PERFORMANCE PREDICTION OF MESSAGE PASSING COMMUNICATION IN DISTRIBUTED MEMORY SYSTEMS

A Thesis

Presented to

the Faculty of the Department of Electrical and Computer Engineering

University of Houston

In Partial Fulfillment

of the Requirements for the Degree of

Master of Science

in Computer and Systems Engineering

by

Aparna Mande

May, 2002

Abstract

Communication is an important component in determining the overall performance of a distributed memory parallel computing application. It therefore becomes essential to predict communication performance of applications on the underlying network hardware of a target distributed memory system. This thesis concentrates on integrating a cycle driven k-ary n-cube network simulator to the existing CAL-SIM distributed memory simulator for evaluating communication performance of message passing applications. The design and implementation of a suitable network interface required for the integration is presented. With detailed network simulation the accuracy of predictions made is very high. The impact on communication performance by varying some of the network design parameters is studied. Other important aspect of this work is the access to an evaluation platform for evaluating network design tradeoffs, using real applications as workload instead of synthetic workloads.

Contents

CHAPTER 1 - INTRODUCTION 3

1.1 BACKGROUND 2
1.2 PROBLEM ADDRESSED 3

CHAPTER 2 - RELATED WORK 5

2.1 MESSAGE PASSING INTERFACE 5
2.2 EVALUATION TOOLS 7
2.2.1 Simulation of Computation Performance of the Application 8
2.2.2 Simulation of Communication Performance of the Application 9

CHAPTER 3 - DISTRIBUTED MEMORY SIMULATOR COMPONENTS 11

3.1 CAL-SIM 12
3.2 NETSIM 18
3.2.1 Design Space of Router 20
3.2.2 Basic Parameter Settings 26

CHAPTER 4 - SYSTEM INTEGRATION 28

4.1 INTERFACE DESIGN AND IMPLEMENTATION 28
4.1.1 Send Side 29
4.1.2 Receive Side 31
4.2 NETWORK EVENT 31
4.3 SCHEDULING THE NETWORK EVENT 32
4.4 REDUCING SIMULATION OVERHEAD 36
4.5 DESIGN GUIDELINES FOR THE NETWORK MODEL 37

CHAPTER 5 - SIMULATION RESULTS 39

5.1 SIMULATOR PARAMETER SETTINGS 39
5.1.1 Linear Delay Model Parameter Settings 40
5.1.2 Network Mode Parameter Settings 40
5.2 RESULTS FOR THE NAS MULTIGRID BENCHMARK 41
5.3 RESULTS FOR THE NAS INTEGER SORT BENCHMARK 45
5.4 RESULTS FOR THE NAS EMBARRASSINGLY PARALLEL BENCHMARK 49
5.5 RESULTS FOR THE FFT BENCHMARK 50
5.7 FUNCTIONAL VERIFICATION OF THE SIMULATOR 52

CHAPTER 6 - CONCLUSION AND FUTURE WORK 55

6.1 CONCLUSION 55
6.2 RECOMMENDATIONS FOR FUTURE WORK 56

REFERENCES 58

List of Figures

Figure 1. Steps in profiling a parallel program ... 13
Figure 2. Process execution cycle in execution driven simulation ... 15
Figure 3. Network delay model parameters ... 17
Figure 4. Canonical model for the router switch used in NetSim ... 20
Figure 5. (a) Two Dimensional Torus Network (b) Two Dimensional Mesh ... 21
Figure 6. Crossbar Switch Architecture (a) N x M crossbar (b) Cascaded crossbar ... 24
Figure 7. Simulation system design ... 29
Figure 8. Scheduling the network event with look-ahead ... 33
Figure 9. Code format for network event ... 37
Figure 10. Avg. message size and total number of messages as a function of machine size ... 42
Figure 11. Predicted average message latency for the MG Class A benchmark ... 43
Figure 12. Comparison of routing algorithm performance ... 43
Figure 13. Percent computation and communication in MG benchmark ... 44
Figure 14. Predicted performance for the Multigrid Class A benchmark ... 45
Figure 15. Avg. message size and total number of messages as a function of machine size for ... 46
the IS Class W Benchmark ... 46
Figure 16. Avg. message latency prediction for the IS Class W benchmark ... 47
Figure 17. Avg. message latency with a variation in routing algorithm ... 47
for the IS benchmark ... 47
Figure 18. Avg. message latency on a 2D mesh for the IS Class W benchmark ... 48
Figure 19. Percent computation and communication in the IS Class W benchmark ... 48
Figure 20. Predicted performance for the IS Class W benchmark ... 49
Figure 21. Average message latency for the EP Class A benchmark ... 50
Figure 22. Predicted performance for the EP Class A benchmark ... 50
Figure 23. Communication time for FFT benchmark ... 51
Figure 24. Performance of the FFT benchmark ... 51
Figure 25. Prediction error for communication performance prediction using ... 52
the linear delay model ... 52
Figure 26. Execution flow for a cluster architecture in simulation ... 57

List of Tables

Table 1. Network parameter settings 41
Table 2. Benchmark Tests.. 53
Table 3. Functional Verification 54

Chapter 1 - Introduction

1.1 Background

Parallel computing systems are widely accepted as an effective technology for solving large, computationally intensive problems in the area of high performance computing. Based on their memory organization, parallel computing systems fall into two categories: shared memory systems and distributed memory systems. Of these, the distributed memory architecture is more scalable and thus preferred for large-scale machines.

The distributed memory system also known as the multicomputer, consists of commodity processors joined by a suitable interconnection network. Inter-processor communication proceeds by exchanging messages through the network. Since processor speed is increasing rapidly, the communication network becomes a limiting factor in the performance of message-passing applications. Using commodity network components to connect the processors results in immense slowdown of the application execution speed. Therefore the design of a high-speed interconnection network becomes critical in distributed memory architectures.

The cost and engineering effort required in building a high performance distributed memory system makes it important to predict the performance of the proposed architectural features. Performance prediction tools provide a low cost means for predicting application performance on proposed systems. The information made available by such a tool assists designers in optimizing their system for the highest possible performance. The evaluation tool models several hardware and software characteristics of a system. By varying these characteristics, the system performance can be predicted for its intended class of applications.

1.2 Problem Addressed

The need was to provide an accurate and efficient tool for evaluating the performance of distributed memory systems under realistic workloads. For most message-passing applications, their communication performance is extremely crucial for achieving higher performance. Variation in communication network can have a significant impact on the application performance. It therefore becomes important that the tool facilitate detailed evaluation of the communication network along with the evaluation of other system characteristics. Thus the pseudo execution environment provided for applications should be very much like the specified hardware, delivering accurate results to the users of the system.

A distributed memory simulator (CAL-SIM) was previously developed as part of our research [1]. The architectural parameter that can be varied within the simulator is the number of processors. The simulator makes simplified assumptions of the network when predicting the communication performance of message passing applications.

An interconnection network simulator (NetSim) for distributed memory systems has also been independently developed [2]. The simulator allows for modeling of various network designs. Like most other network simulators, it makes use of synthetic workloads to generate traffic in the network.

The task we address in this thesis is the design of a suitable network interface within CAL-SIM for integrating the network simulator. The integration of the two evaluation tools, CAL-SIM and NetSim provides the capability to specify an interconnection structure for processors within the CAL-SIM simulation system. It provides with a realistic environment for program execution and more accurate performance measurements. It also allows for evaluating the impact of network design on communication performance of applications and in making refinements to the network model, to suite the application domains requirements.

The other important aspect of this integration work is the access to an evaluation platform for evaluating candidate interconnection networks using real application workload. Most of the existing interconnection network simulators make use of synthetic workloads for performance evaluation. Firstly, these synthetic (dummy) workloads have to be generated by examining the communication behavior of real applications themselves. Secondly, they may not necessarily capture the communication behavior very accurately and often make simplifying assumptions about the workload characteristics, which may be inappropriate and may lead to inaccurate performance predictions. So the best case would be to have real world applications to test the network performance instead of the synthetic workloads.

Chapter 2 - Related Work

Simulation is a very effective technique for the evaluation of parallel systems before incurring the hardware cost. Various simulation techniques have been developed for evaluating the performance of parallel systems as well as for their interconnection networks. In this chapter, we describe some of the major simulation techniques and indicate which technique best matches our requirements. The message passing interface standard adopted by the benchmark applications used in this research is also discussed in brief.

2.1 Message Passing Interface

Message Passing Interface (MPI) [3], has become an accepted standard in implementing communication functions for message based parallel programs in distributed memory systems. The major goal of this standard is to provide for portable and easy to use communication library functions without affecting performance. Amongst the many programming features, the standard provides several mechanisms to perform point-to-point and collective communications.

With point-to-point communication mechanism, communication proceeds between a pair of processes. The basic point-to-point operations are the send and receive operations. The communication semantics for these operations can be either blocking or non-blocking. With the blocking type, the send function call does not return until resources such as the user buffer can be safely reused or the data transfer to the network interface has been completed. But with the non-blocking semantics, the send operation can return right after the communication has been initiated and does not wait to see if the buffer is safe to reuse. The non-blocking semantics is provided for performance reason so that the communication can be overlapped with computation. The sender however has to later issue a send complete call to verify if the data transfer has been completed. With either of the semantics, the operations use one of the following communication modes: standard, buffered, synchronous and ready.

In the standard mode, the MPI implementation can choose to wait for a matching receive to be posted before starting data transfer or can choose to buffer the message in a temporary system buffer and return immediately. In the buffered mode, an outgoing message always gets buffered and the operation will complete whether or not a matching receive has been posted. With the synchronous mode, the MPI implementation ensures that the receiver has started to receive the message and that the send buffer can be safely reused. In this mode, a send can start immediately but can complete only if a matching receive has been posted. In the ready mode, the send call proceeds only if the matching receive is already posted.

The other communication mechanism provided is for collective operations. In a collective operation, a group of processes participate in the communication. Typical collective operations are the barrier synchronization, broadcast, gather, scatter, reduce and all to all exchange. In barrier synchronization, each process is blocked until all other processes in the group have executed the barrier call. A broadcast message involves the root process sending a message to all processes in the group including itself. A gather operation is the reverse of broadcast. The root process now waits to receive a message from every process in the group. In a scatter operation, the message sent by the root process is split into n equal parts and each of the respective parts is sent to the n processes in the group. In a reduce operation each process combines the elements provided in its input buffer using a common specified operation and returns the result for the combined elements to the root process. With an all to all exchange, each process sends distinct data to each of the other processes.

Apart from the point-to-point and collective operations, the MPI standard includes several more features such as process groups, process topologies, communication contexts and derived data types. In this thesis the implementations of only point-to-point and collective communications, which require the use of communication network are considered. These are also the most frequently used MPI functions and are supported by the MPI library within our simulation system.

2.2 Evaluation Tools

Message passing applications include two important components namely, computation and communication. The time spent by the application in computation and in communication needs to be simulated for accurate predictions. The simulation techniques for predicting both the components are discussed below.

2.2.1 Simulation of Computation Performance of the Application

Some of the available techniques for simulating this component of application performance are trace driven simulation, instruction level simulation and execution driven simulation.

Trace Driven Simulation: In this method, a program is instrumented to generate a trace of its execution events, which need to be simulated. The trace can then be used for simulation of the target machine. The method can be accurate while studying cache/memory behavior or study application performance on a uniprocessor system. But trace driven simulation can prove very difficult while studying multiprocessor systems. The execution being multithreaded, the generation of a representative trace is a problem. This method is thus rarely used for simulation of parallel systems.

Instruction level Simulation: Unlike trace driven simulation, instruction level simulation does not involve collecting a trace. This simulation model takes in each instruction of the application program and emulates the behavior of the corresponding instruction for the target architecture. Due to emulation of each target instruction, the number of simulator instructions executed per host instruction is usually greater than hundred. This results in significant slowdown of the simulator operation although the accuracy of prediction is very high for such a technique.

Execution Driven Simulation: This technique is relatively new and most commonly used today. With this technique the execution of the program and the simulation model for the architecture are interleaved. The assembly language code for the application is parsed for basic blocks and the timing information is inserted only at the start of each basic block. Unlike instruction level simulation that emulates every single instruction, this technique executes an entire basic block instead. So there is significant reduction of simulation overhead in execution driven simulation as compared to instruction level simulation, thereby making it the preferred technique.

The distributed memory simulator, CAL-SIM used in this research utilizes the execution driven simulation technique to predict the compute performance of applications. It enables efficient simulation while providing accurate performance predictions.

2.2.2 Simulation of Communication Performance of the Application

For message passing architectures, the communication performance of applications can be determined from the time spent in communication routines as well as in the network hardware. The latency for a message to reach its destination eventually adds up to the execution time of the application, if the receiver of the message has been waiting on it. The major components of message latency are the messaging layer latency, which involves preparing the message for data transmission and the network hardware latency. Although the overhead of messaging layer is significant, for long messages, the network hardware latency dominates the communication latency. So in this research the latency introduced by the networking hardware is considered and the following simulation techniques are discussed with respect to the network hardware.

Analytical Model: Analytical models provide a quick estimate of the message latency from some of the network parameters. However these models lack sufficient accuracy for complex networks that incorporate several design tradeoffs.

Cycle Driven Simulation: This is a commonly adopted technique for evaluating networks because of the number of details that can be incorporated in the simulation. With this approach, the network is made to run as if there were a clock signal driving it and the activities advance every clock cycle, in a way these would proceed in a real network. Accurate results can be obtained as the simulation can model hot spots and contentions in the network, which add to the message latency. However detailed cycle driven simulation tends to slowdown the simulation process.

Event Driven Simulation: With event driven simulation a queue of events maintained in a time order drives the simulation. Simulation time gets updated to the timestamp of an event when it is selected for running from the head of the queue. The technique allows for accurate simulation. However when there are a large number of network parameters, which manipulate the event queue every simulation cycle, the simulation tends to be extremely slow. Cycle driven simulation is preferred in this case.

The network simulator, NetSim used in this research, utilizes the cycle driven simulation technique. The cycle-by-cycle network model is also appropriate for integration with the execution driven distributed memory simulation system, as it provides easier control over network run time in simulation.

Chapter 3 - Distributed Memory Simulator Components

This chapter describes the tools that make up the distributed memory simulation system. The simulation system is designed to run on a uniprocessor host machine.

A parallel architecture is simulated on a uniprocessor machine by creating threads to represent the different processors in the architecture model. Each thread holds a copy of the application program and executes its part of the code. The threads communicate by sending messages to each other during their lifetime. The application program itself can be made transparent to the details of setting up communication and having the messages sent or received by simply making high level function calls, with these functions implemented as part of the system software library. The desired feature would be to use the MPI standard communication calls within the application. This means a run time library implementing the MPI communication routines is required of the simulation tool. CAL-SIM supports simulation of MPI based parallel programs and is described in section 3.1. More details of the simulator can be found in [1].

To simulate the communication behavior of the parallel application, a network simulator configured with the distributed memory simulator is also required. NetSim, a cycle level network simulator is suitable for integration with CAL-SIM and is described in section 3.2. More information on this tool can be found in [2].

3.1 CAL-SIM

CAL-SIM is an execution driven, distributed memory simulator running message passing applications. The tool accepts an MPI application and predicts its execution time on the target architecture in terms of number of cycles. Simulation is carried out on a uniprocessor host while the simulator itself provides for the multithread framework. The CAL-SIM simulator library is made up of several components such as the simulation core, basic network model, MPI library, application profiling tools, timing analyzer etc.

With the execution driven technique, the execution of the application is interleaved with the simulation process. Execution driven technique is presented in [4], [5]. The profiling tool within the simulator parses the assembly level code of the application, identifying basic blocks and inserts timing information code for each basic block. The instrumented code is then compiled and linked with the simulation library to provide the executable. This technique is depicted in Figure 1.

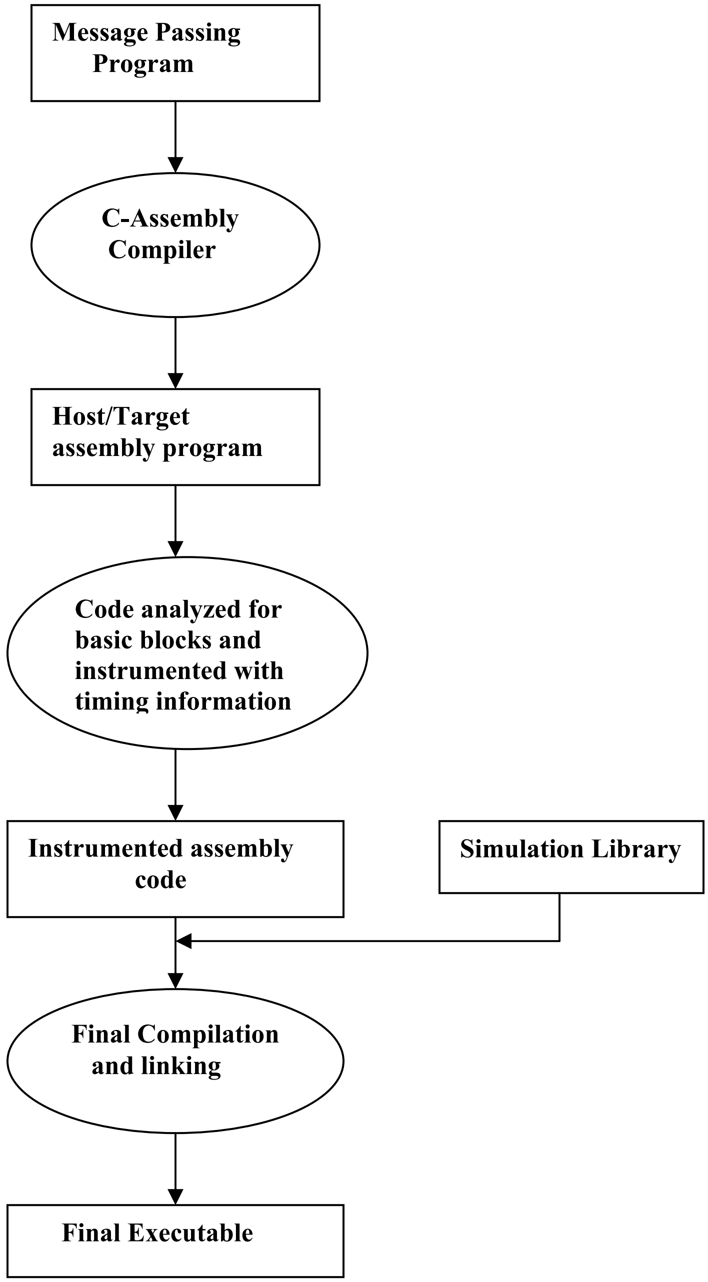

Figure 1. Steps in profiling a parallel program

The instrumented code now has the instructions for updating the time counter executed as the application is being executed. The simulation core begins execution by creating threads equal to the number of nodes being simulated and each thread now holds a copy of the profiled application. With multiple threads executing, the access to global data structures within the simulator such as the time counter needs to be maintained in the correct sequence. The threads would require synchronizing every time there is a need to update one of the global variables. With a large amount of synchronization activity, the slowdown in the simulation speed is significant. Therefore to avoid the high overhead of synchronization, global variables are maintained as one-dimensional array per variable and each thread uses its own copy of the variable indexed by the processor number or thread number. The time counter for example will now be maintained as time counter[NPROCS] and the instructions for updating the processor thread's time will update the array element of the time counter, which is indexed by the thread number. Synchronization routines are used only for access to a processor's message queue.

Once the threads get created and initialized, the simulator core schedules these threads for execution. Note that the execution of threads is serialized on a uniprocessor host. A context switch occurs when the currently running thread performs a communication call. A new thread is then picked up for execution by the simulator core and this process continues until all the threads have executed. The simulation time gets updated to the schedule time of various activities, which are maintained in time order as in event driven simulation. This type of scheduling which is part of the execution driven technique is depicted in Figure 2.

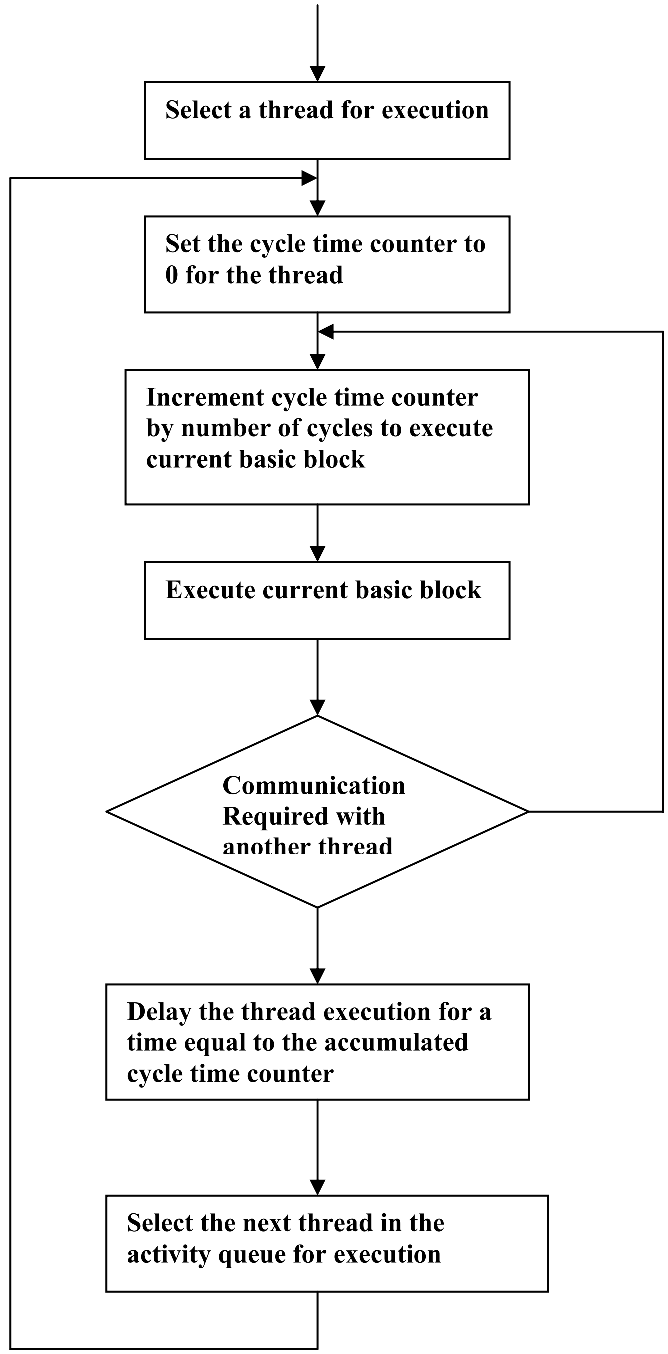

Figure 2. Process execution cycle in execution driven simulation

The above scheduling technique ensures that communication schedules within the processor thread occur at the correct cycle of execution. The communication calls require for a thread to either exchange data with a process running on another processor (another thread in simulation) or participate in a collective operation by all threads on the global data or participate in a synchronization activity. The MPI library implementation for the communication calls is responsible for the setup to have messages transferred to their destination. Message queues are maintained to hold the outgoing and incoming messages for a thread. In a simulated send operation, a message is time stamped with the send time, which is the current time of simulation and with the receive time which is the send time plus the predicted message latency. An event which places the message in the destination thread's message queue is then scheduled to occur at the receive time stamp. In a simulated receive operation the destination thread simply checks its message queue to see if the message has arrived.

The time at which a destination thread receives its message is computed using simple delay functions. The network hardware latency is computed using a linear message latency model, where the latency is proportional to the length of the message and to the distance between the source and destination nodes. This is essentially the ideal latency experienced by a message in the network to get transferred to the destination processor. In addition to the network latency, delay models for overhead at the sender side (e.g., creation of message packets) and at the receiver side (e.g., transfer of message from the network to receiver memory) are provided. So the total message latency is the sum of the three basic parameters shown in figure 3. An event is scheduled at the message arrival time which puts the message into the destination thread's receive queue and signals a receive semaphore. The destination thread executing its receive call, checks the receive queue for its intended message. If the message has arrived, it copies the message in the user level buffer specified by the receive operation and returns successfully. If the message has not yet arrived, then the semantics of the receive operation determines further action. If the receive is non-blocking, the receive call simply returns. The thread has to execute a wait routine at a later point in execution to check if the request had been processed and it will now wait for the message to arrive in the wait routine. If the receive is blocking then the receiving thread blocks itself on the receive semaphore. On a semaphore signal, the receiver checks the arrived message. If this message is the expected one, the receiver copies it to the user buffer else puts it in a checked message queue and again waits on the receive semaphore for the next signal.

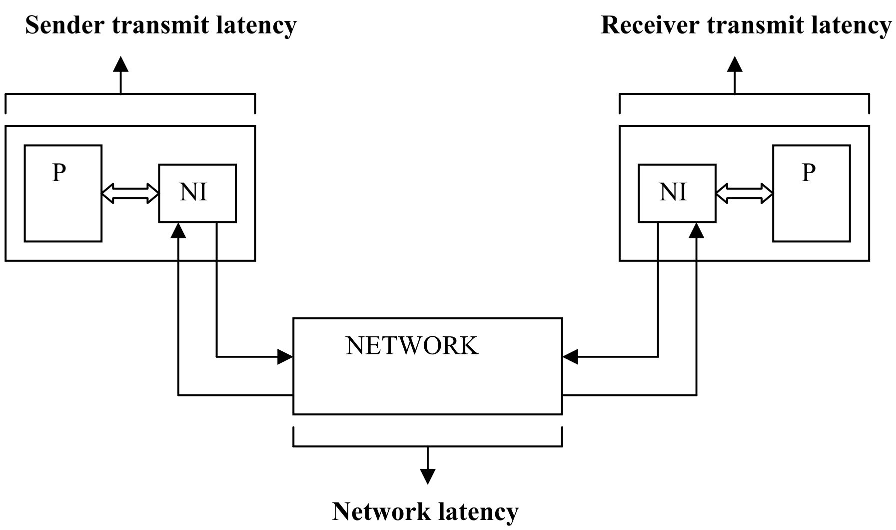

Figure 3. Network delay model parameters

This mode of the simulator that does not support simulation of network activity has been provided if the user of the simulator is interested in predicting application runtime without worrying about the details of the network, requiring knowledge only about the behavior of parallel application and its compute efficiency. The use of simple delay equations in such a case enables faster simulation. This mode within the simulator is however flexible enough to let users of the system use their own delay models for the network parameters of figure 3. If a user does not provide the delay models, then the default models in the system will be used.

A separate mode of operation is required within the simulator for situations in which a parallel application needs to be evaluated for its communication performance. The accuracy of event timing for the send and receive operations becomes very important in this case. The message arrival time obtained from simulation of network activities needs to be used rather than the time predicted by a simple latency model. This is because simple latency expressions cannot account for factors such as network contention, traffic patterns and complex design of messaging algorithms, which can have a significant impact on the message latency. A network simulator thus needs to be connected with CAL-SIM in this mode. The first task now would be to have a suitable interconnection network simulator. NetSim described in section 3.2 is used for the purpose. The next task would be to develop a network interface in CAL-SIM, to have the tools work together and provide for a comprehensive distributed memory simulation system.

3.2 NetSim

For integrating a network simulator with CAL-SIM, a choice has to be made as to which would be the appropriate model: event driven or cycle driven. CAL-SIM being an event driven simulator, having another event driven simulation combined with it would be rather complicated and make it very difficult to ensure that the messages are sent into the network at the correct simulation time. Also the time when the network needs to halt and let other threads continue would be difficult to control. With cycle driven simulation it would be much easier to control the number of cycles to run the network simulator. Moreover the network can be scheduled as an independent event within the event driven simulator. The use of a cycle driven network simulator for integration with an event driven simulator is thus the preferred choice. NetSim [2] is a cycle driven network simulator that models the behavior of multi-computer networks and switches. This network simulator has been used for integration with CAL-SIM and is described in this section.

The main component within NetSim is a router, which handles message communication for the node. A router is connected directly to the routers of the neighboring nodes in a direct network. The router is designed using the canonical model shown in figure 4.

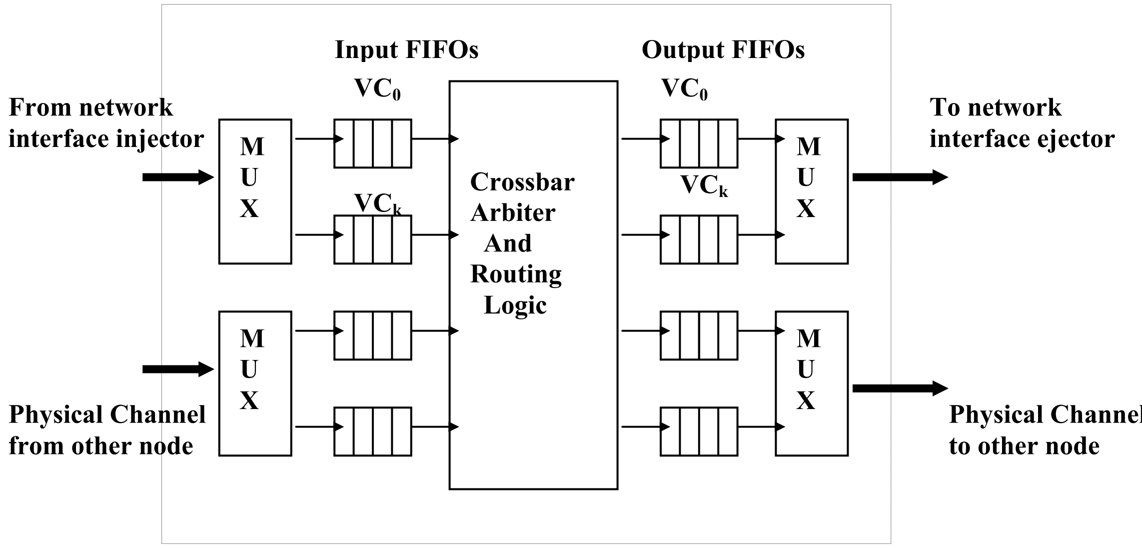

Figure 4. Canonical model for the router switch used in NetSim

With the router design model of figure 4, the flexibility to vary network design parameters is large. The model also provides a common environment for a fair evaluation of the design tradeoffs. The design space of this canonical router in NetSim is discussed below.

3.2.1 Design Space of Router

- **Network topology**

As with most direct networks, k-ary n-cube mesh and torus networks are modeled. For the n-dimensional mesh, the number of nodes k (or the radix) in each dimension is the same, with each node having n to 2n neighbors based on the nodes location in the mesh. With a torus network, variable number of nodes k (variable radix) for each dimension is possible. In a bi-directional torus, all nodes have the same number of neighbors, 2n because of the wraparound channels. For simplicity the simulator models equal number of nodes in each dimension or the radix k is the same for each dimension in a bi-directional torus.

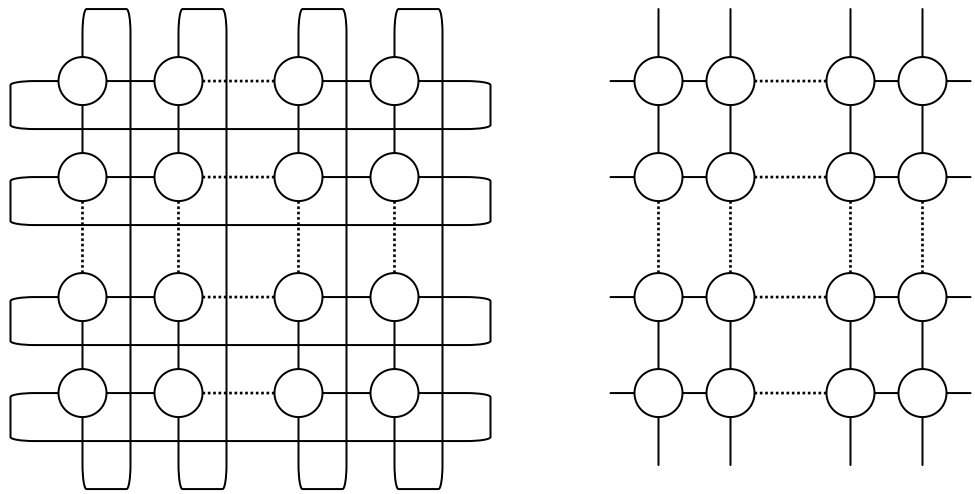

Figure 5. (a) Two Dimensional Torus Network (b) Two Dimensional Mesh

- **Network Size**

With a k-ary n-cube network configuration having k nodes in each of the n dimensions, the size of the network becomes k ^ n. Varying network size in simulation can enable finding the scalability of the proposed network for a large network size.

- **Switching Technique**

The switching technique determines the path establishment between the source and destination nodes and also how a packet uses the buffer resources. Store and forward packet switching was traditionally used in which each individual packet is routed from source to destination. This required buffering the entire packet at an intermediate node before it could be forwarded to the next node. But more recent switching techniques such as virtual cut through and wormhole switching are more popular for multicomputer networks as they attempt to forward an incoming packet as soon as the header information of the packet has arrived. Subsequent bytes of the message follow on the path established for the header previously and the message transfer is essentially pipelined through the network. These switching techniques result in low latency for message transfer than packet switching. The unit of information transfer for these techniques is a flit and a packet is comprised of flits. Virtual cut through and wormhole routing differ in their buffering mechanism. With virtual cut through, if the packet header is blocked for an output channel, the entire packet must be buffered along with the header at that node. Virtual cut through thus requires that every router have buffer space to buffer at least one packet entirely. Wormhole routing has more relaxed buffer requirements than cut through. The buffer at each router can be large enough just to hold a few flits. With the header of a packet blocked on an output channel, the packets flits occupy buffer space in several routers. In the absence of blocking, the packet transfer is pipelined through the network just as in cut through. Both unidirectional and bi-directional wormhole and virtual cut through switching techniques have been implemented within the canonical model.

- **Virtual Channels**

A physical channel may hold several virtual channels that are multiplexed across the physical channel. Each virtual channel can in turn hold buffer space to buffer flits of blocked packets. Use of virtual channels greatly improves latency and throughput of the network. Deadlock prevention in wormhole switched networks is another reason for the introduction of virtual channels in the model. Arbitration is required to decide which virtual channel can get hold of the physical channel, if there are colliding requests among them. Arbitration is either random, first in first out or round robin based in the model.

- **Buffering Scheme**

A very important network parameter is the size and position of buffers in the router. Each virtual lane can have sufficient space to buffer away a few flits. The buffer size is critical to both wormhole and cut through switching but especially more for cut through as it has to guarantee space for at least one packet. The buffers are FIFO buffers and routers can have these buffers per physical channel, per virtual channel or per lane within the virtual channels. The position of the buffer is also a design tradeoff. Buffering may be done only at the inputs (input buffering) or only at the output (output buffering) or at both the input and output side. Buffer size is modeled as a per node parameter in the NetSim router model.

- **Crossbar Switch**

A crossbar is used for connecting the inputs to the outputs. The crossbar switch having N inputs and M outputs allows up to min{N,M} one to one connections when no contention occurs. The two common crossbar configurations are the single crossbar with direct connection between lanes and the cascaded crossbars in which the output of the X crossbar is fed to the input of the Y crossbar. Both are modeled inside NetSim. The crossbar model has ports for all the virtual channels and the lanes within them.

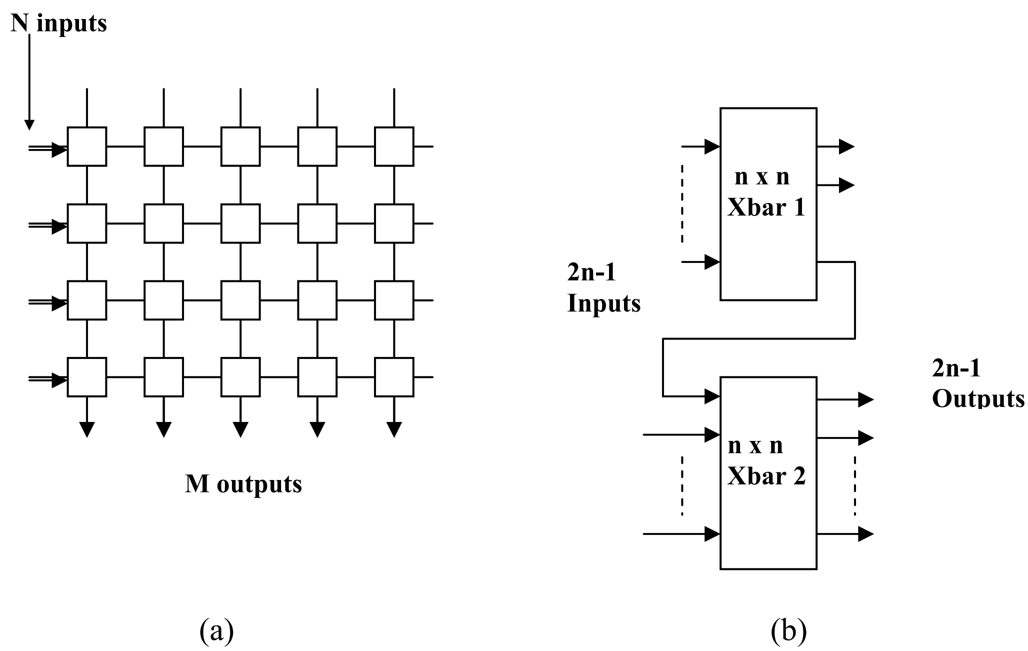

Figure 6. Crossbar Switch Architecture (a) N x M crossbar (b) Cascaded crossbar

- **Arbitration Unit**

When multiple sources request for the same output virtual channel at the same time, arbitration is essential to choose only one amongst the conflicting requests. The selection of the input channel can be done either using the first come first served scheme or randomly or in a round robin fashion. Fast arbitration is crucial to maintain low latency of the switch especially for large network or switch design using full connectivity.

- **Routing Unit**

The routing unit implements the routing algorithm, which selects an output channel for the incoming request and sets the crossbar switch accordingly. Several routing algorithms have been proposed for wormhole and cut through switched networks and each of these is aimed at providing better and faster connectivity from source to destination node. With wormhole switching, cyclic dependencies occur which lead to a deadlock. Routing algorithms must thus be designed to be deadlock free. Also it would be preferred if routing adapts itself to the current network state and selects an alternative route for the incoming packet if the current selected route is busy, so that the packet does not get blocked. Thus the implementation classes of the routing algorithms are deterministic, adaptive and partially adaptive.

With deterministic routing the route selected between a source and destination pair is always the same. Dimension order routing is a deterministic routing algorithm that routes packets by crossing dimensions in a strictly increasing (or decreasing) order. For an n dimensional mesh and hypercube, this routing is deadlock free but for a k-ary n-cube tori network it introduces cycles and thus a possibility of a deadlock. A deadlock free design of the routing algorithm is presented in [8] for the torus network.

Adaptive algorithms on the other hand try to make use of the network state while making a decision on the output path to be selected. This type of routing is preferred because the possibility of a packet holding any buffer space or being blocked is reduced as the packet can now take an alternate route. This adaptivity can be allowed either for a subset of the output physical paths possible (known as partially adaptive routing) or can utilize any of the output physical paths with no restriction at all(known as fully adaptive routing).

More adaptivity improves performance. However increase in adaptivity increases the complexity of the router and can affect its operating frequency. Partially adaptive routing algorithms are thus designed as a tradeoff between speed and cost.

Several fully adaptive routing algorithms have been proposed. Many of them make use of a large number of virtual channels and this tends to slow down the router speed while increasing the chip area. Some of the practical fully adaptive algorithms that use reasonable number of virtual channels have been modeled in the router. These are also designed to be deadlock free.

3.2.2 Basic Parameter Settings

Some parameters are fixed within the simulator and are not programmable by the user. These parameters settings are based upon reasonable assumptions for network simulation and are listed below

- The width of the physical channel is assumed to be one flit.
- There is only one injector port and one ejector port for the processor interface.
- The number of virtual channels is specific to the routing algorithm selected and is computed within the simulator.
- The network is tightly coupled. So the wiring delay is not critical and the transfer from an output channel to a downstream input channel takes one cycle.

- The transmission of flits is pipelined.
- A flit takes one cycle to be transferred from the input buffer to the output buffer within the router or from the output buffer to the input buffer of the next router if there is no congestion along the path. A header flit injected into an empty buffer would take an extra cycle to be routed. The routing for the header flit following the tail flit of another packet will be done in parallel with the transmission of this tail flit.

Chapter 4 - System Integration

In this chapter, the design and implementation of the network interface required for integrating CAL-SIM and NetSim is presented. We focus on how the network synchronizes itself with the processor nodes in simulation.

4.1 Interface Design and Implementation

Within the interface, the function of a network interface card for each node is implemented. This network interface card resides between a processor and the network. It performs the task of processing a message into packets that are to be sent into the network and also informs the processor when a message has been received in its entirety. The network interface thus allows the processor to continue with its job as the message transmission is being done in parallel.

The interface also provides a set of functions for interacting with the network simulator. Its simple design allows users to replace the network simulator with their own version if required. That is, the software interface to the network simulator is not very specific to the implementation details within the network simulator. This flexibility is essential because a large number of network models exist and it is difficult to support all of them in our network simulator. However the user network simulator must adhere to certain design guidelines to work successfully with the simulation system. These design guidelines will be mentioned in this chapter.

The overall simulator design is shown in figure 7. CAL-SIM simulation system includes the MPI library, simulator core, which is responsible for thread scheduling and the network interface, which is used to integrate the network simulator. The network simulator, NetSim includes the network setup, which configures the network parameters for simulation and the network core, which simulates a single cycle of network execution. The network interface now has to coordinate the running of the network core with the processor threads every time a MPI communication routine gets called. It must also interact with the network simulator to have a message sent or received from the network.

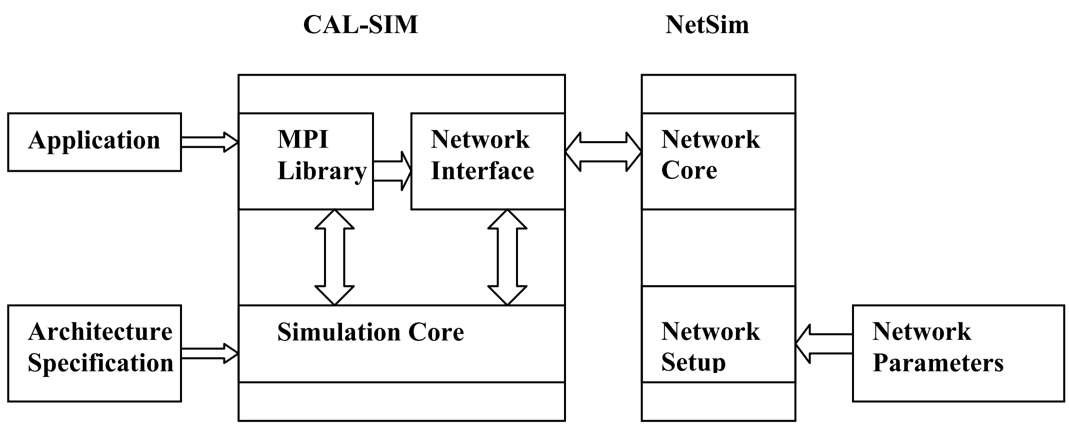

Figure 7. Simulation system design

4.1.1 Send Side

When a message is to be sent, the processor signals the network interface using an interface routine. The network interface further performs the tasks of transferring the message from processor's memory to the interface memory, creating message packets and transmitting the packets to the node router.

For the transfer of a message from the processor's memory to the network interface memory, a memory-mapped device or a DMA within the interface is assumed. In simulation this data transfer can be done by creating a copy of the message within the interface and releasing the sender data buffer. The send function call can now return, as the user buffer is safe to reuse. The overhead for initiating the message transfer is computed using a delay model. The first packet of the message is thus created at a time equal to the current simulation time plus the sender transmit time, where sender transmit time is the initial setup time to transfer data to the interface memory.

The network interface then computes the number of message packets from the message size and the flit size. Packet creation involves creating the header flit that contains routing information for the packet and its following data flits. The header is assumed to be one flit and a single packet creation takes one cycle. The interface maintains a packet queue to hold the outgoing message packets. The size of the packet queue is set to a default value and can be reset by the user. This feature enables to see the effect of queuing latency when the network is saturated, which results in packets holding buffer space within the interface memory and preventing any further packet creation.

Within the network interface, flit injection for a previous packet proceeds in parallel with the creation of a new packet. To inject a flit into the router injection channel buffer, the network interface calls the injector routine of the network simulator.

4.1.2 Receive Side

A global message list is maintained by the interface to keep track of the messages being sent into the network. When the network simulator notifies the interface about the arrival of a message packet, the message list is looked up by an interface function to check which message the packet belongs to and if this is the last packet for the message. If all the packets for the message have been received, the network interface schedules a message arrival event at a time equal to the current simulation plus receiver transfer time. The receiver transfer time represents the time to transfer a message from the interface to the processor memory. The message arrival event places the message in the receive queue of the destination processor and signals a receive semaphore.

A processor thread waiting for a message on the receive semaphore, is triggered by the semaphore signal. The receiver checks to see if the intended message has arrived and if so, copies it into the user buffer and returns. If the expected message has not yet arrived, then the receiver simply waits for another signal from the receive semaphore.

4.2 Network Event

The simulation library within CAL-SIM consists of two simulation activities, processes and events. The main difference between these two entities is that a process can temporarily suspend execution while an event must execute till completion. Once the body of an event terminates, the event thread is destroyed. The only way to have an event schedule itself multiple times (reschedule itself) is to have the event as a non-deleting entity.

The interface to the network and the network simulator together form a network event, which is designated as a non-deleting event. An event is chosen over a process to avoid the overhead for a process when a context switch occurs from the network thread to some other ready thread. Moreover an event does not affect simulation time; it takes 0-simulation time to execute an event.

4.3 Scheduling the Network Event

The interface to the network simulator is responsible for synchronizing the network thread with the running processor threads. The network event should schedule itself at the correct simulation time and keep its simulation time synchronized with that of other running threads. This is essential so that message packets are injected into the network at the correct simulation time and also received at the correct time. Two important issues need to be considered here. Firstly, all the processor threads must have run before the network thread can run. This way all the messages for the current simulation time get inserted in the message queue. Secondly, there needs to be a way of knowing how long the network simulator can run. It may happen that if the network simulator is run till a message has reached, the cycle count would advance and a processor thread that has not been able to catch up to that time, will send messages meant to be sent at an earlier simulation time. This will lead to incorrect network simulation. There needs to be a way to get around both the problems for accurate network simulation.

To start with, all the threads get scheduled at simulation time 0. When the network thread enters execution, its first task is to check if any other activity is scheduled for the same time as itself. This is possible using a simulator core function that allows performing a look-ahead into the activity schedule queue. This queue is ordered by the activity schedule times and allows for insertion of activities in the queue dynamically, as these get scheduled from the current simulation time. The look-ahead function returns the schedule time of the next activity and the interface routine checks to see if this time is equal to the current simulation time. It then lets this activity proceed first by rescheduling itself at the current simulation time. This adds the network event back to the activity queue without its further execution. When the current activity is finished, the network event is again activated and if no other thread is still scheduled, the interface proceeds to execute the network code. This way all the processor threads have had a chance to run and put their messages in their respective message queues before the network thread can run. Figure 8 depicts the rescheduling of the network event by performing a look-ahead into the activity queue.

Time = 0	(T)
Proc 1	10
Proc 2	10
Net Event	10
Proc 3	20
Proc 4	20

→

Time=10	(T)
Net Event	10
Proc 1	10
Proc 2	10
Proc 3	20
Proc 4	20

→

Time=10	(T)
Proc 1	10
Proc 2	10
Net Event	10
Proc 3	20
Proc 4	20

→

Time=10	(T)
Net Event	10
Proc 3	20
Proc 4	20
Proc 1	40
Proc 2	55

Figure 8. Scheduling the network event with look-ahead

The other situation where the difficulty in scheduling the network event arises is when all the processors are waiting for their messages on their message receive semaphore. Such a situation frequently arises during execution, as the receiving processor enters its receive routine even before the sender has had a chance to enter its send routine. The receiver, if it is in a blocking receive mode or is performing a wait operation to have the data buffer filled, simply suspends itself on the receive semaphore. Only when the network interface puts the message into the receiver's message queue does the receiver get awakened and start further execution. In the meantime, the sender after sending the message can itself perform a receive operation waiting for a message from the receiver of its message. So both the sender and receiver are now blocked on their respective semaphores and the time of their further activity is unknown. Activities waiting on a semaphore do not get added to the activity schedule list but instead are added to the semaphore list of the receive semaphore.

With all the processor threads waiting on their respective semaphores, the look-ahead method mentioned above fails. This is because the interface checks for the next event time and the look-ahead function not finding any activities in the queue returns a negative value upon which the network event may delete itself thinking all other processes have terminated. The way to get around this problem is to have the network event check for two conditions.

1. To check for the next event time using look-ahead
2. If the look-ahead returns a negative value then check to see if there are processors waiting on their semaphores.

The second condition can be checked using another simulator core function. In such a situation the network event has to keep running until at least one of the processors is awakened for the look-ahead to work again. However, if both the look-ahead and the check for semaphore wait return negative results, it means that all the processor threads have terminated and it is now safe for the network event to exit.

Once the network event is scheduled, the next question is how long should the network run. This is also an important issue here, as the network has to run only for certain cycles so that the processor threads can catch up and the simulation of network activities occurs at the correct simulation cycle. The network running for a time to have an entire message delivered to its destination may result in the network cycle count exceeding the processors simulation time. Any further insertion of packets in the network by the processor will be at the incorrect cycle for the network, leading to inaccurate simulation results. The best way to avoid this is to have the network run for a time less than the schedule time of the next activity. The look-ahead function is used here again. The network can run for a time frame between current simulation time and the next event's time that is returned by the look-ahead function. So lets say the network event is scheduled at time 0 and the next event time is 10, the network simulator is run from time $t = 0$ to $t = 9$.

The additional case where the time for the network simulator run needs to be decided is when processors are waiting on their message semaphore as mentioned above. In this case as look-ahead fails, some other approach needs to be taken to decide how long the network simulator can run. We know that when the final packet for a message gets ejected, the network interface signals the destination processor about the message arrival. This wakes up the destination processor and adds it back to the activity schedule queue. With at least one other activity besides the network event in the activity queue, the look-ahead method will start functioning again. So the method adopted here is to let the network run till at least one message reaches its destination. The network event can then reschedule itself at the message arrival time letting the awakened processor thread resume its execution.

4.4 Reducing Simulation Overhead

Detailed cycle-by-cycle simulation of the network tends to be slow for a large network size and for a large number of messages through the network. It would thus be essential to reduce as far as possible the overhead introduced by detailed simulation. Unnecessary network simulation, for example when there are no packets in the network can be skipped by the network event. If the network event is to run from the current simulation time to the time of the next activity which may be tens of thousands of cycle away, the number of packets injected into the network can be checked. If all the packets have reached their destination, further simulation till the next event time is a large overhead and so the network cycle is simply made to advance to the next event time. The basic code frame for the network event is shown below.

```
Network_Event( )  {
   next_event_time = time_of_next_activity_in_queue
      if(next_event_time != -1)  {        // look-ahead succeeds
         if(cur_time == next_event_time)
         Reschedule_Network_Event
         else
         while(cur_time < next_event_time) {
            if( ! injected_pkts )
               network_cycle = next_event_time
               break
            else
               simulate_a_network_cycle
         }
      }
      else if any processors waiting on message semaphore {
         while ( ! msg_arrived)
            simulate_a_network_cycle
      Reschedule Network_Event at network_cycle – cur_time
   }
   else terminate Network _Event
```

Figure 9. Code format for network event

4.5 Design Guidelines for the Network Model

The network interface provides for easy integration of different network simulators with the simulation system. The interface interacts with the network simulator through a set of functions. This way unnecessary details of CAL-SIM implementation are hidden to the network simulator and vice versa. The network simulator however has to follow certain design rules to ensure its correct working with the system. These are listed below.

- The network simulator must be cycle driven
- The network simulator code must be organized in two parts; an initialization part which sets up the network configuration using a wrapper function called NS_Init( ) and a single cycle execution part, which simulates the activities in the network for a single cycle in a wrapper function called NS_SimCycle( ).

- A global data structure called Net and a pointer to it need to be maintained within the network simulator. The elements of this data structure required for integration are the PacketArray and the cycle counter variable called global_cycle. PacketArray must contain variables called pkt_cylcreated (the cycle at which the packet got injected into the router), pkt_cylarrived (the cycle at which the packet is ejected through the ejector port of the router), pktid (the id of the packet ejected) and msgnum (the message number to which this packet belongs). These fields are required by the interface when a packet gets ejected, to maintain a check of when a message arrives in its entirety.
- The global_cycle variable is the network cycle counter, it is incremented for every simulated cycle of the network. But control of this variable is also required by the interface.
- A function NS_QuePkt( ) must be provided which creates a packet in the PacketArray and returns a positive value if the packet could be successfully created.
- When the tail flit of a packet gets ejected through the ejector port, a wrapper function called NS_Eject( ) must be called, which informs the interface of the incoming packet.

Chapter 5 - Simulation Results

In this chapter simulation results from the integration of CAL-SIM and NetSim are presented which demonstrate the working of the two tools together. We examine the communication performance of MPI benchmark applications on target systems and report on system performance parameters such as message latency. We also compare the message latencies predicted by a linear message delay model and by detailed network simulation for the benchmarks.

All the simulations were carried out on a 360MHz Sun UltraSparcII, solaris2.7 system. Simulation results are presented for the NAS 2.3 Integer Sort Class W benchmark, Multigrid and Embarassingly Parallel Class A benchmarks and for the Fast Fourier Transform application. NAS benchmarks (except IS) are in fortran77 and as the simulator can run C/MPI applications, the benchmarks are converted to C using the F2C tool.

5.1 Simulator Parameter Settings

A two-dimensional torus network is assumed in the simulations for comparing the message latency predicted by the simple mode of the simulator using linear message delay and the network mode involving network simulation. The network sizes used are 4 x 4, 8 x 8 and 16 x 16. Thus the number of nodes simulated is 16, 64 and 256.

5.1.1 Linear Delay Model Parameter Settings

The ideal message latency, which does not take into account network traffic and contention, is presented in [7]. This delay model is used in the simple mode of the simulator. Message latency is this case is given by

$$\textbf{Message Latency} = D * (Tr + Ts + Tw) + Ts(L/W) \qquad (1)$$

where D – average distance,

Tr – router delay, Ts – Switching delay, Tw – Wire delay,

L – Message Length in flits, and W – Physical Channel Width in flits

The average distance in a two-dimensional torus is 2*K/4, where K is the number of nodes in a dimension. This is true when K is even. The parameters Tr, Ts, Tw are set according to the way the network simulator has modeled them to provide fairness of comparison. Our evaluations are for tightly coupled networks, where the wiring delay is not critical and so Tw is set to one cycle. The routing delay and the switch delay are also set to be one cycle each. The width of the physical channel is equal to one flit, which is equal to one 64-bit word. The message latency equation can thus be re-written as

$$\textbf{Message Latency} = (2 * K / 4) * (1 + 1 + 1) + (1) * (L / 1) \qquad (2)$$

5.1.2 Network Mode Parameter Settings

The detailed network simulation mode allows for setting of more communication network parameters than the simple delay model presented above and is an advantage associated with network simulation. The following network parameters are assumed in network simulation:

Table 1. Network parameter settings

Topology	2D Torus
Width (Radix K)	4, 8, 16
Wiring	Bidirectional Wormhole
FIFO size per node	108 flits
Packet Size in Flits	8
Flit size in bits	64
Routing Algorithm	Deterministic TRC

5.2 Results for the NAS Multigrid Benchmark

The benchmark under evaluation here is the NAS MultiGrid (MG) Class A benchmark. The benchmark solves a 3D Poisson partial differential equation. The code has a good mix of short and long distance communication. The dominant communication type here is point-to-point with blocking sends and non-blocking receives.

It has been shown [9] for the NAS benchmarks that with increasing machine size, the size of a message usually decreases. However the number of messages is likely increased. The results in figure 10 say the same. Only the messages sent over the communication network are considered. With increased number of nodes, the communication is of finer granularity leading to a decrease in the average message size. However the communication pattern now requires interaction with more number of nodes increasing the total number of messages.

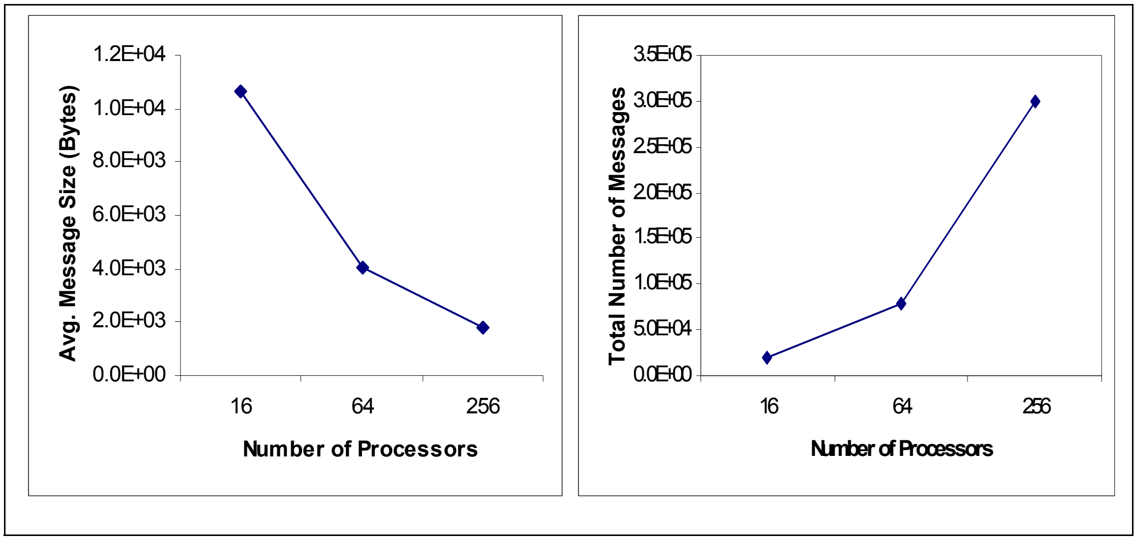

Figure 10. Avg. message size and total number of messages as a function of machine size

With smaller message size, the average message latency experienced by a message would be expected to reduce. This can be observed in figure 11. However the network mode for 256 processors shows an increase in the average message latency. As the number of nodes is increased beyond 64, the communication pattern for the benchmark changes rapidly.

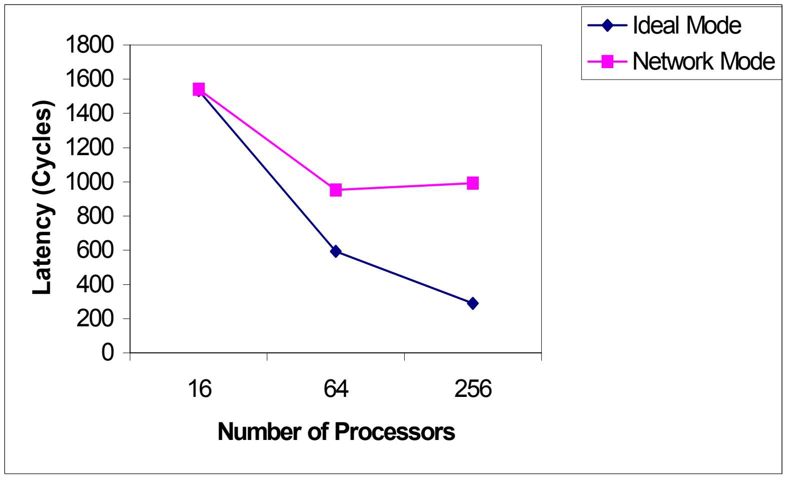

Figure 11. Predicted average message latency for the MG Class A benchmark

To observe the impact of the routing algorithm in such a case where the static routing scheme performance degrades, the algorithms are varied to be partially and fully adaptive. Figure 12 shows the average message latency change with a change in the routing algorithm. The adaptive algorithms under test here are Duatos partial and fully adaptive and the fully adaptive T3D-like.

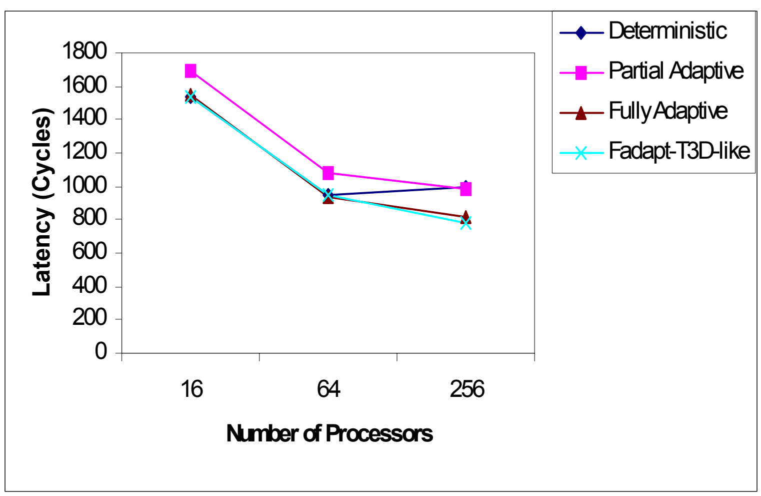

Figure 12. Comparison of routing algorithm performance

From figure 12, it is seen that the adaptive algorithm scales well with increasing machine size. Partial adaptive however does not perform any better than deterministic routing for this particular workload.

Even though the number of messages and message size is significant for the benchmark, computation still dominates the overall execution time. The total amount of communication is less than 20% as seen in figure 13. MG benefits from the increased number of processors. The execution time and change in communication performance across processors is presented in figure 14. The communication time considered here includes the time spent by processor threads inside a communication routine waiting for the intended message. The wait time may be due to imbalance in the send and receive schedule times and also due to the actual communication network cost.

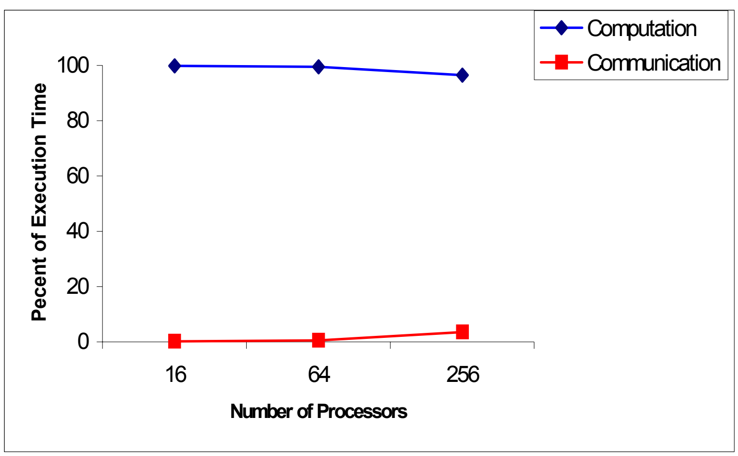

Figure 13. Percent computation and communication in MG benchmark

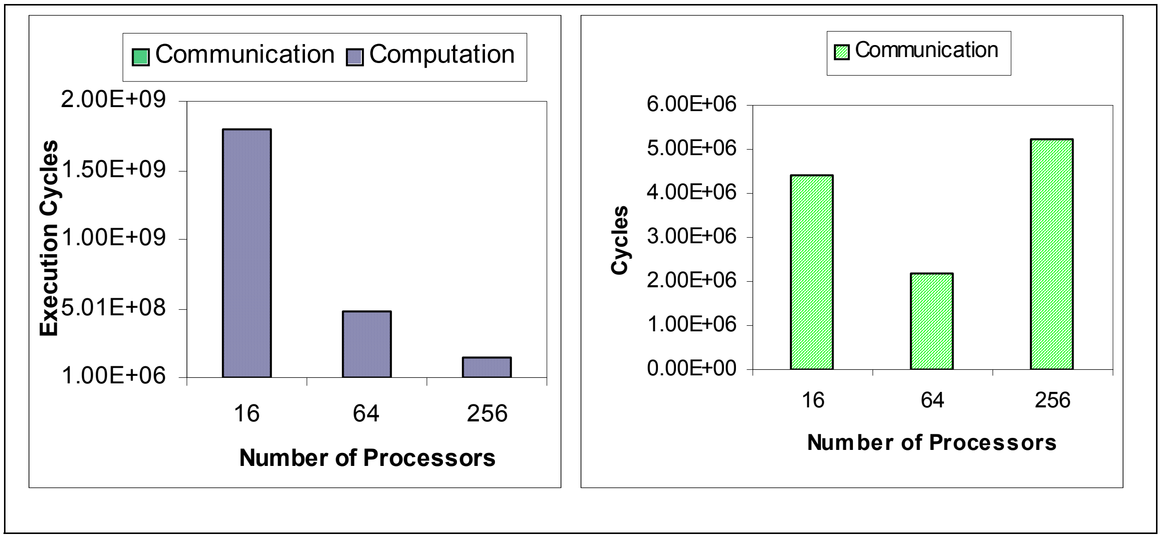

Figure 14. Predicted performance for the Multigrid Class A benchmark

5.3 Results for the NAS Integer Sort Benchmark

The Integer Sort (IS) benchmark tests a sorting operation on primarily integer data type. The benchmark has a significant amount of communication as compared to most other NAS benchmarks. Reduction and all to all function calls dominate the communication. Usually for collective communication routines, the vendor specific MPI library implementation plays an important role in deciding how many messages get generated. The routines may be optimized to reduce the volume of traffic to be sent over the network. The MPI library within the simulator currently has a straightforward and reasonable implementation of the collective communications that provide functional support for those found in the NAS benchmarks. Some deviation in the number of messages generated for such collective communications is therefore expected here. The average message size and the total number of messages for the benchmark are shown in figure 15. Due to all to all communication the number of messages generated increases significantly with increase in number of processors especially from machine size of 64 to 256.

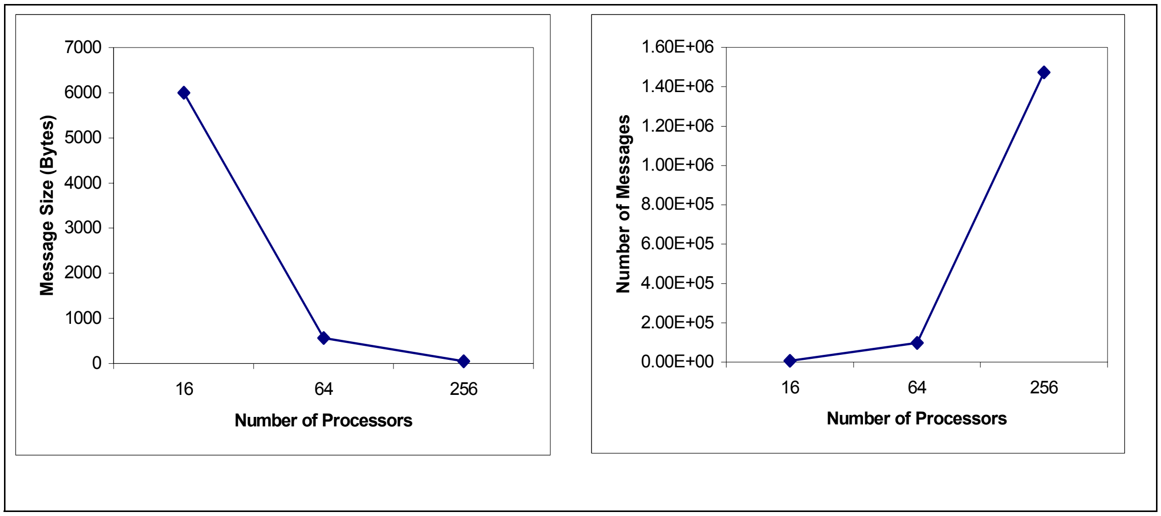

Figure 15. Avg. message size and total number of messages as a function of machine size for the IS Class W Benchmark

Figure 16 shows the predicted average message latency in the simple and network mode of the simulator. The routing algorithm was varied to see the impact on message latency as in figure 17. The topology was next selected as mesh to observe how much performance advantage the torus layout offers because of the wraparound channels. Figure 18 presents the average message latency on a mesh. The torus layout only very slightly performed better than the mesh and the difference is almost negligible.

Although the computation efficiency of the benchmark increases with machine size, the communication increase is very large. Communication thus dominates the benchmark performance at very large machine sizes. Figure 19 shows the communication increase with the increasing number of processors.

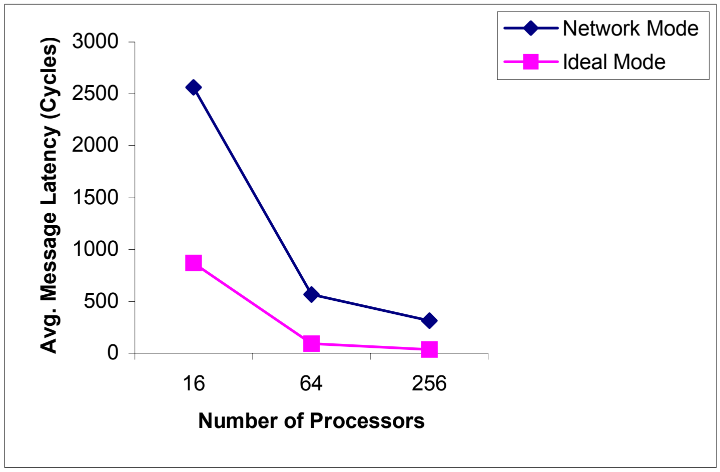

Figure 16. Avg. message latency prediction for the IS Class W benchmark

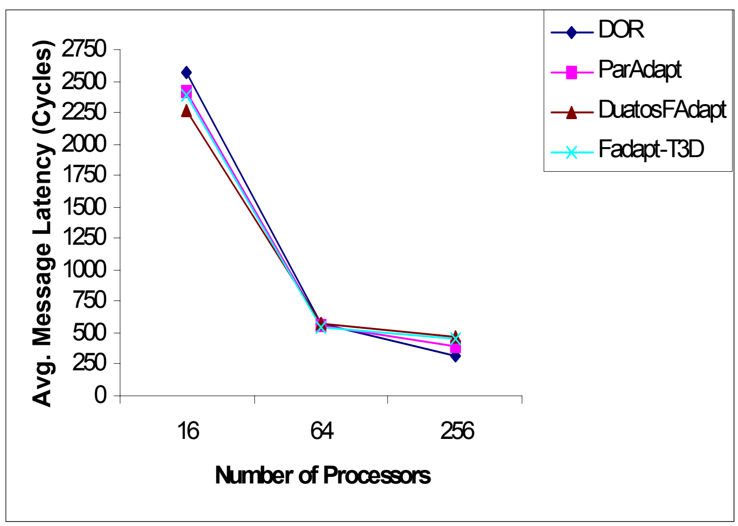

Figure 17. Avg. message latency with a variation in routing algorithm for the IS benchmark

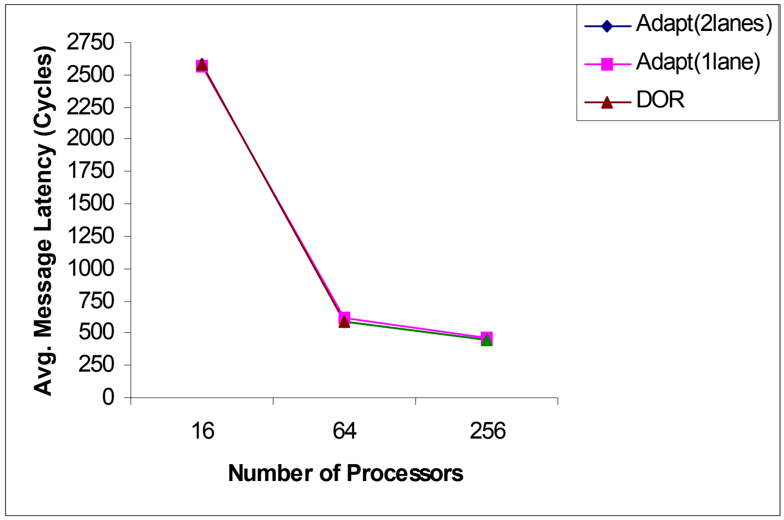

Figure 18. Avg. message latency on a 2D mesh for the IS Class W benchmark

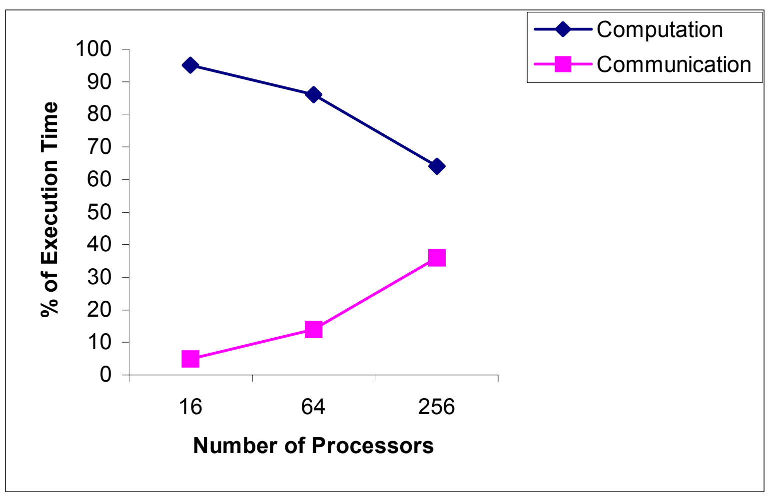

Figure 19. Percent computation and communication in the IS Class W benchmark

Figure 20 shows the performance of the benchmark. The code benefits from increased number of processors up to 256. For communication performance of the application, the effect of startup latency is not considered. The communication behavior helps in understanding the contribution of network cost to the communication time of the application.

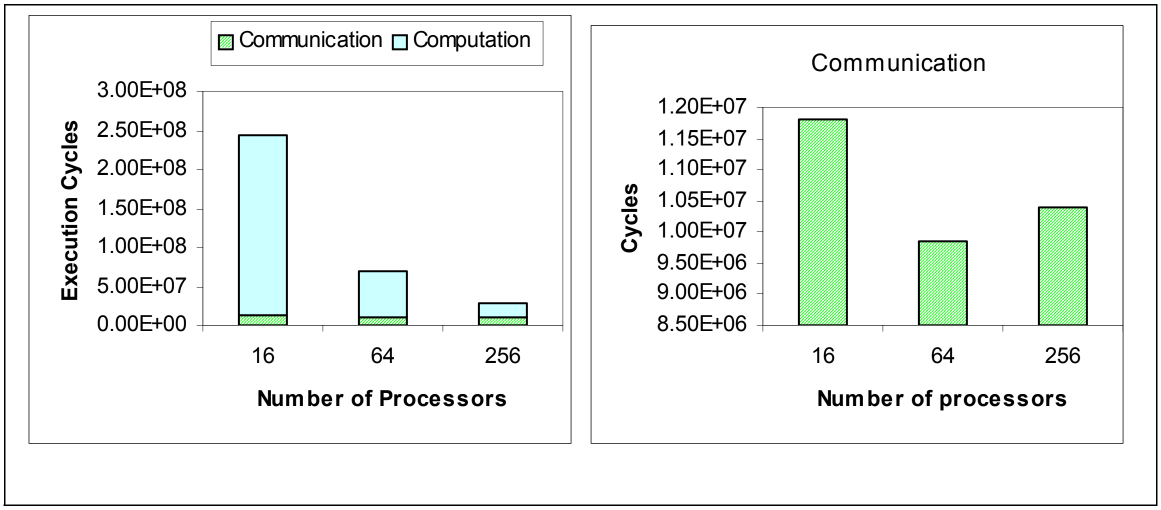

Figure 20. Predicted performance for the IS Class W benchmark

5.4 Results for the NAS Embarrassingly Parallel Benchmark

The Embarrassingly Parallel benchmark generates a large number of gaussian pseudorandom numbers. The benchmark has very little communication. Most of this modest communication that is present, is collective type. One reason the benchmark was selected for study here was that it has a constant message size across varying machine size. It allows seeing the network performance for a fixed message size. The average message latency is seen to increase with increase in number of processors as in Figure 21. The predicted performance for the benchmark is shown in figure 22.

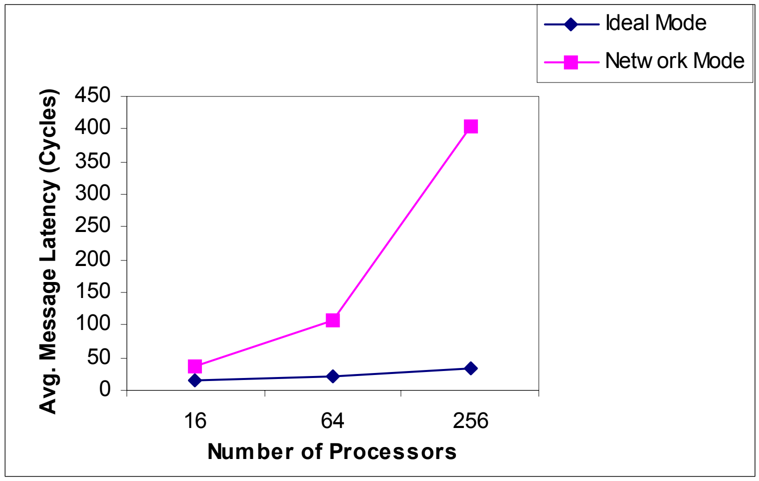

Figure 21. Average message latency for the EP Class A benchmark

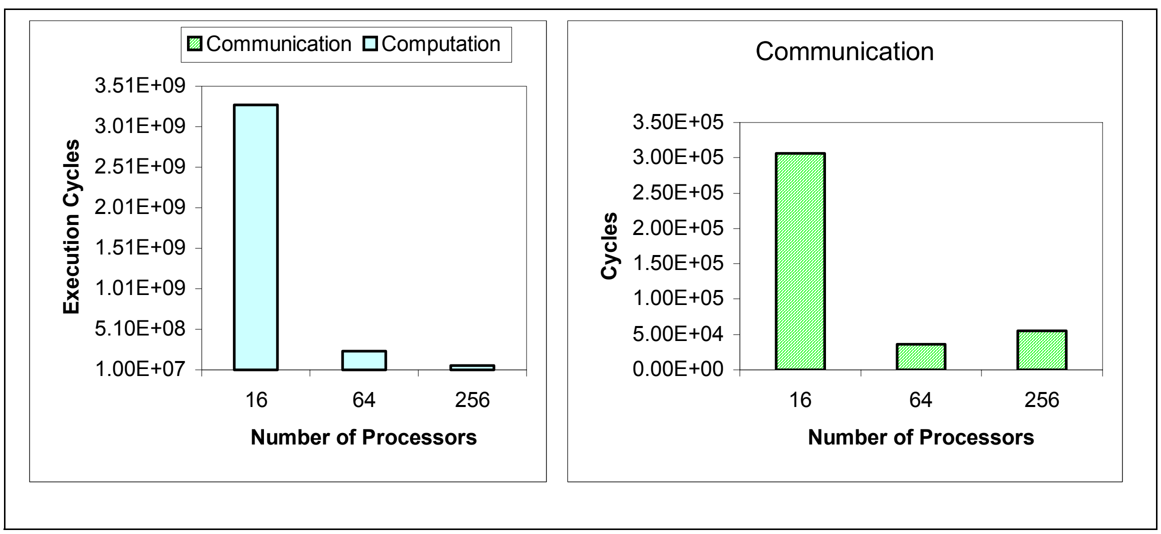

Figure 22. Predicted performance for the EP Class A benchmark

5.5 Results for the FFT Benchmark

The FFT application solves the partial differential equation using fast fourier transforms of given N numbers. Each of the P processors computes the FFT of N/P numbers and communicates with other processors to compute for other remaining elements. The FFT application does less than 20 percent communication in the code for processors up to 64. Figure 23 shows the communication time spent by FFT and figure 24 gives the performance for FFT.

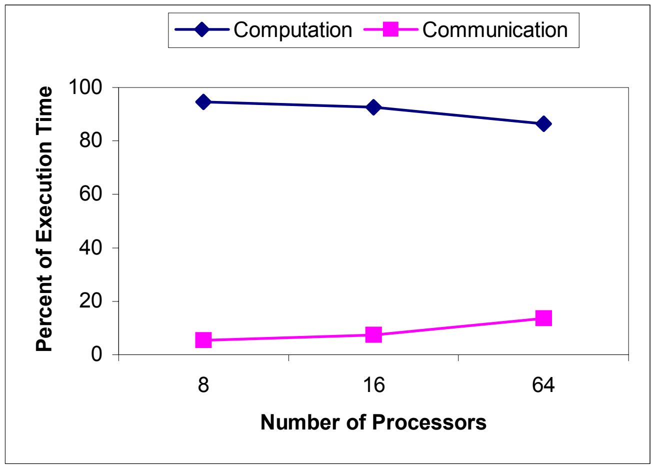

Figure 23. Communication time for FFT benchmark

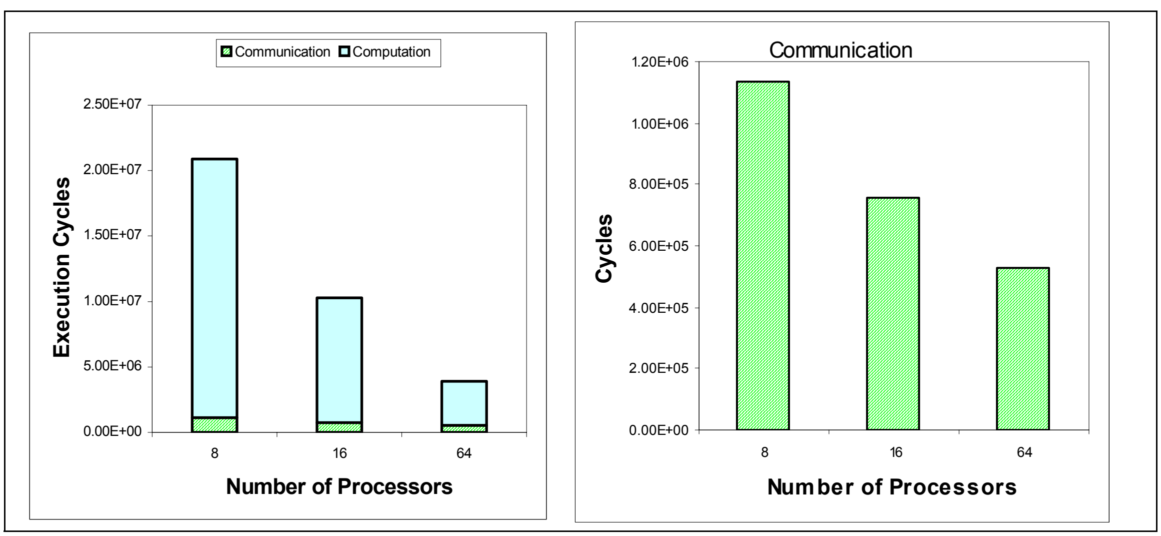

Figure 24. Performance of the FFT benchmark

5.6 Prediction Error for Simple Mode of the Simulator

Figure 25 shows the difference in the communication performance prediction for the MG and IS benchmarks by the simple mode of the simulator, which uses a linear delay model and by the network mode of the simulator, which supports detailed network simulation. The linear delay model introduces error in the predictions, as it does not account for the current workload and contention in the communication network.

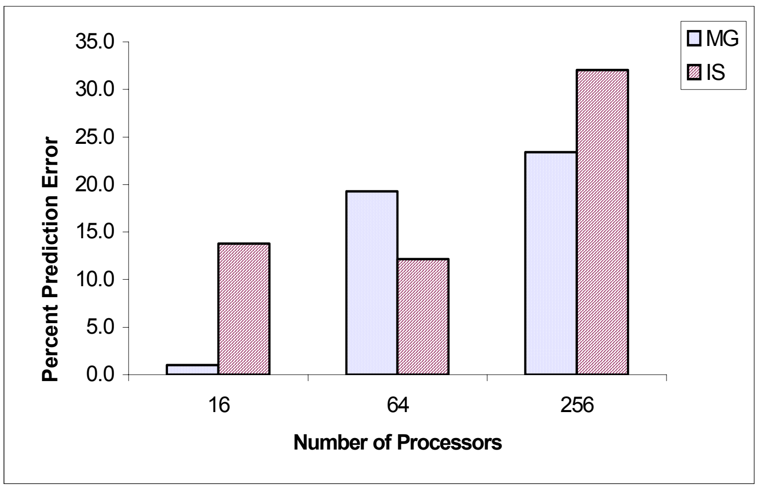

Figure 25. Prediction error for communication performance prediction using the linear delay model

5.7 Functional Verification of the Simulator

To observe if the interface functions correctly two things need to be verified. First if the number of packets created from the message size and packet size are correct and second if the packets are being created and inserted in the network at the appropriate cycle. Since a common clock is assumed for the processor cycles and network cycles, it is pretty straightforward to observe if the packets are being inserted at the right cycle.

A sample code in which each processor sends a message to every other processor and then waits for the other processor messages was used as the initial test. The number of processors and the message size was varied. The MPI library was instrumented to see when the message was created and the instrumented interface code was checked to see if the packets get created in the following cycles. Information was also collected to see if the packet flits get injected into the network at the correct cycle count.

The NAS benchmarks perform self-verification to see if the result obtained is equal to the precomputed result. The simulator has been successfully tested for the following benchmarks.

Table 2. Benchmark Tests

Application	Class Type	Number of Processors
NPB MG	W, A	1,2, 4, 8, 16, 32, 64, 256
NPB EP	W, A	1,2, 4, 8, 16, 32, 64, 256
NPB IS	W	1, 2, 4, 8, 16, 64, 256
FFT		1, 2, 4, 8, 16, 64

Another test for functional verification is to use the ideal delay model given above for a 2D torus and have a single message's packets travel through the network. This way there is no contention in the network and the delay predicted by the network mode and the ideal mode should be close in value. There may be a slight difference in the two values as the ideal mode uses average distance and the network mode uses the actual distance. But for case of comparison here, an 8 x 8 network size is chosen and the message is sent from processor 0 to processor 10, with the distance between node 0 and node 10 equal to the average distance within the simulator. This way same results should be obtained from both the modes. The results from network simulator verify this as shown in table 3, with varying message size. The remaining parameters for the network and simple mode are the same as given in section 5.1.

Table 3. Functional Verification

Message Size in Flits	Message Packets Created	Message Latency In Ideal Mode	Message Latency In Network Mode
64	10	92	92
128	19	164	164
256	37	308	308
512	74	604	604

Chapter 6 - Conclusion and Future Work

6.1 Conclusion

In this work, a flexible network interface for supporting network simulation within the CAL-SIM simulation system is designed and implemented. The following goals have been achieved with this work:

1. The capability to specify an interconnection structure for the processors in the simulated architecture model. This allows us to evaluate the impact of network designs on the communication performance of applications, which are executed on the system.
2. More accurate performance measurements. Detailed network simulation provides more accurate event timing for the send and receive operations than the simple latency model. The event queue within the simulator as a result processes the events in a more precise order.
3. A platform for evaluating candidate network designs using real applications as workload instead of synthetic workloads. The traffic pattern generated by the application directly simulates the network. The effect of traffic burstiness and bimodal messages through the network can also be captured. These however cannot be easily modeled with synthetic workloads.
4. The flexible network interface allows easy integration of different network simulators with the simulation system.

6.2 Recommendations for Future Work

Network simulation tends to be slow when a large network with a great amount of network activity needs to be simulated. Parallelizing the simulation system could help in reducing the time of simulation.

The current simulator model assumes a single processor per node. But looking at the current trend in supercomputing most new architectures are designed to be cluster architectures with more than one processor per node. Within the node shared memory is used and between nodes a message-passing network is used. Examples of such architectures include the IBM RS6000 SP, SGI Origin, Compaq AlphaServer. So the next task in the simulator design extension would be to include support for simulating a cluster based architecture, by making each node a shared memory multiprocessor.

This task could be simplified by integrating an existing shared memory simulator with the simulation system. An example of such a publicly available shared memory simulator that can be used for integration is the Multiprocessor SimpleScalar tool set [14]. A simple model that can be adopted for the execution of applications on simulated cluster architectures is presented in Figure 26.

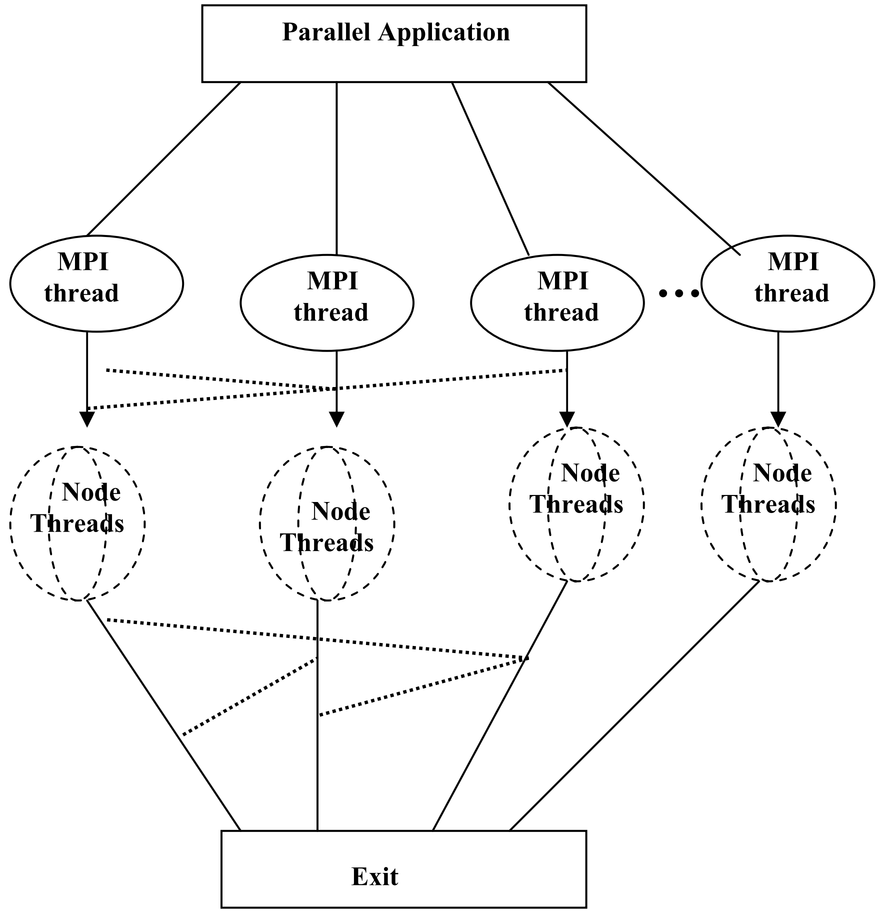

Figure 26. Execution flow for a cluster architecture in simulation

Here the master threads representing each node get created first. Then each master thread forks new threads, which represent the processors per node. Within a single node, all the threads communicate using shared memory. But only the master thread of a node can carry out any communication with a processor thread in some other node. This inter-node communication proceeds using MPI communication routines. Finally when all the slave threads for a node exit, the master thread also terminates.

References

1. Yun Lin, “CAL-SIM: A Distributed Memory MPI Multi-Computer Simulation System”, Masters Thesis, University of Houston, Spring 2001

2. Jinming Ge, “Cost-Effectiveness Evaluation of a Dynamic-Balanced Adaptive Wormhole Router”, PHD Thesis, University of Houston, Fall 2001

3. MPI: A Message Passing Interface Standard, Message Passing Interface Forum, June 12, 1995

4. Covington, R.G., Madala, S., Mehta, V., Jump., J.R., Sinclair, J.B., “The Rice Parallel Processing Testbed”, Proceedings of the 1988 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems, May 1988, pages 4-11

5. Covington, R.G., Dwarkadas, S., Jump, J.R., Sinclair, J.B., Madala, S., “Efficient Simulation of Parallel Computer Systems”, International Journal in Computer Simulation, June 1991, Vol 1, pages 31-58

6. Dwarkadas S., Jump, J.R., Sinclair, J.B., “Execution Driven Simulation of Multiprocessors: Address and Timing Analysis”, Journal of Transactions on Modeling and Computer Simulation, 1994

7. Herbordt, M.C., Ge, J., Sanikp, S., Olin K., Le H., “Design Trade-Offs of Low Cost Multicomputer Network Switches”, Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation, 1999, pages 25-34

8. Duato, J., Yalamanchili, S., and Ni, L., “Interconnection networks: An Engineering Approach”, IEEE Computer Society Press, Los Alamitos, CA, 1997

9. Wong, F.C., Martin, R.P., Arpaci-Dusseau, R.H., Wu, D.T., Culler, D.E., “Architectural requirements and scalability of the NAS parallel benchmarks” Proceedings of Supercomputing’99, 1999

10. Hwang, K., Wang, Choming, Wang, Cho-Li, “Evaluating MPI collective communication on the SP2, T3D, and Paragon Multicomputers”, Third International Symposium on High Performance Computer Architecture, Feb 1997, pages 106-115

11. McKinley, P.K., Trefflz, C., “MultiSim: A Simulation Tool for the Study of Large-Scale Multiprocessors”, Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Networks,Jan 19993, pages 57-62

12. Cappello, F., Etiemble, D., "MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks", Proceedings of Supercomputing, 2000

13. Henty, D.S., "Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling", Proceedings of Supercomputing, 2000

14. Manjikian, N., "Multiprocessor Enhancements of the SimpleScalar Tool Set"

15. Bailey, D.H., Barszcz, E., Dagum, L., Simon, H., "NAS Parallel Benchmark results", Feb 1993

16. Sawyer, W., "FVCCM Parallelization Primer", Oct 1999 http://www.dao.gsfc.nasa.gov/NASanCAR/primer/

17. Rauber, T., Gudula, R., "Modeling the Runtime of Scientific Programs on Parallel Computers", Proceedings of the International Workshop on Parallel Processing, Aug. 2000, pages 307-314