULTICAST FOR 16 NODE MESH NETWORK WITH 1 PE CHANNEL.

		I	Node 0	Ch. 0-1	Node 1	Ch. 1-2	Node 2	Ch. 2-3	Node 3	Latency	Rec. time
MSG1	H T	0	1-5 17-21	5-25	5-9 21-25					25	25
MSG2	H T	22	23-27 39-43	27-47	27-31 43-47	31-51	31-35 47-51			29	51
MSG3	H T	44	45-49 61-65	49-69	49-53 65-69	53-73	53-57 69-73	57-77	57-61 73-77	33	77

The message is generated by the source PE at the time indicated in column *I* and is consumed by the destination node at time when the tail of a message finishes its route (after waiting in the router buffer in the destination node).

A channel between a source and a destination node is free when the input buffer of a destination node is empty. Looking at the utilization of the channel between node 0 and 1 we notice the time periods when the channel is busy (including the time needed to clear the input buffer in node 1). The timing is:

5 - 25

27 - 47

49 - 69

Therefore, from Table II we see that two ticks were actually wasted between sends of subsequent messages.

Let's now consider the impact of the number of PE channels on network performance. Table III shows the timing for messages sent through the 16 node mesh network with two PE channels.

TABLE III

TIMING FOR 3 MESSAGE MULTICAST FOR 16 NODE MESH NETWORK WITH 2 PE CHANNELS

		I	Node 0	Ch. 0-1	Node 1	Ch. 1-2	Node 2	Ch. 2-3	Node 3	Latency	Rec. time
MSG1	H T	0	1-5 17-21	5-25	5-9 21-25					25	25
MSG2	H T	0	1-5 38-42	26-46	26-30 42-46	30-50	30-34 46-50			50	50
MSG3	H T	22	23-27 59-63	47-67	47-51 63-67	51-71	51-55 67-71	55-75	55-59 71-75	53	75

Referring to the two channel example (Table III) we see that the additional channel created contention in the network, since message two was blocked in the router buffer by message one. Message two could not proceed through the channel until message one reached its destination, thus gaining additional latency. Message three however, could be generated right after message one left the input buffer, thus being ready to proceed immediately when the channel was freed. This way the operating system overhead time has been overlapped with the time the message had to wait for a free channel, while in the one PE channel model, operating system overhead is added to the total time needed to traverse the network. Hence we gained two ticks of overall receive time, due to the operating system overhead.

In our experiments we have set the operating system overhead to be very small, considering only the message latency that is due to the routing scheme. In real systems, operating system overhead is much higher than message latency [33], and the benefit that we can get from overlapping operating system overhead is very important when optimizing system performance.

In the two PE example messages generated by the processor of node 0 keep waiting in the PE buffers ready to start traversal whenever the network channel becomes idle. However, the messages experience additional delay, because they are generated much earlier than they would be in the one PE model. The wait time in the buffers adds to the overall message latency.

Message Pattern 2

Let's now change the order of sending messages through the network and examine the impact of routing algorithm on network performance. In the modified algorithm node 0 sends the messages in the reverse order, to node 3 first, then to node 2 and node 1. Analysis for one and two PE channel model is given in tables IV and V.

TABLE IV

TIMING FOR 3 MESSAGE MULTICAST FOR 16 NODE MESH NETWORK WITH 1 PE CHANNEL (REVERSE ORDER)

		I	Node 0	Ch. 0-1	Node 1	Ch. 1-2	Node 2	Ch. 2-3	Node 3	Latency	Rec. time
MSG3	H T	0	1-5 17-21	5-25	5-9 21-25	9-29	9-13 25-29	13-33	13-17 29-33	33	33
MSG2	H T	22	23-27 39-43	27-47	27-31 43-47	31-51	31-35 47-51			29	51
MSG1	H T	43	44-48 60-64	48-68	48-52 64-68					25	68

TABLE V

TIMING FOR 3 MESSAGE MULTICAST FOR 16 NODE MESH NETWORK WITH 2 PE CHANNELS (REVERSE ORDER)

		I	Node 0	Ch. 0-1	Node 1	Ch. 1-2	Node 2	Ch. 2-3	Node 3	Latency	Rec. time
MSG3	H T	0	1-5 17-21	5-25	5-9 21-25	9-29	9-13 25-29	13-33	13-17 29-33	33	33
MSG2	H T	0	1-5 38-42	26-46	26-30 42-46	30-50	30-34 46-50			50	50
MSG1	H T	22	23-27 59-63	47-67	47-51 63-67					45	67

As in Message Pattern 1 the receive times of messages are similar for one and two PE channel configurations. However, comparing data for the same number of PE channels in Message Pattern 1 and 2 we obtain much lower receive times in Message Pattern 2. This is due to the modified routing algorithm:

68 ticks (Table IV), versus 77 ticks (Table II),

67 ticks (Table V), versus 75 ticks (Table III).

From tables II-V we observe that the routing algorithm can be improved by sending messages to the furthest destinations first.

Message Pattern 3

Let's now consider the case where messages are sent in different directions: to node 1 and 4. The timing analysis for one and two PE channel models is presented in Tables VI and VII.

TABLE VI

TIMING FOR 2 MESSAGE MULTICAST FOR 16 NODE MESH NETWORK WITH 1 PE CHANNEL

		I	Node 0	Ch. 0-1	Node 1	Ch. 1-4	Node 4	Latency	Rec. time
MSG1	H T	0	1-5 17-21	5-25	5-9 21-25			25	25
MSG4	H T	22	23-27 39-43			27-47	27-31 43-47	25	47

TABLE VII

TIMING FOR 2 MESSAGE MULTICAST FOR 16 NODE MESH NETWORK WITH 2 PE CHANNELS

		I	Node 0	Ch. 0-1	Node 1	Ch. 1-4	Node 4	Latency	Rec. time
MSG1	H T	0	1-5 17-21	5-25	5-9 21-25			25	25
MSG4	H T	0	1-5 17-21			5-25	5-9 21-25	25	25

From the data shown in the Tables VI and VII we notice that the latency of messages for the two

experiments was the same, but the receive time for one PE channel model was much higher.

From Tables II-VII we observe:

- With the proper message distribution, we can improve receive time by having as many as four PE channels for each router.
- Inefficient multicast algorithm causes, that we waste added resources.

We could improve multicast algorithm by:

- sending in different dimensions first,
- sending to the furthest nodes first.

To optimize program performance we should use both improvements.

From the Message Patterns 1-3 we see that with additional PE channels latency of messages was higher, but there was a good chance to improve receive time with the proper message distribution. Therefore, latency alone is not a good performance measure. With the multiple PE channels, latency does not reflect the actual contention level of the network channels. By adding PE channels to the router we cause, that messages that would not be normally generated until the channels get freed, now get generated early and wait in FIFO queue, ready to start their routes whenever channels become available.

The benefit of having the multiple PE channels is the fact that the operating system overhead occurs while the network channels are busy, and messages cannot proceed. Therefore, the operating system overhead of one message is overlapped with the time when another message uses the channel effectively. Once the traveling message leaves the channel, the message just generated can start its way through the channel without a delay.

Receiving Messages.

Message Pattern 4

To analyze network behavior from the point of view of receiving messages in the destination nodes for different number of PE channels consider the following situation: in a 16 node mesh, node 5 receives 4 messages, 1 from each direction: from node 1, 4, 6 and 9 (Figure 19). Timings for 1 and 2 PE channels are showed in tables VII and VIII.

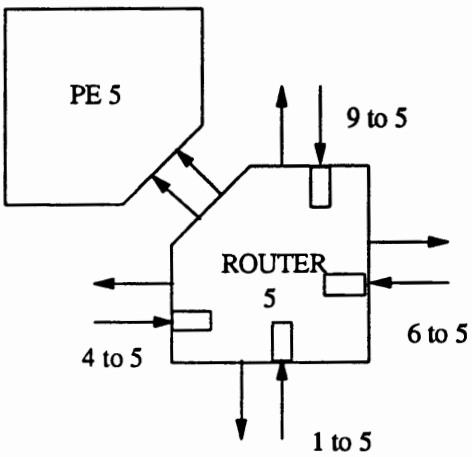


Figure 19. Message receive in 16 node mesh with 2 PE channels (node 5).

TABLE VIII

TIMING FOR RECEIVING 4 MESSAGES IN THE SAME DESTINATION FOR 16 NODE MESH NETWORK WITH 1 PE CHANNEL

		I	Source	Ch.	Node 5	Latency	Rec. time
MSG4	H T	0	1-5 17-21	5-25	5-9 21-25	25	25
MSG1	H T	0	1-5 34-38	22-42	22-26 38-42	42	42
MSG6	H T	0	1-5 51-55	39-59	39-43 55-59	59	59
MSG9	H T	0	1-5 58-62	46-66	46-50 62-76	76	76

TABLE IX

TIMING FOR RECEIVING 4 MESSAGES IN THE SAME DESTINATION FOR 16 NODE MESH NETWORK WITH 2 PE CHANNELS

		I	Source	Ch.	Node 5	PE Channel	Latency	Rec. time
MSG1	H T	0	1-5 17-21	5-25	5-9 21-25	0	25	25
MSG4	H T	0	1-5 17-21	5-25	5-9 21-25	1	25	25
MSG6	H T	0	1-5 34-38	22-42	22-26 38-42	0	42	42
MSG9	H T	0	1-5 34-38	22-42	22-26 38-42	1	42	42

From the tables we see that the latency of every message is equal to its receive time. When the number of PE channels increases, more messages can be accepted by the processor at the same time, thus improving performance. For two PE channels, two messages can be accepted by a processor at the same time, decreasing the receive time from 76 to 42 ticks. Similarly, for four PE channels node 5 would receive all the messages at time 25, with latency 25. If we have more than four PE channels in the network, only four of them can do useful work. Others just waste the resources, since there are only four network channels available. Average message latency as a function of the number of PE channels in the network is presented in Figure 20.

Summary of Observations for Low Contention.

- From the point of view of inserting messages into the network, adding PE channels causes higher latency. That happens because of the router-network bottleneck. With multiple PE channels more messages are generated in a short period of time. The messages wait in the PE buffers for appropriate network connections. Once the connections are granted messages can immediately proceed towards the destinations. Therefore, messages can leave the source nodes earlier than they would in the one PE channel network. However, they also spend

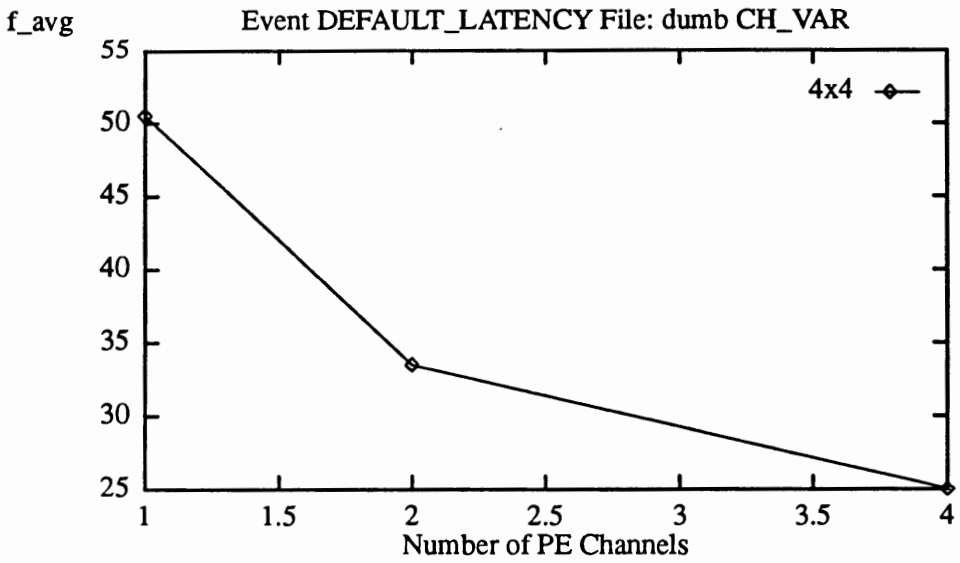


Figure 20. Average message latency for varying number of PE channels when receiving four messages.

more time in the source node PE buffers because they are generated earlier and have to compete for network channels not only with messages from neighboring nodes, but also with messages generated by the local PE. Because messages are generated earlier than they would be with the one PE channel network, they can start and finish their routes earlier, thus improving network performance. Hence, latency is not a good measure for comparing performance of networks with multiple PE channels. It only reflects the level of contention in the network. A more accurate measure of network performance is the latency calculated from the moment when the first flit of a message leaves the source node's router until the last flit is consumed by the destination PE. However, such a measure is not supported by the simulator. Therefore, we will continue to use our initial definition of message latency to observe contention for different network configurations.

- From the point of view of receiving messages by the nodes, adding PE channels shortens latency and receive time.

- The dominant factor determining network performance is the algorithm for distributing messages through the network. To optimize performance, messages should be sent in different dimensions. Messages that have the longest distance to traverse should be sent first.

NETWORK PERFORMANCE IN PRESENCE OF HIGH CONTENTION

For this set of experiments we analyze 4, 16 and 64 node mesh networks. The application program used the gravitational N-body problem [34, 35]. In this program each processor of a network is modeling one "star" in the galaxy. Each star needs to calculate its location and velocity based on the information about location of all other stars at each time step. Hence, each processor sends a broadcast message to all other processors to distribute the information about star's location in the galaxy. Then it waits for messages from all other processors. After receiving the required messages, each node computes a new position of its star in the galaxy. Finally, the result of computation is broadcast, thus repeating the cycle.

The N-body application has been chosen for performance analysis of the mesh network because of its parallel computational model. All processors are equally utilized in the network. The amount of traffic that is generated in each iteration of the program causes quick network congestion. There are 10 iterations of the broadcast-receive cycle in the program. In each iteration there are $(N-1) \times N$ messages injected into the network, where N is the number of nodes in a given network configuration.

The activities in the network are the following: in the first iteration all nodes start broadcasting messages at the same time. After the send activity is finished, each node starts receiving the messages. When all the messages of a current iteration have been received, the computation of a new star location is performed (a PMS wait statement is used for modeling the computation time). Computation time is the same for every node in the network. After the computation period, a new iteration begins with broadcasting messages again. The time that it takes to execute one iteration of a program is different for each node, hence broadcast phases after the

first are not synchronized in all the nodes. The computation time is used to remove the messages of previous iterations from the network, starting a new iteration with an almost empty network.

The communication model is based on a "dumb" broadcast algorithm, where nodes need to generate as many messages as the number of destinations to be serviced. No copying in the router takes place. The routing of messages is in dimension-order. The messages are distributed in a fixed order, first to node 0, then nodes 1, 2, ... , and N-1, where N is the size of the network. A pseudocode description of the program solving the N-body problem is shown in Figure 21.

```
configure_network();

set_wait_time();

for (i=0; i < 10; i++) {

        // Broadcast message

        msg_counter = 0;

        while (msg_counter < nbr_nodes) {
                if (msg_counter != my_node) {
                        destination = msg_counter;
                        send (type, destination, length);
                        msg_counter ++;
                }
        }

        // Receive messages

        msg_counter = 0;

        while (msg_counter < nbr_nodes) {
                receive (type, source, length);
                msg_counter ++;
        }

        wait(time);    // wait models computation
}
```

Figure 21. Pseudocode description of N-body problem.

Node Execution Time.

We define node execution time as the time needed to execute an application program by each node of the network. It is the same as the receive time of the last message received in a program by a given node. For the first set of experiments, the computation time of each node (modeled with wait statement) was scaled with the size of the network. The idea of scaling the computation time was to completely clear the network from the messages sent in the previous iterations of the program before starting new iteration. Therefore, for 2x2 mesh we set computation time to 256 ticks, for 4x4 mesh to 1024 ticks, and for 8x8 mesh to 4096 ticks. In the experiment we varied the number of PE channels in the network: 1, 2, 4, 6 and 8. The program was instrumented with default monitoring options: execution time and latency. Execution time was collected in each node, and then the average for a given network was calculated.

Figure 22 shows the average execution time of a program for varied number of PE channels.

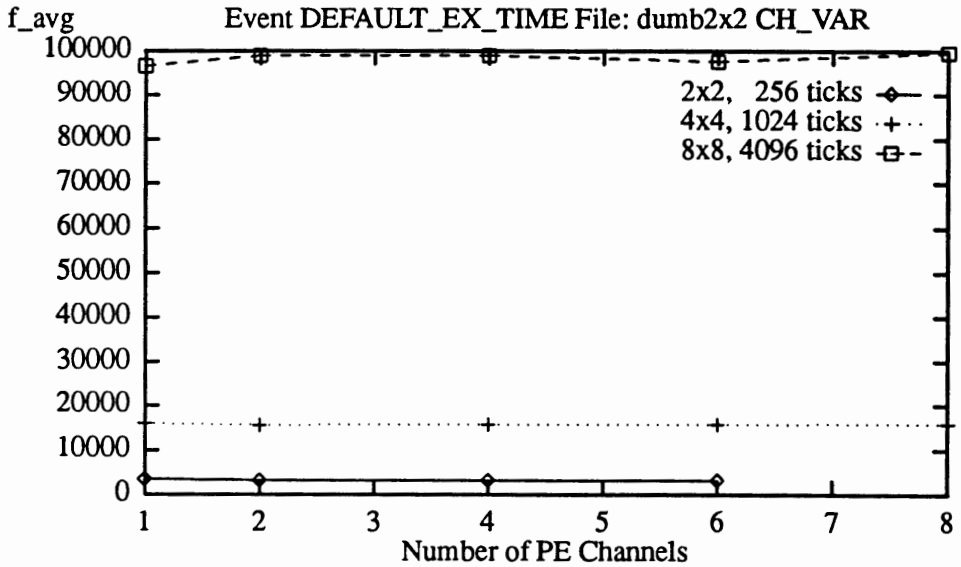


Figure 22. Average execution time for varying number of PE channels (scaled computation time).

From the figure we notice that the program executed much faster for smaller networks. There are several reasons for that:

1. Execution time depends on a number of messages generated by the program, which is given by the equation:

$$nu\_msgs = (n-1) \times n \times I$$

where:

nu_msgs - number of messages created,

n - number of nodes in the network,

I - number of iterations.

According to the above equation, for 2x2 mesh there were 120 messages generated, for 4x4 mesh - 2400 messages, and for 8x8 mesh - 40320.

2. Size of the network: it takes more time to travel longer distances.

3. Computation time: its value is set proportionally to the network size.

We will show that for fixed computation time for all network sizes, the execution time is still scaled with the network size due to 1. and 2. above. Figure 23 shows the execution time of the program for fixed computation time of 256 ticks. This control experiment has been performed to check if scaling computation time reduces contention in the network between iterations. From Figures 22 and 23 we see that the execution time obtained for the 8x8 mesh with 4096 ticks of computation time is approximately equal to the execution time obtained for the 8x8 mesh with computation time set to 256 ticks plus the additional time spent in the computation phase $((4096 - 256) \times 10)$. Therefore, we abort scaling computation time with network size in further experiments. Thus, we reduce the overhead of preparing different versions of application program for each network size for the simulations.

From Figures 22 and 23 we could reason that execution time does not depend on the number of PE channels in the network. This is not exactly the case, because the differences are not visible on a large scale. Figures 24, 25 and 26 show the curves from Figure 22 on separate graphs. From the figures we see that execution time for a given number of PEs does depend on

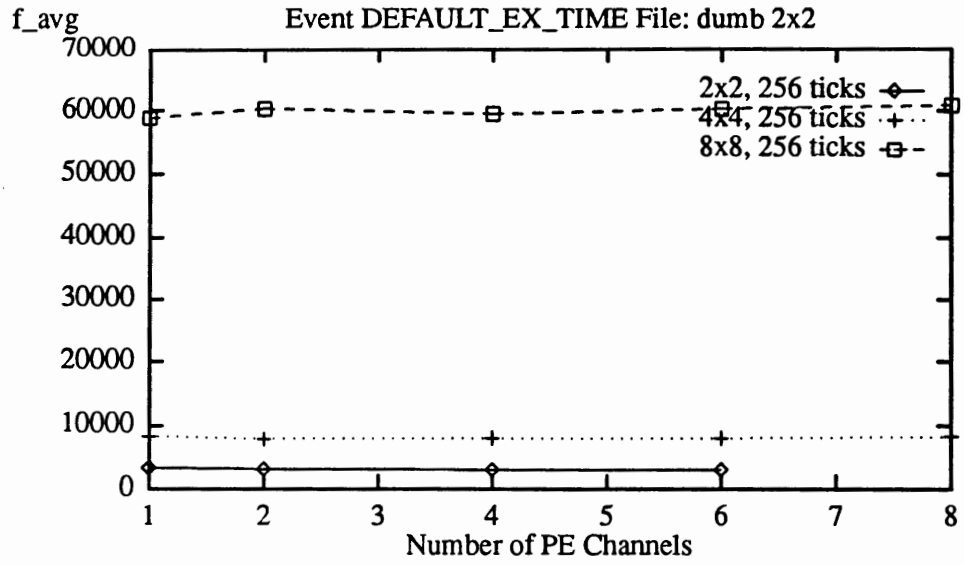


Figure 23. Average execution time for varying number of PE channels (fixed computation time).

the number of PE channels.

Figure 24 shows average execution time for 4 node mesh configuration, measured for 1, 2, 4 and 6 PE channels in the network. From the figure we see, that the best possible execution time is achieved for 4 and 6 channels. After minimizing the number of pins in the router chip, the optimal configuration would be the one with four PE channels. We reduced execution time by about 10% compared to the worst case (1 PE channel).

Figure 25 shows average execution time for the 16 node mesh configuration. For this network, the best number of PE channels is two. For more PE channels, execution time is larger. It is an illustration of the impact of the routing algorithm on the execution time. In the case of our program, we pay more for bad routing while the messages are sent than we gain when receiving messages by the nodes. Adding PE channels into the network actually worsens the performance of the program. For the best case (two PE channels) execution time is about 3% less than the worst case.

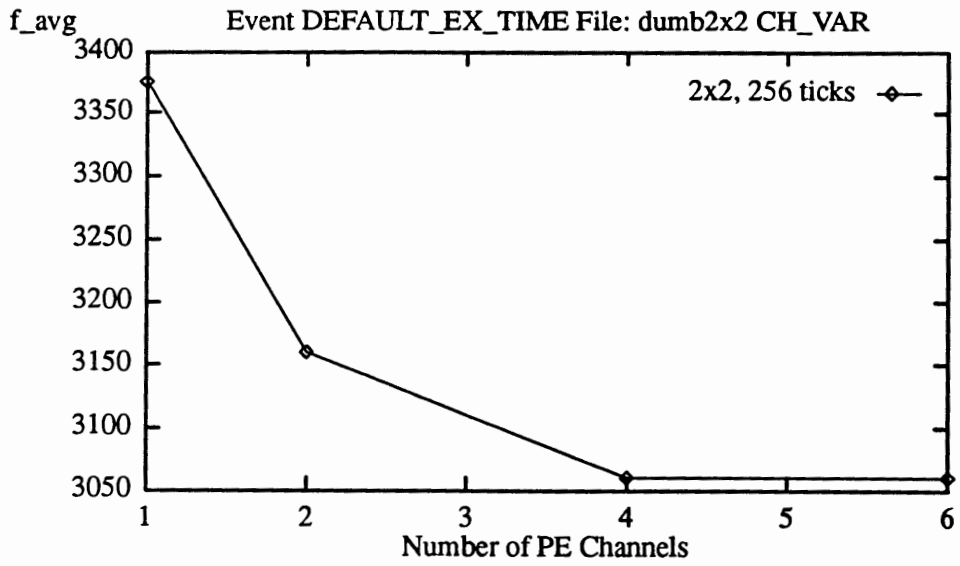


Figure 24. Detail of Figure 22 for 2x2 mesh and varying number of PE channels.

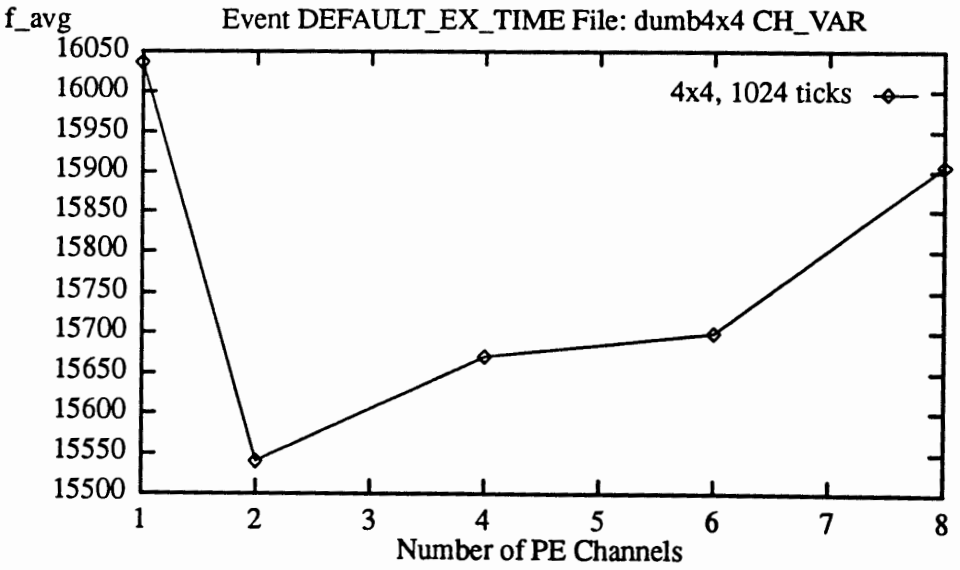


Figure 25. Detail of Figure 22 for 4x4 mesh and varying number of PE channels.

Figure 26 shows average execution time for 8x8 mesh network. For this configuration, the best number of PE channels is one. Let's analyze the behavior of a program for the 8x8 mesh. The computation time was equal 4096 ticks. For comparison, a similar experiment has been

performed for computation time set to 256 ticks (Figure 27).

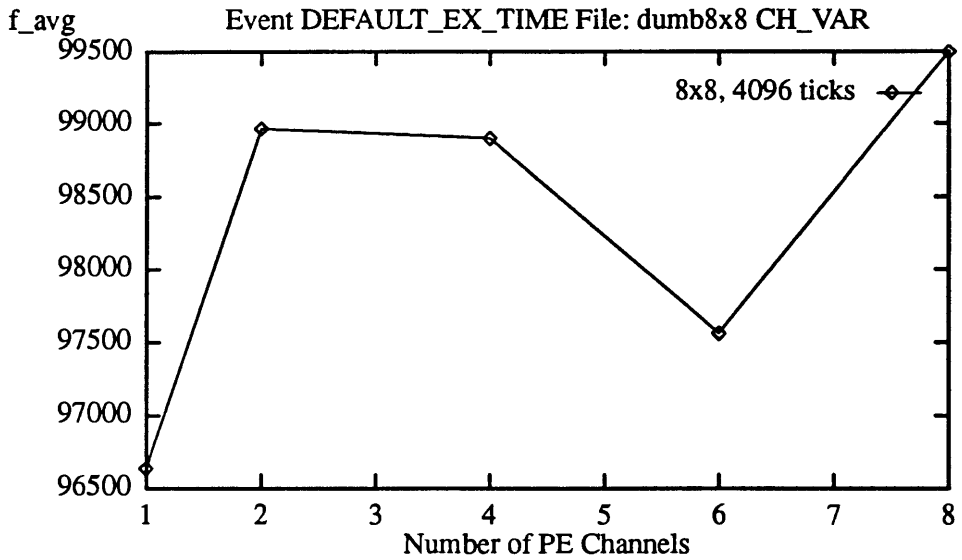


Figure 26. Detail of Figure 22 for 8x8 mesh and varying number of PE channels.

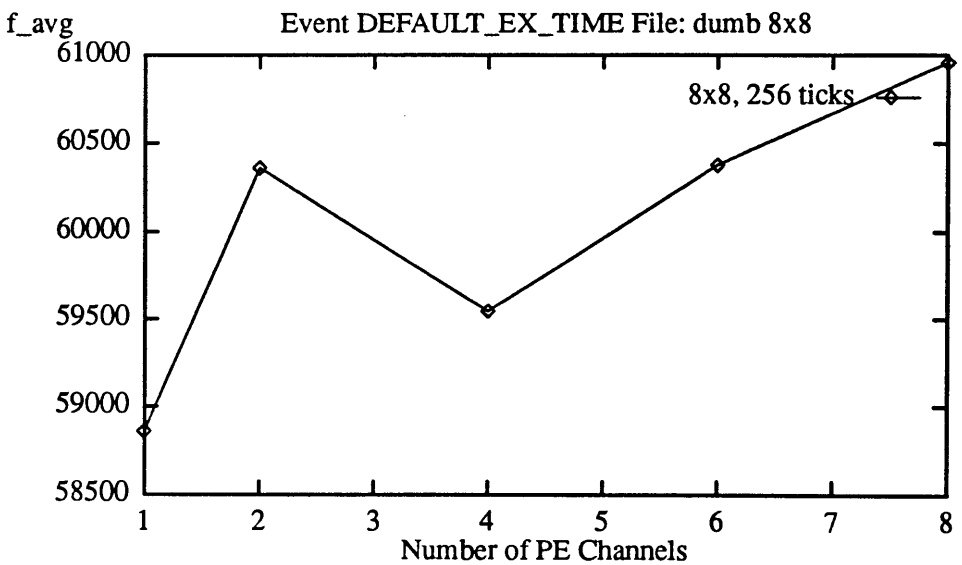


Figure 27. Detail of Figure 23 for 8x8 mesh and varying number of PE channels.

From Figures 26 and 27 we see that the local minimum moved from 6 PE channels (Figure 26) to 4 PE channels (Figure 27). This behavior may be explained by the changed message traffic caused by shortened computation time. Times at which messages arrived to the routers as well as times the messages were forwarded towards their destinations have changed. Also, some nodes already started sending messages in next iteration while others were still receiving messages from the previous iteration.

Summary of Observations about Node Execution Time

- Execution time is proportional to the number of messages generated by the program.
- Execution time is proportional to the size of the network (messages travel longer distances).
- From the performed experiments it is not clear how many PE channels should be used to obtain the minimum execution time. Network performance is very sensitive to the message traffic due to the inefficient algorithm for routing messages.

Message Latency.

Message latency is defined as the time period starting when a message has been generated by the source node, and ending when the last flit of a message arrives at the destination node. For this set of experiments the latency has been collected for each message. Also the average message latency in each node has been collected using the default latency option of the instrumentation preprocessor. Average message latency is calculated by summing the latencies of all messages and dividing by the total number of messages in the network. Measurements of latency have been performed for fixed computation time (256 ticks). Figure 28 shows the average message latency for 4, 16 and 64 node mesh network.

Latency grows with the size of the network (messages have to travel longer distances). Also latency grows with the number of PE channels in the network (messages have to wait

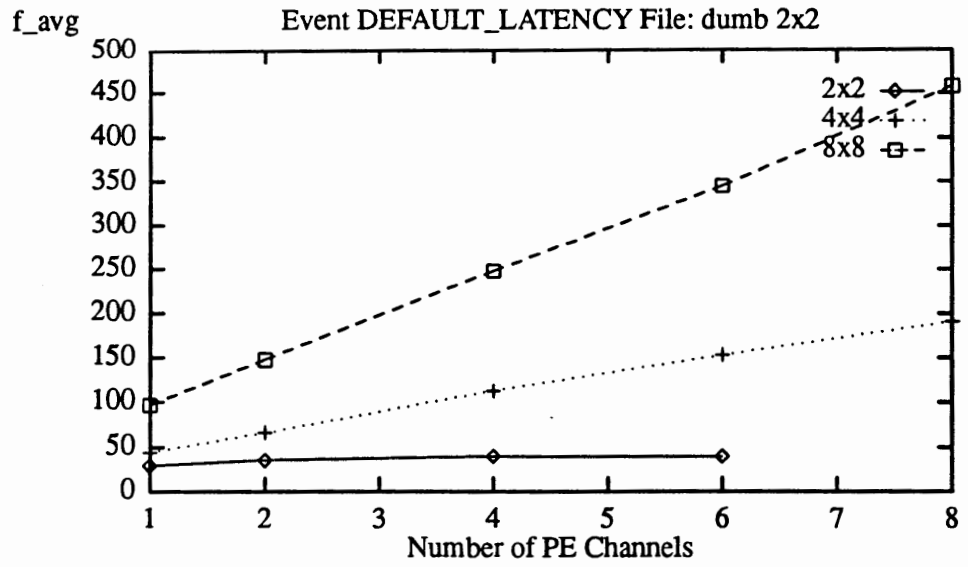


Figure 28. Average message latency for varying number of PE channels.

longer in the FIFO queues for this pattern of message traffic). Figures 29, 31 and 33 show the latency of every message in the 16 node mesh network (1 point corresponds to 1 message). The corresponding frequency distributions for message latency are presented in Figures 30, 32 and 34. Figure 29 shows the latencies for a network with one PE channel. The x-axis shows the time at which a message was delivered to the destination, and the y-axis shows the latency of that message. By subtracting the latency from the time at which message was received, we can obtain the time at which the message was generated.

The density of points which have low latency (about 50 ticks) suggests that most messages were received with similar delay. The x-axis values of points presented in the graph indicate the times when the network nodes were actively receiving messages. In this experiment periods of network inactivity indicate the end of receive activity and start of a new iteration of the program. The messages are always cleared from the network before subsequent iterations start.

The data from Figure 29 have been transformed into the frequency distribution graph, shown in Figure 30. From Figure 30 we see that: 25% of messages had latency less than 32

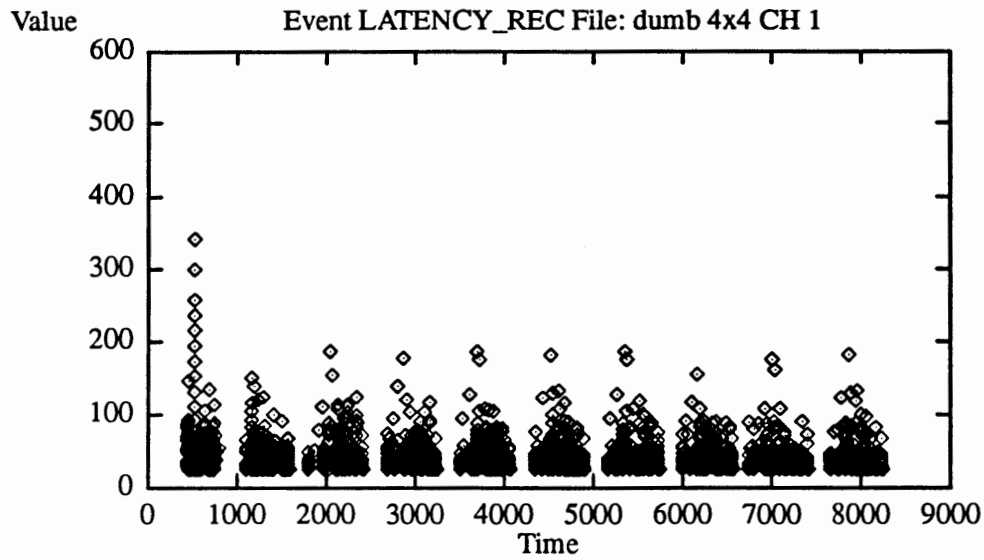


Figure 29. Message latency for 16 node mesh network with 1 PE channel.

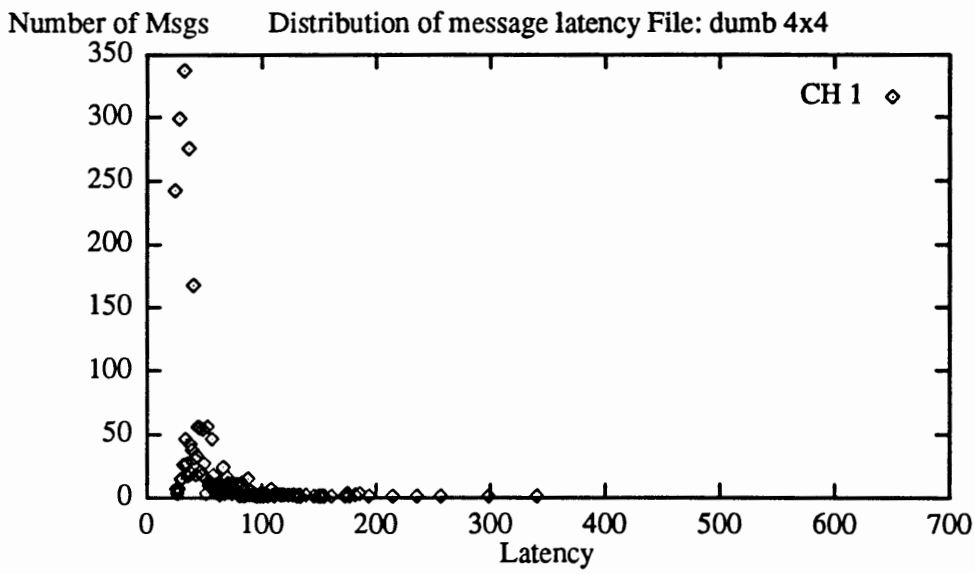


Figure 30. Distribution of message latency for 16 node mesh network with 1 PE channel.

ticks, 50% of messages had latency less than 37 ticks, and 75% of messages had latency less than 48 ticks. The average message latency was 44 ticks. We will show how these data change for different number of PE channels in the network.

Figure 31 shows the latencies for 16 node mesh with two PE channels. Comparing with Figure 29, we notice that points are spread more uniformly along the y-axis. That means, that more messages had greater latency for two channels. However, not all the messages are removed from the network before the start of new iterations. Some messages still travel towards the destination nodes after other nodes already have finished computation phase starting a new broadcast phase. Analyzing the distribution function (Figure 32) we see that: 25% of messages had latency less than 40 ticks, 50% of messages had latency less than 55 ticks, and 75% of messages had latency less than 82 ticks. The average message latency was 66 ticks.

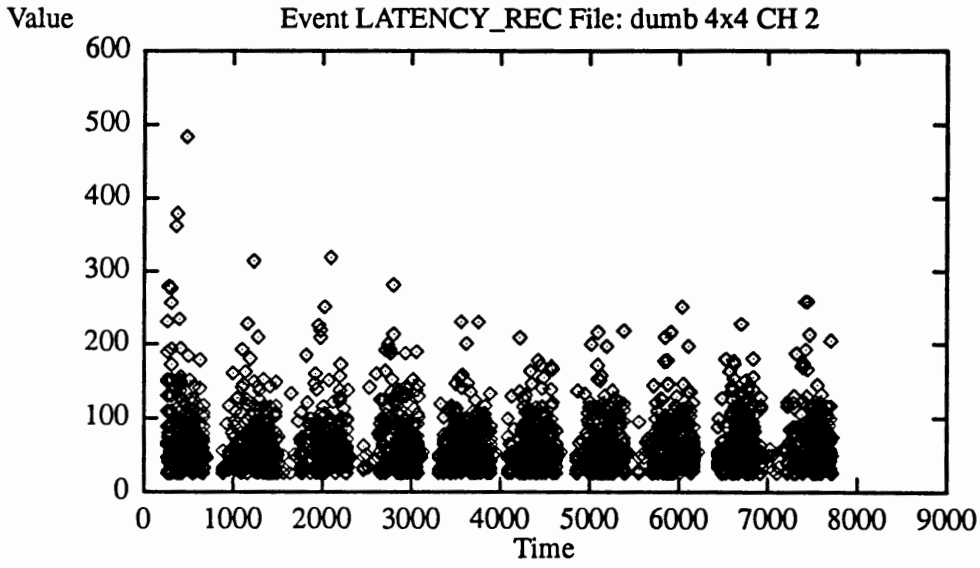


Figure 31. Message latency for 16 node mesh network with 2 PE channels.

Similar graphs have been collected for the network with 4 PE channels. Referring to the message latency graph (Figure 33), we see that network is not idle between iterations. The iterations overlapped, due to the small computation time of 256 ticks. The network does not have time to remove all the messages before new messages are created. Also latencies of messages are greater than in the networks with one and two PE channels (Figure 29 and 31). That is natural, since messages wait longer in the buffers, ready for their turn to take over the channels.

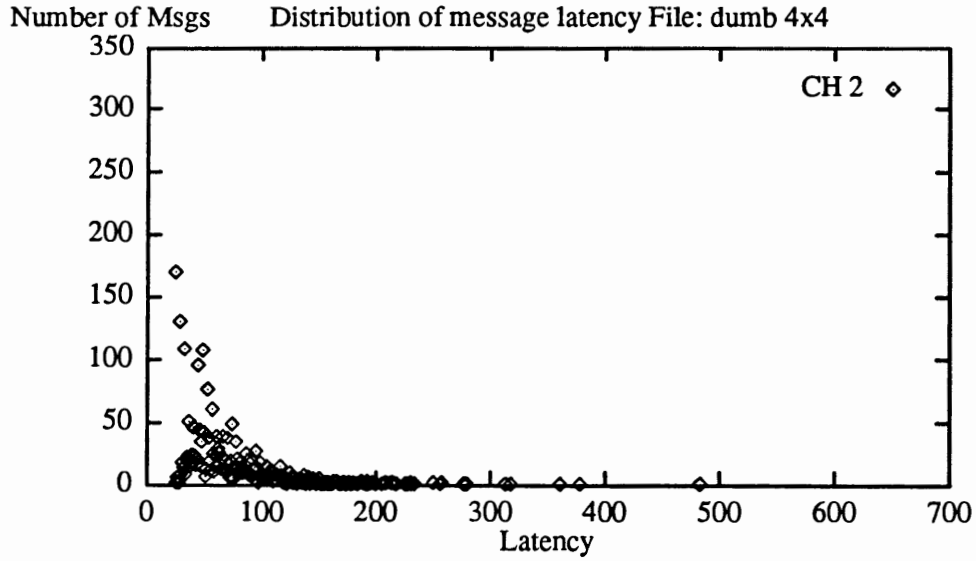


Figure 32. Distribution of message latency for 16 node mesh network with 2 PE channels.

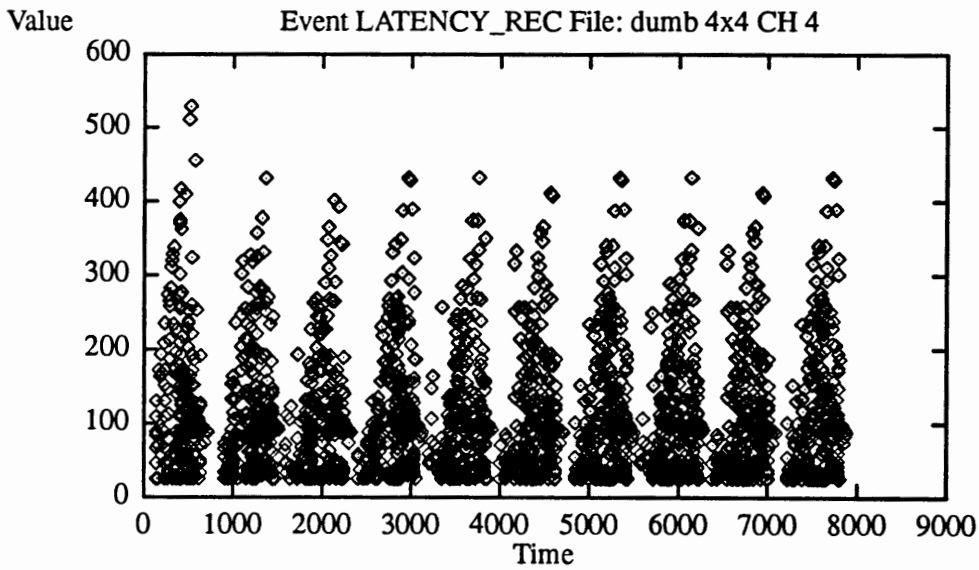


Figure 33. Message latency for 16 node mesh network with 4 PE channels.

From the distribution function (Figure 34) we see that: 25% of messages had latency less than 45 ticks, 50% of messages had latency less than 92 ticks, and 75% of messages had latency less than 152 ticks. The average message latency was 113 ticks.

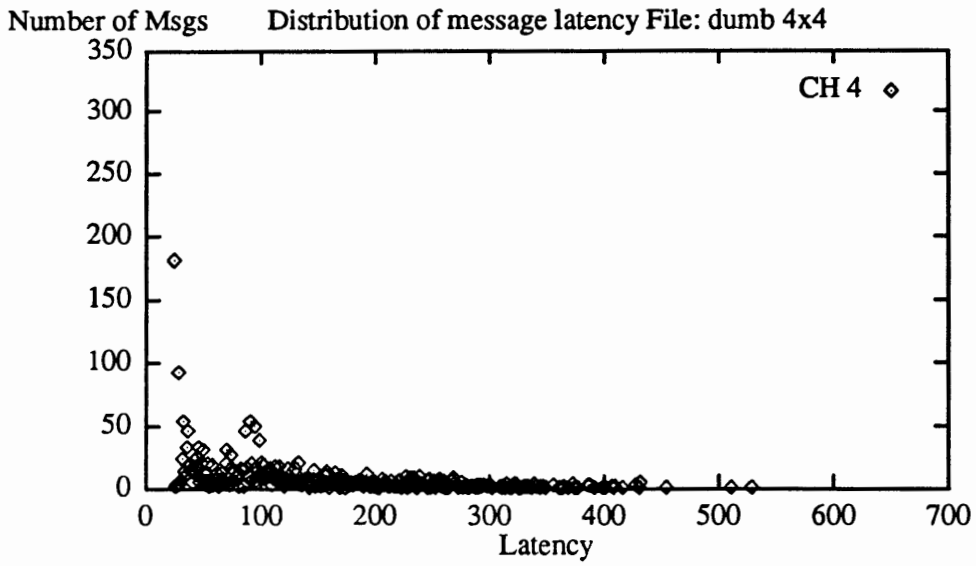


Figure 34. Distribution of message latency for 16 node mesh network with 4 PE channels.

Summary of Observations about Message Latency.

From the analysis of the performed experiments we observe that for networks with several PE channels:

- The maximum latency is higher,
- A greater percentage of messages have higher latency. For example, the 50% point on the distribution graph is moving to the right on the x-axis for more PE channels, indicating that the latency of the message that is in the 50th percentile is higher.
- Higher network contention causes message latency to grow. Network contention is a function of the number of PE channels, since more messages can be injected into the network in a shorter time, causing messages to be blocked for the longer period of time.

To better understand latency as a measure of network performance we performed a simplified experiment in which the network is empty. For this experiment, we used a 16 node mesh network, varying the number of PE channels. We sent just one broadcast message from node 0.

As before, the algorithm starts sending messages to node 1, then 2, ... 15. Similarly to the previous figures, we show the message latency versus time when the message is received. Figure 35 shows the latency for the network with one PE channel.

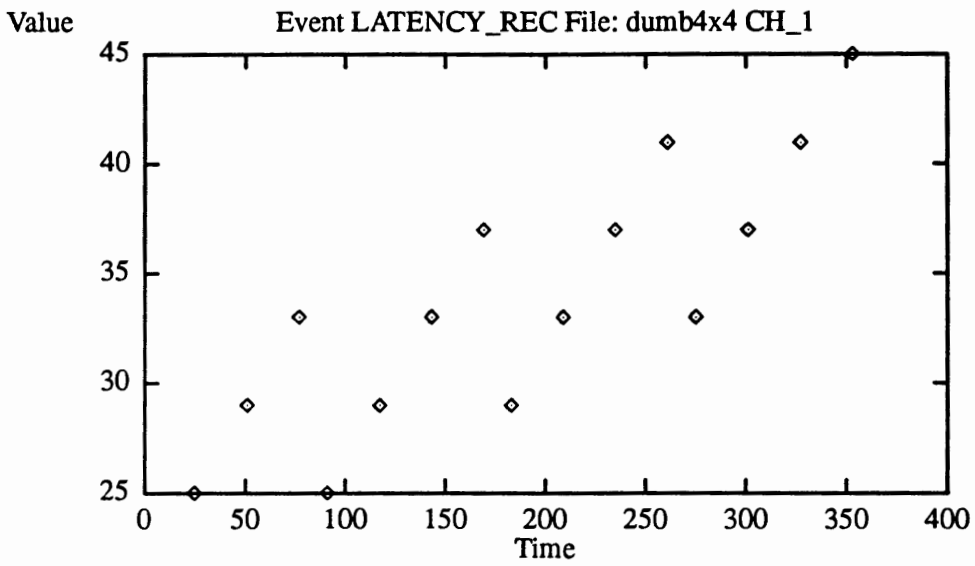


Figure 35. Message latency for 16 node mesh network with 1 PE channel with no contention.

Figure 36 shows the latencies and execution times mapped to the specific nodes of the network. We notice that latency values correspond directly to the distance the message has to travel. We have six different levels of latency values, which is equal to the diameter of the network. This simple correspondence of latency to distance is no longer the case for the increased number of PE channels.

Figures 37 and 38 show the message latencies for a 4x4 mesh network with 2 and 4 PE channels respectively. For the case when the number of PE channels is greater than one, the number of different latency values is no longer equal to the diameter of the network. Instead, there are as many levels of similar latency values as the number of PE channels per node.

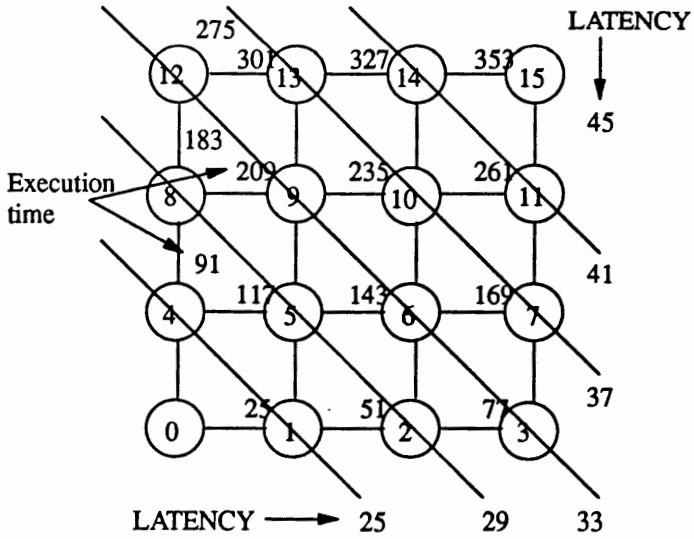


Figure 36. 4x4 mesh network with 1 PE channel with no contention: latency and execution time.

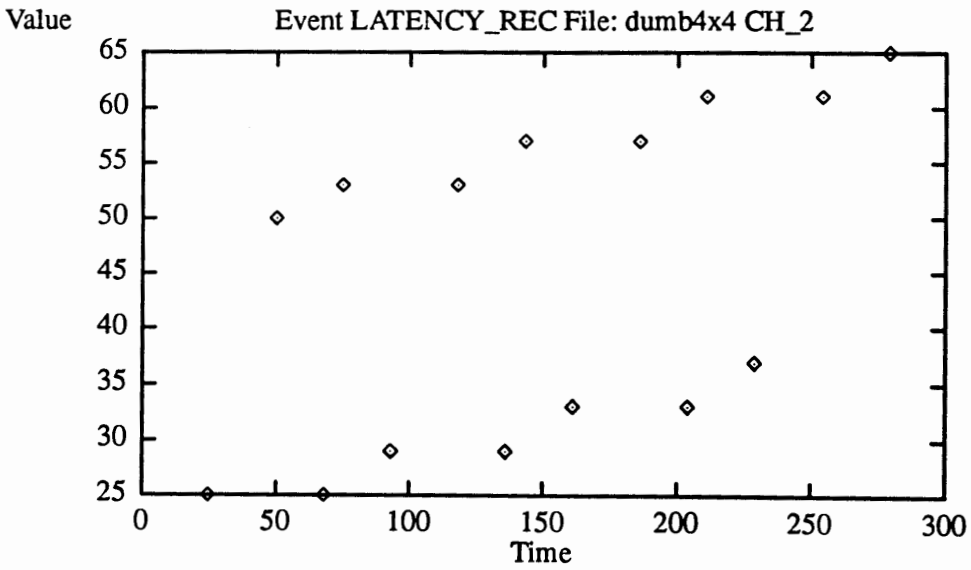


Figure 37. Message latency for 16 node mesh network with 2 PE channels with no contention.

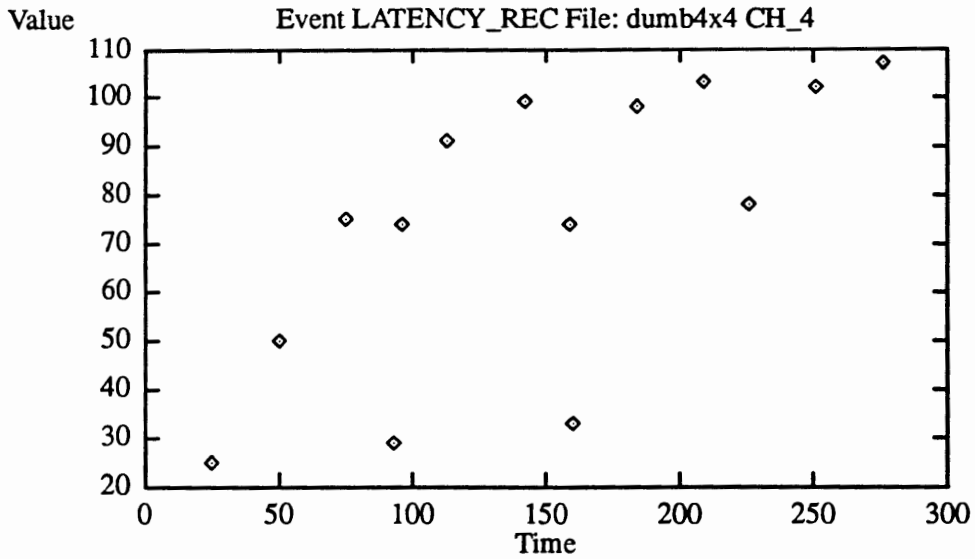


Figure 38. Message latency for 16 node mesh network with 4 PE channels with no contention.

Network Activities.

To illustrate the network activities in the N-body application program, two configurations have been chosen: the 4x4 mesh network with one PE channel and eight PE channels. The eight PE channel configuration has been chosen to obtain maximum network contention, and therefore better contrast with the network where the message traffic is constrained by a limited number of PE channels. The computation time is fixed and set to 256 ticks.

In the following experiments, a time stamp has been collected in each node, when the message was generated (send instruction), and also when the message was received (before and after each receive instruction).

Sending Messages.

Figure 39 shows how much time each processor needs to send a broadcast message in each iteration for the network with one PE channel. Each point on a graph corresponds to a message sent in the broadcast phase. Similarly to the latency graphs, we notice ten activity

periods: ten iterations of the program. The time between each send period is dedicated to receiving messages, and computation in each node.

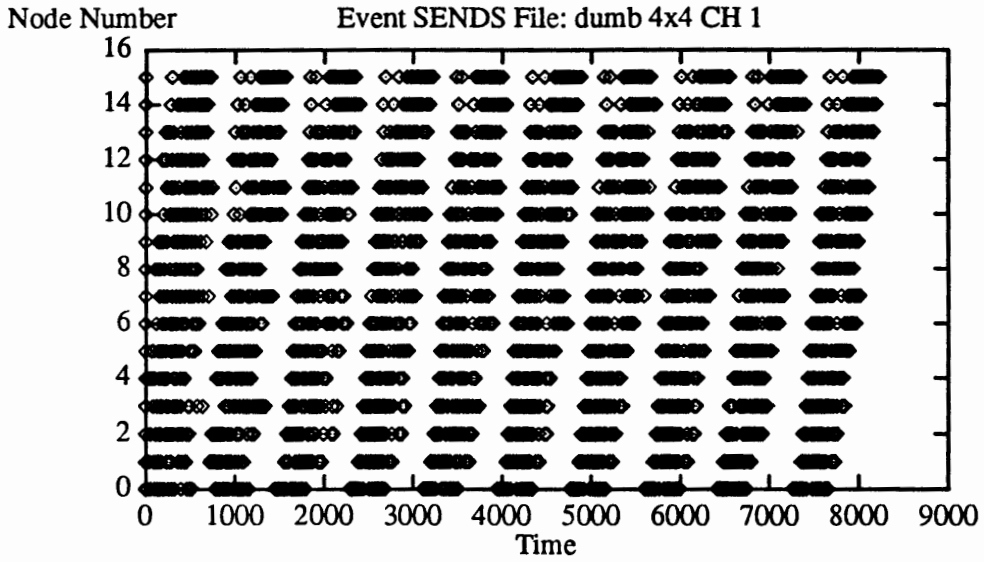


Figure 39. Send activity in 16 node mesh network with 1 PE channel.

Looking at the message generation pattern for the first iteration, we notice that at some nodes there is a significant delay before the second message gets created. The first message created is dedicated to node 0, and according to the dimension-order routing algorithm, it has to travel through dimension X first, and then through dimension Y. Since the mesh is asymmetric, a message generated by node 15 has to travel all the way through X dimension (3 hops), and then do the same for Y (another 3 hops). However, the same route has already been taken by messages generated by nodes 14, 13 and 12. Thus the message from node 15 has to wait until all messages going through its route free the required channels. Moreover, since all nodes start sending to node 0 first, messages going from nodes 4, 5, 6 and 7 also compete for the channel between nodes 0 and 4. Similarly, messages going from nodes 8, 9, 10 and 11 compete for channels between nodes 8 and 0 (Figure 40). The delay is visible for the nodes that are situated on the boundary of the network. In subsequent iterations the drawback of the bad routing

algorithm is visible only for nodes 14 and 15 but actually occurs for other boundary nodes.

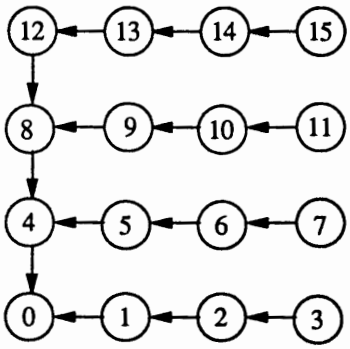


Figure 40. Routes allocated for first message generated in a broadcast phase (destination - node 0).

Figure 41 shows the send activity in the 16 node mesh network with eight PE channels. The first eight messages are generated at the same time, and have to compete for the network channels. When a message leaves the buffer, a new one is created and takes its place in the PE buffers.

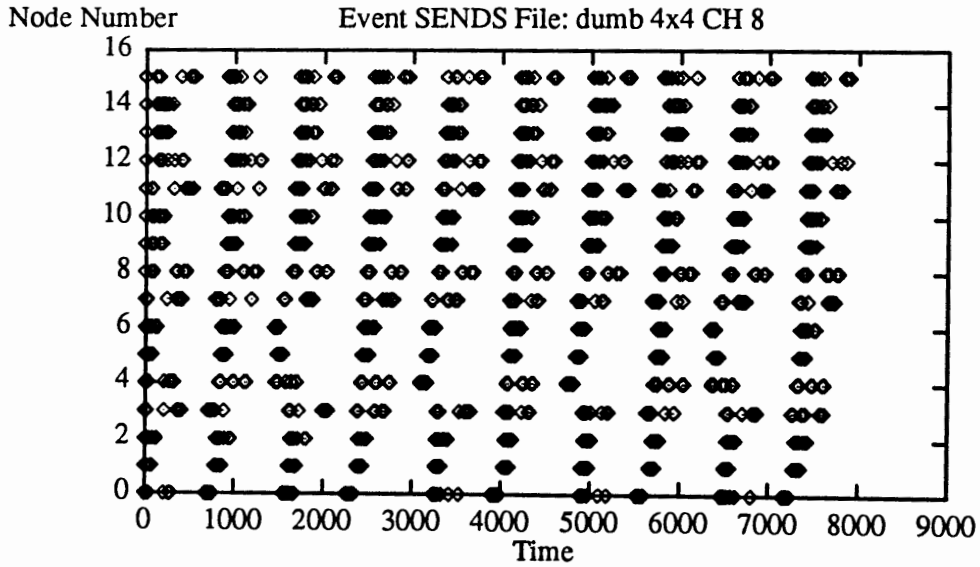


Figure 41. Send activity in 16 node mesh network with 8 PE channels.

Generally, send activity takes less time for eight PE channels than for one PE channel. We can also notice how the network asymmetry impacts the send activity. For nodes that are located on network boundaries, specifically nodes: 0, 4, 8, 12, 3, 7, 11 and 15 there are delays in message generation, caused by increased contention in that region of the network.

Receiving Messages.

To monitor receive activity in the network, time stamps have been collected for the start of the activity (when the node starts looking for the message) until the end of the activity (when the last flit of the message gets into the node). For the one PE channel network (Figure 42), the receive period in each iteration is short compared to the send period in this configuration. This is due to the fact that nodes start looking for messages to be received after they are done with broadcasting. In the meantime most of the messages have already arrived at their destinations.

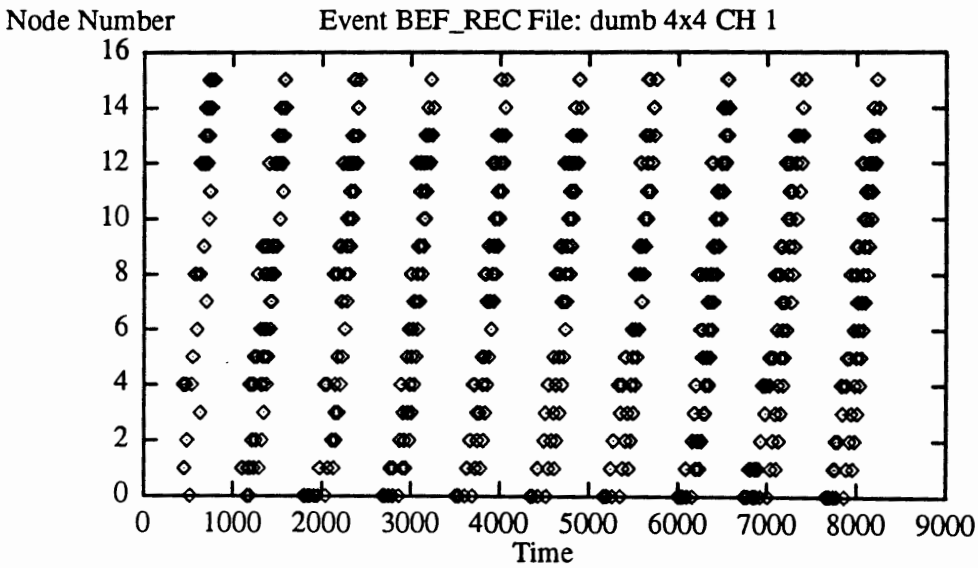


Figure 42. Receive activity in 16 node mesh network with 1 PE channel.

In contrary, for the eight PE channels mesh configuration, receive periods are much longer (Figure 43). This is due to the fact that send activity is much shorter for this network, and

messages are still on their way when nodes start looking for them. To shorten the execution time of the program, one might change the application program so that nodes are always ready to receive messages. When the message arrives at its destination, it would be immediately consumed. This way send activity would be overlapped with receive activity, and thus the overall execution time would be shorter. Also the latency of the messages would be shorter, because they would not be blocked at the destination.

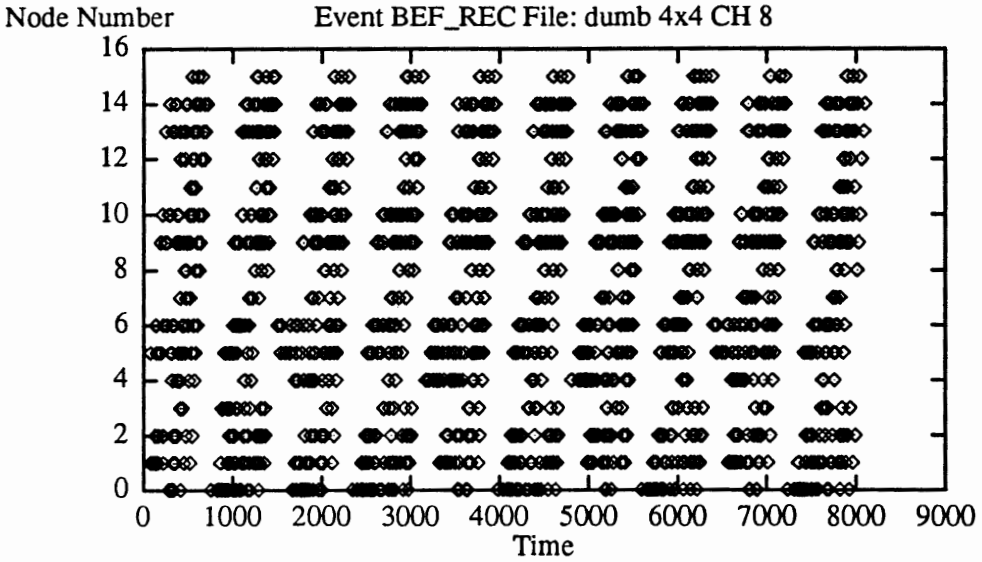


Figure 43. Receive activity in 16 node mesh network with 8 PE channels.

Summary of Observations.

While investigating the best number of PE channels for mesh interconnection network, we found out that performance of the network depends not only on the number of PE channels used, but also on the algorithm for distributing messages over the network. The algorithm according to which the messages are generated has far greater impact on network performance than the number of PE channels. To make effective use of more PE channels one should first fix the performance problems caused by an inefficient message generation scheme. Specifically,

for broadcast algorithms, messages should be sent to furthest destinations first. Also, the messages should be sent in different directions to make good use of the network channels. After improving the performance of the program with respect to the message generation pattern, one would see the benefits obtained by adding multiple PE channels.

Similar observations about the impact of the routing algorithms on performance of a hypercube network with different numbers of PE channels have been made by Johnson and Ho [36]. In [36] specific algorithms have been developed for networks with multiple PE channels to take the advantage of additional available hardware resources. These observations agree with the experimental results that we obtained when solving the network design problem. This validates our design methodology and correctness of the Parsim Common Environment.

CHAPTER VI

CONCLUSION

In this thesis we have developed and evaluated a methodology for performance evaluation of multicomputer system designs. Our methodology is based on simulation. A simulator is used to model the behavior of the computer system under development. We prepare application programs that contain performance monitoring code and execute them using the simulator. Data traces collected during the simulation are then analyzed and presented using visualization tools.

Key features of our methodology are "plug and play" simulation and allowing for hardware/software interaction during the hardware design process. We can easily use different simulators with our performance evaluation system to balance functionality, accuracy and performance of the simulator. The simulator executes user code, which makes it possible to experiment with different hardware configurations to check the impact of chosen hardware parameters on software efficiency.

We have built a software performance evaluation environment for multicomputer interconnection network design. Our tool, called PCE (Parsim Common Environment), contains a preprocessor for instrumenting application programs with performance monitoring code, a network simulator, and analysis and visualization modules.

Using PCE, we have investigated a specific network design example. With the support of PCE we found the problems that the hardware engineer would encounter when designing multicomputer interconnection networks. We also found ways to improve network performance. Therefore our software system proved to be useful during the hardware design process. It helped us to spot the performance problems, to explain why they happened, and to find the ways

to solve them.

Moreover, using our tools made performance evaluation easy. To set up performance experiments we annotated the application programs with flags indicating the performance measurements to be made. A preprocessor converted the annotations into performance monitoring code. From the graphical user interface we were able to set simulation parameters. We did not need to manually alter the variables in the programs. The simulation environment could perform a set of experiments automatically varying the chosen design parameter. We could set up the performance analysis environment before the simulation run by specifying what functions should be applied to data collected for a specific monitored event. The simulation and analysis of data could be performed in the background, since for bigger network configurations they were very time consuming. After performing data analysis, the results were presented in a graphical form. We could observe communication patterns in the network, that would be very hard to present without software analysis and visualization tools. Additional analysis of data could be performed on collected traces when a more thorough explanation of results was required.

The tools saved us many hours of confusion and frustration that we would have to spend trying to interpret the collected data for every experiment. Since we had our share of tedious manual evaluation of data before the development of the system we appreciate even more its benefits: flexibility, convenience, and time savings. In the preparation phase of performance instrumented application programs we gain the flexibility and convenience when using our tools. The preprocessor performs insertion of performance monitoring code into the programs based on annotations made by the user. Parameters and configuration of the network can be altered from the graphical user interface, without modifying the program before every simulation run. Prior to the development of PCE, manual changes of simulation variables with additions of monitoring code were required in application programs for each experiment.

At the analysis stage we also gain flexibility of use, but most importantly we gain a significant amount of time when analyzing data. To illustrate the time saving benefits, let's compare several examples of experiments performed with and without using our performance evaluation system. Before the tools were developed, we had to perform several validation experiments of the simulator model. One of the experiments involved processing 25 kbytes of data. The traces were a collection of receive times and latencies of every message generated by the algorithm solving the N-body problem (three iterations). Trying to sort and present data for every processor in one representative configuration of 16 node network took about 16 man-hours of work. Also, analysis of larger networks without computer support would be impossible. For comparison, the examples presented in Chapter V of this thesis involved similar analysis of data for 5 sets of simulation parameters for 16 node interconnection network for 10 iterations of N-body algorithm. The size of processed data traces was 900 kbytes and the analysis of this volume of data was made in three minutes of computation time. The analysis was performed in a multiuser environment on a Sparc-4 workstation.

In another validation experiment for the simulator we analyzed 64 node mesh and hypercube network topologies. In each node we collected the execution times of every iteration of the N-body algorithm and then calculated the average execution time for each loop. The analysis took us about 10 man-hours and involved 20 kbytes of data. Using PCE such analysis takes a couple of seconds.

Using software tools for performance evaluation of parallel systems is not a new idea. Many similar performance monitoring systems already exist. They are built on real, performance-instrumented parallel computers. However, these systems are intended for performance tuning of parallel software applications. Replacing the parallel computer with the simulator gives us the flexibility to evaluate a set of computer architectures supported by the simulator. Using our approach, computer designers can make architectural decisions before they start

building prototypes.

Simulation as a base for estimating computer performance has been a common practice, since mathematical models are usually too inaccurate to be effectively used. Setting up a performance experiment involves the preparation of an application program (or a set of programs) in such a way that specific events are detected during a program run and performance data of interest can be collected. With a graphical user interface we can define a set of experiments to perform and monitor program performance for different values of specified architectural variables. Users of our system are released from customizing the application for performance measurements for every event of interest.

In multicomputer systems, where huge volumes of performance data are usually generated, it is impossible to perform analysis of the obtained sets of numbers without computer support. We have provided a set of functions that can be executed on a collection of data to extract the information of interest. The final, analyzed set of data can be presented using graphical displays.

LIMITATIONS AND FUTURE WORK

The factor that always plays an important role when designing any system is time. It limited us to having only a small set of performance visualization tools, specifically two-dimensional displays. We believe that having more sophisticated graphical presentation forms would benefit us by giving us an insight into many design problems.

However, developing graphical tools is very time consuming, and usually hardware designers stick with simple sets of tools that are commonly available. Preparing data traces that would match the format needed for existing performance visualization tools would provide an alternative to the commonly used two-dimensional displays. A simulator, if designed with that goal in mind, would be able to interface with complex visualization system.

However, the problem that arises is the lack of standards for performance data traces. This means that every visualization system requires data in a different form. Therefore, one must decide beforehand what visualization system to use. Based on this decision the user would develop a simulator to provide data traces in the required format.

We started development of our performance monitoring system after the simulator had already been written. This involved the decision of creating our own visualization tools, which turned out to be very time consuming to develop and thus limited us to simple display forms. In our future work we will alter the simulator to provide data traces acceptable by the performance visualization tool of our choice, and evaluate the benefits of displaying data using various presentation techniques.

REFERENCES

[1] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, Inc., New York, New York, 1993.

[2] W. J. Dally and C. L. Seitz, "The Torus Routing Chip," *Distributed Computing*, vol. 1, pp. 187-196, Springer-Verlag, 1986.

[3] J. A. Payne, *Introduction to Simulation: Programming Techniques and Methods of Analysis*, McGraw-Hill Book Company, New York, New York, 1982.

[4] D. A. Reed et al., *The Pablo Performance Analysis Environment*, Technical Report, Department of Computer Science, University of Illinois, Urbana, Illinois, 1992.

[5] D. K. Bradley and J. L. Larson, "A Parallelism-Based Analytic Approach to Performance Evaluation Using Application Programs," *Proceedings of the IEEE*, vol. 81, no. 8, pp. 1126-1135, August 1993.

[6] A. D. Malony and D. A. Reed, *Performance Analysis Tools and Techniques for High Performance Parallel Computers*, Supercomputing '93, Tutorial, Portland, Oregon, 1993.

[7] M. Simmons and R. Koskela, *Performance Instrumentation and Visualization*, ACM Press, New York, New York, 1990.

[8] A. D. Malony, D. A. Reed, and D. C. Rudolph, "Integrating Performance Data Collection, Analysis, and Visualization," in *Performance Instrumentation and Visualization*, ed. M. Simmons and R. Koskela, ACM Press, New York, New York, 1990.

[9] M. Simmons and R. Koskela, *Instrumentation for Future Parallel Computing Systems*, ACM Press, New York, New York, 1989.

[10] A. Gottlieb, K. Hwang, and S. Sahni, "Special Issue on Tools and Methods for

Visualization of Parallel Systems and Computations," *Journal of Parallel and Distributed Computing*, vol. 18, no. 2, Academic Press, Inc., San Diego, California, June 1993.

[11] L. Kleinrock, "On the Modeling and Analysis of Computer Networks," *Proceedings of the IEEE*, vol. 81, no.8, pp. 1179-1190, August 1993.

[12] T. Feng, "A Survey of Interconnection Networks," *IEEE Computer*, pp. 5-18, December 1981.

[13] L. N. Bhuyan, Y. Quing, and D. P. Agrawal, "Performance of Multiprocessor Interconnection Networks," *IEEE Computer*, pp. 5-16, February 1989.

[14] D. Lenoski, J. Laudon, and K. Gharachorloo, "The Stanford Dash Multiprocessor," *IEEE Computer*, vol. 25, no. 3, pp. 63-79, March 1992.

[15] W. J. Dally, "Performance Analysis of k-ary n-cube Interconnection Networks," *IEEE Transactions on Computers*, vol. 39, no. 6, pp. 775-785, June 1990.

[16] D. A. Reed and R. Fujimoto, *Multicomputer Networks: Message Based Parallel Processing*, The MIT Press, Cambridge, Masachusets, 1987.

[17] W. J. Dally, "A Universal Parallel Computer Architecture," *New Generation Computing*, vol. 11, pp. 227-246, 1993.

[18] W. J. Dally, "Fine-Grain Message-Passing Concurrent Computers," *3rd Conference on Hypercube Concurrent Computers and Applications, Proceedings*, pp. 2-12, 1988.

[19] R. Duncan, "A Survey of Parallel Computer Architectures," *IEEE Computer*, pp. 5-16, February, 1990.

[20] W. J. Dally, "Virtual Channel Flow Control," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 2, pp. 194-204, March 1992.

[21] J. H. Kim and A. A. Chien, *Evaluation of Wormhole Routed Networks under Hybrid Traffic Loads*, Submitted for publication, Department of Computer Science, Urbana,

Illinois, 1992.

[22] W. J. Dally and C. L. Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Transactions on Computers*, vol. C-36, no. 5, pp. 547-553, May 1992.

[23] A. A. Chien and J. H. Kim, "Planar-Adaptive Routing: Low cost Adaptive Networks for Multiprocessors," *19th Annual International Symposium on Computer Architecture, Proceedings*, pp. 268-277, 1992.

[24] K. Bolding, *Chaotic Routing - Design and Implementation of an Adaptive Multicomputer Network Router*, University of Washington PHD Thesis, 1993.

[25] R. Traylor and D. Dunning, *Routing Chip Set for Intel Paragon Parallel Supercomputer,* Seminar at Portland State University, Portland, Oregon, 1993.

[26] J. K. Ousterhout, *An Introduction To Tcl and Tk*, Copyright 1993 Addison-Wesley Publishing Company, Inc., Draft on Internet: sprite.berkeley.edu via public FTP, 1992.

[27] B. J. Porcella, *Parsim Users Manual*, Technical Report #PSU-EE-93-010, Department of Electrical Engineering, Portland State University, Portland, Oregon, 1993.

[28] A. M. Kolinska, *Performance Monitoring System for Parsim Simulator*, Technical Report #PSU-EE-93-020, Department of Electrical Engineering, Portland State University, Portland, Oregon, 1993.

[29] A. M. Kolinska, *User Defined Events in the Measurement System for Parsim Simulator*, Technical Report #PSU-EE-93-030, Department of Electrical Engineering, Portland State University, Portland, Oregon, 1993.

[30] A. M. Kolinska, *A Report on Performance Monitoring System for Parsim Simulator - Command Line Prototyping Tool*, Technical Report #PSU-EE-93-040, Department of Electrical Engineering, Portland State University, Portland, Oregon, 1993.

[31] A. D. Malony and K. Nichols, "Standards Working Group Summary," in *Performance Instrumentation and Visualization*, ed. M. Simmons and R. Koskela, ACM Press, New York, New York, 1990.

[32] A. M. Kolinska, *Parsim Simulations in Presence of no Contention for Mesh Topology*, Technical Report #PSU-EE-93-050, Department of Electrical Engineering, Portland State University, Portland, Oregon, 1993.

[33] D. K. Bradley, B. A. Nazief, D. C. Grunwald, and D. A. Reed, "Picasso: An Experiment in Hypercube Operating System Design," *3rd Conference on Hypercube Concurrent Computers and Applications, Proceedings*, pp. 364-373, 1988.

[34] C. L. Seitz, "The Cosmic Cube," *Communications of ACM*, vol. 28, no. 1, pp. 22-33, January 1985.

[35] A. L. Couch, "Locating Performance Problems in Masively Parallel Executions," *Proceedings of the IEEE*, vol. 81, no. 8, pp. 1116-1125, August 1993.

[36] S. L. Johnsson and C. T. Ho, *Optimum Broadcasting and Personalized Communication in Hypercubes*, Technical Report #YALEU/DCS/TR-610, Department of Computer Science, Yale University, December 1987.

APPENDIX

The following are the application programs used to generate data for the Network Design Example in Chapter V. Files with extension *.cpl* are the programs with inserted performance monitoring flags. Files with extension *.pre* are the performance instrumented programs obtained by running the preprocessor on the application programs with extension *.cpl*. (Some comments have been altered later to indicate the right names of the files and their purpose).

NETWORK PERFORMANCE WITH LOW CONTENTION

Example 1.

dumb_ex1.cpl

```
c CONFIG_DT 0 0x00000220
c CONFIG_DT 1 0x00002090
c CONFIG_DT 2 0x00010448
c CONFIG_DT 3 0x1bfa2949
c CONFIG_DT 4 0x2df02948
c LINK_NODE 1
c LINK_ROUTER 1
c CFG_WORM_CH 4 4 4
c CFG_NET 2 4 1
c OS_OVHD 1
```

```
! ************************ example 1 ************************** !
!                                                               !
! This file is used to generate data for tables I and II.       !
! Node 0 sends a message to nodes 1, 2 and 3                    !
! in 4x4 mesh network.                                          !
!                                                               !
! ************************ dumb_ex1.cpl ************************ !
```

```
        define max m
        define dest e    ! destination node
        LOAD a 1         ! message type
        LOAD c @MSG      ! message length, passed from GUI
```

```
        LOAD max 4


! ---------------------- MULTICAST SESSION --------------------

LOAD dest 0


!c MA START send_macro

   IF my_node EQ zero           ! only node 0 sends messages
        WHILE dest NE max       ! send to node 1, 2, 3
        DO
                IF dest NE my_node
                        send a dest c

                ENDIF

                INC dest

        END0
        ENDWHILE
   ENDIF

!c MA STOP send_macro


! ---------------------- RECEIVE SESSION ----------------------

!c MA START rec_macro

        IF my_node GT zero
                IF my_node LT max       ! nodes 1, 2, 3 receive
                                        ! one message
                        rec_type a g c

                ENDIF
        ENDIF

!c MA STOP rec_macro


HALT


! ---------------------- DEFINITIONS OF USER'S MACROS ----------


! send_macro records time when sending messages:
!         BEF_SEND - indicates start of a send process
```

```
!         AFT_SEND - indicates an end of a send process


!M send_macro
!M
!M before_instr:
!M          instr send:
!M                  output BEF_SEND my_node
!M after_instr:
!M          instr send:
!M                  output AFT_SEND my_node
!M
!M end



! rec_macro records time when receiving messages,
! and their latencies:
!         BEF_REC     - time at which a node is ready to receive
!                       a message
!         AFT_REC     - indicates an end of a receive process
!         LATENCY_REC - latency of a received message,
!                       and time when it arrived


!M rec_macro
!M
!M before_instr: instr rec_type:
!M          output BEF_REC my_node
!M after_instr:
!M          instr rec_type:
!M                  output AFT_REC my_node
!M                  output LATENCY_REC rec_lat
!M
!M end
```

dumb_ex1.pre

```
c CONFIG_DT 0 0x00000220
c CONFIG_DT 1 0x00002090
c CONFIG_DT 2 0x00010448
c CONFIG_DT 3 0x1bfa2949
c CONFIG_DT 4 0x2df02948
c LINK_NODE 1
c LINK_ROUTER 1
c CFG_WORM_CH 4 4 4
c CFG_NET 2 4 1
c OS_OVHD 1

DEFINE counter0 Z
LOAD counter0 0
DEFINE accum0 Y
LOAD accum0 0



! DU START DEFAULT_EX_TIME
OUT B0 my_node timer

! AC START DEFAULT_LATENCY REC_LAT

! CO START DEFAULT_REC_MSG ALL_REC_INSTR


! ************************ example 1 ************************** !
!                                                               !
! This file is used to generate data for tables I and II.       !
! Node 0 sends a message to nodes 1, 2 and 3                    !
! in 4x4 mesh network.                                          !
! Performance monitoring code is inserted into the program.     !
!                                                               !
! ************************ dumb_ex1.pre ************************ !



        define max m
        define dest e    ! destination node
        LOAD a 1         ! message type
        LOAD c @MSG      ! message length, passed from GUI
        LOAD max 4



! ---------------------- MULTICAST SESSION --------------------

        LOAD dest 0
```

```
! MA START send_macro

   IF my_node EQ zero            ! only node 0 sends messages
        WHILE dest NE max        ! send to node 1, 2, 3
        DO
                IF dest NE my_node
OUT U3 my_node my_node
OUT S3 my_node TIMER
                           send a dest c
OUT U4 my_node my_node
OUT S4 my_node TIMER

                ENDIF

                INC dest

        END0
        ENDWHILE
   ENDIF

! MA STOP send_macro



! ---------------------- RECEIVE SESSION -----------------------


! MA START rec_macro

        IF my_node GT zero
                IF my_node LT max        ! nodes 1, 2, 3 receive
                                         ! one message
OUT U0 my_node my_node
OUT S0 my_node TIMER
INC counter0
                           rec_type a g c
OUT U1 my_node my_node
OUT S1 my_node TIMER
OUT U2 my_node rec_lat
OUT S2 my_node TIMER
ADD accum0 accum0 REC_LAT

                ENDIF
        ENDIF

! MA STOP rec_macro



! CO STOP DEFAULT_REC_MSG ALL_REC_INSTR
```

```
! AC STOP DEFAULT_LATENCY REC_LAT

OUT E0 my_node timer
! DU STOP DEFAULT_EX_TIME



IF counter0 GT zero
        DIV fa accum0 counter0
        OUT A0 my_node fa
ENDIF

HALT



! ---------------------- DEFINITIONS OF USER'S MACROS ----------


! send_macro records time when sending messages:
!         BEF_SEND - indicates start of a send process
!         AFT_SEND - indicates an end of a send process


!M send_macro
!M
!M before_instr:
!M          instr send:
!M                  output BEF_SEND my_node
!M after_instr:
!M          instr send:
!M                  output AFT_SEND my_node
!M
!M end




! rec_macro records time when receiving messages,
! and their latencies:
!         BEF_REC     - time at which a node is ready to receive
!                       a message
!         AFT_REC     - indicates an end of a receive process
!         LATENCY_REC - latency of a received message,
!                       and time when it arrived


!M rec_macro
!M
!M before_instr: instr rec_type:
!M          output BEF_REC my_node
!M after_instr:
```

```
!M          instr rec_type:
!M                  output AFT_REC my_node
!M                  output LATENCY_REC rec_lat
!M
!M end
```

Example 2.

dumb_ex2.cpl

```
c CONFIG_DT 0 0x00000220
c CONFIG_DT 1 0x00002090
c CONFIG_DT 2 0x00010448
c CONFIG_DT 3 0x1bfa2949
c CONFIG_DT 4 0x2df02948
c LINK_NODE 1
c LINK_ROUTER 1
c CFG_WORM_CH 4 4 4
c CFG_NET 2 4 1
c OS_OVHD 1

! ************************* example 2 ************************** !
!                                                                !
! This file is used to generate data for tables III and IV.      !
! Node 0 sends a message to nodes 3, 2 and 1                     !
! in 4x4 mesh network.                                           !
!                                                                !
! ************************* dumb_ex2.cpl ********************** !



        define max m
        define dest d    ! destination node
        LOAD a 1         ! message type
        LOAD c @MSG      ! message length, passed from GUI
        LOAD max 4


! -------------------------  MULTICAST SESSION -----------------


        LOAD dest 3
```

```
!c MA START send_macro

   IF my_node EQ zero            ! only node 0 sends messages

        WHILE dest NE my_node    ! send to node 3, 2 and 1
        DO
                send a dest c

                DEC dest

        END0
        ENDWHILE
   ENDIF

!c MA STOP send_macro



! ------------------------  RECEIVE SESSION --------------------


!c MA START rec_macro

        IF my_node GT zero
                IF my_node LT max        ! nodes 1, 2 and 3
                                         ! receive one message
                        rec_type a g c

                ENDIF
        ENDIF

!c MA STOP rec_macro


HALT



! ----------------------- DEFINITIONS OF USER'S MACROS ----------


! send_macro records time when sending messages:
!         BEF_SEND - indicates start of a send process
!         AFT_SEND - indicates an end of a send process


!M send_macro
!M
!M before_instr:
!M         instr send:
```

```
!M                output BEF_SEND my_node
!M after_instr:
!M         instr send:
!M                output AFT_SEND my_node
!M
!M end



! rec_macro records time when receiving messages,
! and their latencies:
!         BEF_REC     - time at which a node is ready to receive
!                       a message
!         AFT_REC     - indicates an end of a receive process
!         LATENCY_REC - latency of a received message,
!                       and time when it arrived


!M rec_macro
!M
!M before_instr: instr rec_type:
!M         output BEF_REC my_node
!M after_instr:
!M         instr rec_type:
!M                output AFT_REC my_node
!M                output LATENCY_REC rec_lat
!M
!M end
```

Example 3.

dumb_ex3.cpl

```
c CONFIG_DT 0 0x00000220
c CONFIG_DT 1 0x00002090
c CONFIG_DT 2 0x00010448
c CONFIG_DT 3 0x1bfa2949
c CONFIG_DT 4 0x2df02948
c LINK_NODE 1
c LINK_ROUTER 1
c CFG_WORM_CH 4 4 4
c CFG_NET 2 4 2
c OS_OVHD 1


! ************************* example 3 ************************** !
!                                                                 !
! This file is used to generate data for tables V and VI.         !
! Node 0 sends a message to nodes 1 and 4 in 4x4 mesh network.    !
!                                                                 !
! ************************* dumb_ex3.cpl ********************** !


        define dest1 d
        define dest2 e
        LOAD a 1                ! message type
        LOAD c @MSG             ! message length, passed from GUI


! -------------------------  MULTICAST SESSION ------------------

        LOAD dest1 1                ! node 1
        LOAD dest2 4                ! node 4


!c MA START send_macro

   IF my_node EQ zero                ! only node 0 sends messages

                send a dest1 c       ! send to node 1
                send a dest2 c       ! send to node 4

   ENDIF

!c MA STOP send_macro
```

```
! ------------------------  RECEIVE SESSION --------------------


!c MA START rec_macro

        IF my_node EQ dest1         ! node 1 receives a message

                        rec_type a g c

        ENDIF

        IF my_node EQ dest2         ! node 2 receives a message

                rec_type a g c

        ENDIF

!c MA STOP rec_macro


HALT



! ----------------------- DEFINITIONS OF USER'S MACROS ----------


! send_macro records time when sending messages:
!         BEF_SEND - indicates start of a send process
!         AFT_SEND - indicates an end of a send process


!M send_macro
!M
!M before_instr:
!M          instr send:
!M                  output BEF_SEND my_node
!M after_instr:
!M          instr send:
!M                  output AFT_SEND my_node
!M
!M end



! rec_macro records time when receiving messages,
! and their latencies:
!         BEF_REC     - time at which a node is ready to receive
!                       a message
!         AFT_REC     - indicates an end of a receive process
```

```
!         LATENCY_REC - latency of a received message,
!                       and time when it arrived


!M rec_macro
!M
!M before_instr: instr rec_type:
!M         output BEF_REC my_node
!M after_instr:
!M         instr rec_type:
!M                 output AFT_REC my_node
!M                 output LATENCY_REC rec_lat
!M
!M end
```

Example 4.

dumb_ex4.cpl

```
c CONFIG_DT 0 0x00000220
c CONFIG_DT 1 0x00002090
c CONFIG_DT 2 0x00010448
c CONFIG_DT 3 0x1bfa2949
c CONFIG_DT 4 0x2df02948
c LINK_NODE 1
c LINK_ROUTER 1
c CFG_WORM_CH 4 4 4
c CFG_NET 2 4 1
c OS_OVHD 1


! ************************ example 4 ************************** !
!                                                               !
! This file is used to generate data for tables VII and VIII.   !
! Nodes 1, 4, 6 and 9 send a message to node 0                  !
! in 4x4 mesh network.                                          !
!                                                               !
! ************************ dumb_ex4.cpl ********************** !

        define S_F Q
        define source1 z
        define source2 y
        define source3 x
        define source4 u
        define dest d
```

```
        LOAD a 1
        LOAD c @MSG

        LOAD source1 4
        LOAD source2 1
        LOAD source3 6
        LOAD source4 9

        LOAD dest 5
        LOAD S_F 0

! ------------------------  MULTICAST SESSION ------------------

        IF my_node EQ source1          ! S_F flag indicates a send
                LOAD S_F 1             ! action
        ENDIF

        IF my_node EQ source2
                LOAD S_F 1
        ENDIF

        IF my_node EQ source3
                LOAD S_F 1
        ENDIF

        IF my_node EQ source4
                LOAD S_F 1
        ENDIF

        IF S_F EQ a                    ! nodes 1, 4, 6 and 9 send
                send a dest c          ! a message
        ENDIF



! ------------------------  RECEIVE SESSION -------------------

        LOAD f 4                       ! counter of messages

!c MA START rec_macro

        IF my_node EQ dest             ! node 5 receives messages
                WHILE f NE zero
                DO
                        rec_type a g c
                        DEC f
                ENDDO
                ENDWHILE
        ENDIF
```

```
!c MA STOP rec_macro

        halt



! ---------------------- DEFINITIONS OF USER'S MACROS ----------


! rec_macro records time when receiving messages,
! and their latencies:
!         BEF_REC     - time at which a node is ready to receive
!                       a message
!         AFT_REC     - indicates an end of a receive process
!         LATENCY_REC - latency of a received message,
!                       and time when it arrived


!M rec_macro
!M
!M before_instr: instr rec_type:
!M          output BEF_REC my_node
!M after_instr:
!M          instr rec_type:
!M                  output AFT_REC my_node
!M                  output LATENCY_REC rec_lat
!M
!M end
```

one_broadcast.cpl

```
c CONFIG_DT 0 0x00000220
c CONFIG_DT 1 0x00002090
c CONFIG_DT 2 0x00010448
c CONFIG_DT 3 0x1bfa2949
c CONFIG_DT 4 0x2df02948
c LINK_NODE 1
c LINK_ROUTER 1
c CFG_WORM_CH 4 4 4
c CFG_NET 2 2 2
c OS_OVHD 1

! ************************************************************ !
!                                                              !
! This file is used to generate data for figures 35 and 36.    !
! Node 0 sends one broadcast message to all other nodes        !
! in 4x4 mesh network.                                         !
!                                                              !
! ************************* one_broadcast.cpl ***************** !


        define dest e

        LOAD a 1                ! message type
        LOAD c @MSG             ! message length, passed from GUI
        LOAD d 256              ! wait time



! ------------------------  BROADCAST SESSION ------------------


        LOAD dest 0

!c MA start send_macro

   IF my_node EQ  zero                  ! only node 0 sends messages

        WHILE dest NE nbr_nodes         ! send to all nodes
        DO
                IF dest NE my_node
                        send a dest c

                ENDIF

                INC dest

        END0
```

```
        ENDWHILE

   ENDIF

!c MA stop send_macro



! ------------------------  RECEIVE SESSION --------------------

!c MA start rec_macro

        IF my_node NE zero              ! all nodes except node 0
                rec_type a g c          ! receive a message
        ENDIF

!c MA stop rec_macro



        halt



! ----------------------- DEFINITIONS OF USER'S MACROS ----------


! send_macro records time when sending messages:
!         BEF_SEND - indicates start of a send process
!         AFT_SEND - indicates an end of a send process


!M send_macro
!M
!M before_instr:
!M         instr send:
!M                 output BEF_SEND my_node
!M after_instr:
!M         instr send:
!M                 output AFT_SEND my_node
!M
!M end



! rec_macro records time when receiving messages,
! and their latencies:
!         BEF_REC     - time at which a node is ready to receive
!                       a message
!         AFT_REC     - indicates an end of a receive process
```

```
!         LATENCY_REC - latency of a received message,
!                       and time when it arrived


!M rec_macro
!M
!M before_instr: instr rec_type:
!M         output BEF_REC my_node
!M after_instr:
!M         instr rec_type:
!M                 output AFT_REC my_node
!M                 output LATENCY_REC rec_lat
!M
!M end
```

n-body.cpl

```
c CONFIG_DT 0 0x00000220
c CONFIG_DT 1 0x00002090
c CONFIG_DT 2 0x00010448
c CONFIG_DT 3 0x1bfa2949
c CONFIG_DT 4 0x2df02948
c LINK_NODE 1
c LINK_ROUTER 1
c CFG_WORM_CH 4 4 4
c CFG_NET 2 2 2
c OS_OVHD 1

! ************************* N-BODY PROBLEM ******************** !
!                                                               !
! Each node sends a broadcast message and then receives         !
! messages from other nodes.                                    !
! The process of sending and receiving messages is repeated     !
! in a loop.                                                    !
!                                                               !
! ************************* n-body.cpl ************************ !


        LOAD a 1            ! message type, counter of iterations
        LOAD b 10           ! number of iterations
        LOAD c @MSG         ! message length, passed from GUI
        LOAD d 256          ! wait time


WHILE a LE b                ! loop control
DO
```

```
! ------------------------ BROADCAST SESSION -------------------

        LOAD e 0

!c MA start send_macro

        WHILE e NE nbr_nodes
        DO
                IF e NE my_node              ! send to all nodes
                        send a e c           ! except the source

                ENDIF

                INC e

        END0
        ENDWHILE

!c MA stop send_macro

! ------------------------ RECEIVE SESSION ---------------------

        LOAD e 0
        MOVE f nbr_nodes
        DEC f

!c MA start rec_macro

        WHILE e NE f
        DO
                rec_type a g m              ! receive from all nodes
                                            ! except the destination
                INC e
        ENDDO
        ENDWHILE

!c MA stop rec_macro

!  WAIT
        wait d          ! wait statement models computation time


! ------------------------ END OF ITERATION --------------------

        INC a

ENDDO
ENDWHILE


        halt
```

```
! ---------------------- DEFINITIONS OF USER'S MACROS ----------


! send_macro records time when sending messages:
!         BEF_SEND - indicates start of a send process
!         AFT_SEND - indicates an end of a send process


!M send_macro
!M
!M before_instr:
!M          instr send:
!M                  output BEF_SEND my_node
!M after_instr:
!M          instr send:
!M                  output AFT_SEND my_node
!M
!M end



! rec_macro records time when receiving messages,
! and their latencies:
!         BEF_REC     - time at which a node is ready to receive
!                       a message
!         AFT_REC     - indicates an end of a receive process
!         LATENCY_REC - latency of a received message,
!                       and time when it arrived


!M rec_macro
!M
!M before_instr: instr rec_type:
!M          output BEF_REC my_node
!M after_instr:
!M          instr rec_type:
!M                  output AFT_REC my_node
!M                  output LATENCY_REC rec_lat
!M
!M end
```

n-body.pre

```
c CONFIG_DT 0 0x00000220
c CONFIG_DT 1 0x00002090
c CONFIG_DT 2 0x00010448
c CONFIG_DT 3 0x1bfa2949
c CONFIG_DT 4 0x2df02948
c LINK_NODE 1
c LINK_ROUTER 1
c CFG_WORM_CH 4 4 4
c CFG_NET 2 2 2
c OS_OVHD 1

DEFINE counter0 Z
LOAD counter0 0
DEFINE accum0 Y
LOAD accum0 0



! DU START DEFAULT_EX_TIME
OUT B0 my_node timer

! AC START DEFAULT_LATENCY REC_LAT

! CO START DEFAULT_REC_MSG ALL_REC_INSTR


! ************************* N-BODY PROBLEM ******************** !
!                                                               !
! Each node sends a broadcast message and then receives         !
! messages from other nodes.                                    !
! The process of sending and receiving messages is repeated     !
! in a loop.                                                    !
! Performance monitoring code is inserted into the program.     !
!                                                               !
! ************************* n-body.pre ************************ !



        LOAD a 1            ! message type, counter of iterations
        LOAD b 10           ! number of iterations
        LOAD c @MSG         ! message length, passed from GUI
        LOAD d 256          ! wait time


WHILE a LE b                ! loop control
DO

! ------------------------ BROADCAST SESSION -------------------

        LOAD e 0
```

```
! MA START send_macro

        WHILE e NE nbr_nodes
        DO
                IF e NE my_node              ! send to all nodes
OUT U3 my_node my_node
OUT S3 my_node TIMER
                        send a e c           ! except the source
OUT U4 my_node my_node
OUT S4 my_node TIMER

                ENDIF

                INC e

        END0
        ENDWHILE

! MA STOP send_macro


! ------------------------ RECEIVE SESSION ---------------------

        LOAD e 0
        MOVE f nbr_nodes
        DEC f


! MA START rec_macro

        WHILE e NE f
        DO
OUT U0 my_node my_node
OUT S0 my_node TIMER
INC counter0
                rec_type a g m             ! receive from all nodes
OUT U1 my_node my_node
OUT S1 my_node TIMER
OUT U2 my_node rec_lat
OUT S2 my_node TIMER
ADD accum0 accum0 REC_LAT
                                           ! except the destination
                INC e
        ENDDO
        ENDWHILE

! MA STOP rec_macro


!  WAIT
        wait d          ! wait statement models computation time
```

```
! ------------------------ END OF ITERATION ---------------------

        INC a

ENDDO
ENDWHILE



! CO STOP DEFAULT_REC_MSG ALL_REC_INSTR

! AC STOP DEFAULT_LATENCY REC_LAT

OUT E0 my_node timer
! DU STOP DEFAULT_EX_TIME



IF counter0 GT zero
        DIV fa accum0 counter0
        OUT A0 my_node fa
ENDIF

HALT



! ----------------------- DEFINITIONS OF USER'S MACROS ----------


! send_macro records time when sending messages:
!         BEF_SEND - indicates start of a send process
!         AFT_SEND - indicates an end of a send process


!M send_macro
!M
!M before_instr:
!M          instr send:
!M                  output BEF_SEND my_node
!M after_instr:
!M          instr send:
!M                  output AFT_SEND my_node
!M
!M end



! rec_macro records time when receiving messages,
! and their latencies:
!         BEF_REC     - time at which a node is ready to receive
```

```
!                          a message
!          AFT_REC      - indicates an end of a receive process
!          LATENCY_REC  - latency of a received message,
!                          and time when it arrived


!M rec_macro
!M
!M before_instr: instr rec_type:
!M          output BEF_REC my_node
!M after_instr:
!M          instr rec_type:
!M                   output AFT_REC my_node
!M                   output LATENCY_REC rec_lat
!M
!M end
```