Simulation of Modern Parallel Systems: A CSIM-Based Approach

Dhabaleswar K. Panda, Debashis Basak[*], Donglai Dai, Ram Kesavan, Rajeev Sivaram, Mohammad Banikazemi and Vijay Moorthy

Department of Computer and Information Science
The Ohio State University
Columbus, OH - 43210-1277

July 16, 1997

ABSTRACT

Components of modern parallel systems are becoming quite complex with many features and variations. An integrated modeling of these components (interconnection network, messaging layer, programming model, and computation-communication characteristics of applications) is essential to derive design guidelines for next generation parallel systems. Most of the current simulation-based modeling platforms do not support such integrated modeling. This paper presents our effort at The Ohio State University towards integrated modeling of parallel systems. Basic features of our CSIM-based *Wormhole-routed Multiprocessor Simulator* (WORMulSim) are outlined. A set of techniques used in our simulator to model different network components (such as switches, links, wormhole/cut-through switching techniques, routing protocols, network interfaces), messaging layer with basic communication primitives, distributed shared memory programming model, and computation-communication characteristics of applications are presented. Some sample performance measures of our simulator on current generation workstations are reported to demonstrate the feasibility of integrated modeling with low computational overhead.

[*]Currently with Fore Systems, Inc., Warrendale, PA, after graduation.

Contents

1 INTRODUCTION

Design of a parallel system involves the integration of several hardware and software components. The overall performance of these systems is dependent on the interaction between these components as well as on the computation-communication characteristics of applications. Since these components interact at an instruction level, modeling and evaluation of these components by simulation has been a standard practice before building a parallel system.

For the purpose of modeling, a parallel system can be divided into four major layers: 1) the interconnection network layer, 2) the messaging layer with basic communication primitives, 3) the programming model layer, and 4) the application layer.

Traditionally, there have been two schools of thought on modeling the above layers. Under the first school of thought, the interconnection network layer is modeled in great detail (topology, switching technique, flow control, buffering, etc.). These models are then evaluated in isolation using synthetic traffic such as uniform and hot-spot traffic, which are generated based on probabilistic models. These studies model network contention/congestion accurately. The performance of the interconnection network is typically evaluated using two measures: latency vs. throughput and sustained load vs. applied load. However, these studies typically ignore layers 3 and 4. These studies also ignore the *cause-effect* relationship between messages in a network – an important factor in the execution of parallel programs. Thus, this approach is not suitable for evaluating the overall performance of an application on the parallel system being designed.

The second school of thought focuses on modeling layers 3 and 4 in great detail. However, these models use very simplistic assumptions about the interconnection network and messaging layers. These simplistic assumptions ignore contention/congestion inside the network as well as at the network interfaces. Thus, these evaluations ignore important interaction between computation and communication steps.

Modern interconnection networks and network interfaces are becoming very sophisticated with a lot of features and flexibilities. Designs of current generation parallel systems show that layers 2 and 3 need to be supported in an extremely efficient manner on a parallel system (supporting either distributed memory or distributed shared memory paradigms) to achieve low-latency and high-bandwidth communication. Thus, it is critical that all four layers be modeled in an integrated manner to accurately estimate the overall execution time of an application on a proposed parallel system. We at OSU realized this need a few years ago and took on the challenge of modeling parallel systems in an integrated manner.

In order to implement integrated modeling, a simulation environment is needed where actual applications (in user-level languages like C or Fortran) can be executed on the simulated model of the system encompassing layers 1-4. Such an environment should be capable of producing application results during execution as well as providing user-level and system-level performance numbers (such as overall execution time of an application, time elapsed between two points in the execution, average latency of messages, and load on the network).

A *process-oriented* simulation package like CSIM (Schwetman, 1988) is ideal for such integrated modeling of a parallel system. Over the last five years, we have used such an approach in our *Parallel Architecture and Communication* (PAC) group[1] at The Ohio State University. In this paper we present our techniques and methodologies for implementing such an integrated modeling of parallel systems using CSIM.

We first provide an overview of our *Wormhole-routed Multiprocessor Simulator* (WORMulSim) and introduce its basic features. Next, we illustrate a set of techniques and methodologies which can be used with CSIM structures and processes to model the components of the four layers (interconnection network, messaging and basic communication primitives, programming model, and application). Finally, some performance measurements of our simulator on current generation workstation platforms are reported.

The paper is organized as follows. Section 2 outlines the basic feature of WORMulSim. Sections 3-6 present techniques to model the four layers. The performance measures are presented in Section 7. Concluding remarks and future work are presented in Section 8.

2 OVERVIEW OF WORMulSim

In this section we present an overview of WORMulSim. This simulator is designed with modularity in mind, i.e., various different models of interconnection networks, messaging layers, programming models, and application models are available to the user. New models for the above layers can also be easily incorporated into the simulator. Fig. 1 shows an overview of WORMulSim, and the various layers of the integrated modeling approach.

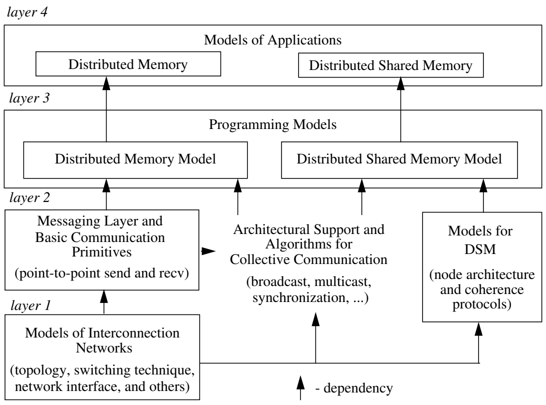

Figure 1: Overview of WORMulSim.

At the lowest layer, the simulator models a wide range of interconnection networks (Duato et al., 1994). Techniques to model these components using CSIM are presented in the following section. Flexibility exists for specifying the following architectural parameters in order to configure an interconnection network according to an architect's choice: network resources (number of switches/routers, number of physical/virtual channels per link, and number of injection/consumption channels per node), topology (regular/clustered/irregular interconnection), choice of switching technique (wormhole/buffered wormhole/virtual cut-through), and routing scheme (deterministic/partial adaptive/fully adaptive). The simulator also provides flexibility

[1] http://www.cis.ohio-state.edu/~panda/pac.html

to specify a range of timing parameters for the network: routing delay (t_{route}), switch delay (t_{sw}), time to transfer a flit[2] across a physical channel (t_{phys}), and injection/consumption time per flit (t_{inj}, t_{cons}).

The next layer models message-passing mechanisms and basic communication primitives. Two major types of *point-to-point* communication primitives are supported: asynchronous non-blocking *send* and blocking *recv*. Flexibility exists to support packetization at the network interface level. A user can specify the following parameters: message length, maximum packet size, communication overhead at the sending and receiving hosts per message ($t_{host}^{send}, t_{host}^{recv}$), and communication overhead at the sender/receiver network interface per packet ($t_{nic}^{send}, t_{nic}^{recv}$). Details of this layer are described in Section 4.

In addition to the point-to-point communication primitives, the simulator also supports *collective communication* as defined by the MPI standard (MPI Forum, 1994). A lot of current research has focused on providing communication and architectural support as well as on designing efficient algorithms for achieving low-latency collective communication. Many of these architectural supports and the associated algorithms are implemented in WORMulSim to provide user-level collective communication primitives (broadcast, multicast, complete exchange, etc.). However, in order to keep this paper focused, we do not discuss these issues in this paper.

The next layer implements two popular programming models: distributed memory and distributed shared memory (DSM). The distributed memory programming model directly uses the basic messaging layer and the send/recv communication primitives. However, supporting the DSM model requires additional architectural support in a parallel system. WORMulSim supports a wide-range of models for evaluating hardware cache-coherent DSM systems (popularly known as CC-NUMA systems). These systems are designed with integrated node architecture (processor, memory, cache, directory, and specialized network interfaces) and cache-coherency protocols (Stenstrom, 1990). Flexibility exists in the simulator to configure the organization of the integrated node architecture (such as the size of memory/cache, access time of memory/cache, number of request/reply queues between processor-cache-memory-directory-network, overhead associated with these queues), to select suitable cache coherency protocols (FIFO/non-FIFO, fully/limited), and to select suitable consistency semantics (sequential/release). The details of how we model these components are presented in Section 5.

Modeling computation-communication characteristics of applications is very important in the integrated modeling approach. Since our simulator is designed using CSIM, layers 1-3 provide a SPMD (Single Program Multiple Data) interface to the applications. This provides the ability to directly run distributed memory and DSM programs written in the SPMD style on WORMulSim. The computation blocks of the programs are instrumented to reflect the actual latency incurred while executing these blocks on a processor. Flexibility exists for specifying the clock rate of the processor. Thus, WORMulSim provides very accurate estimation of the execution time of an application. These issues are discussed in detail in Section 6.

[2] A unit of data transfer on which the flow control is used. It is typically 1-4 bytes in current generation networks.

3 MODELING INTERCONNECTION NETWORKS

We now describe the components of the interconnection layer in a parallel system and the method that we adopt to model it using CSIM.

3.1 Network Resources

CSIM (Schwetman, 1988) is a process-oriented simulation package. It allows multiple processes to be created, and manages the mutually exclusive sharing of resources by these processes. This is done by a CSIM feature called a *facility*. A process can *reserve* a facility for any number of clock ticks and subsequently *release* it. Another useful feature of CSIM is an *event*. This corresponds to a signal that a process can *set* or *wait* upon. We use these two basic CSIM features to model most of the network resources in WORMulSim.

A typical parallel system consists of hosts connected together through an interconnection network. In such systems, the job of sending and receiving messages is either completely handled by the host processor, or a network interface card (NIC) processor handles some part of the job. Each host processor and NIC processor corresponds to a facility in WORMulSim, which allows various processes to reserve these processors in a mutually exclusive manner to model their computational requirements. Each injection and consumption channel of a host also corresponds to a facility which can be reserved and released by outgoing and incoming packets from and to the host.

The interconnection consists of a set of physical links and routers (or switches). By modeling each physical link as a facility, WORMulSim ensures that no two packets traverse a physical link at the same instant, which is typical of parallel system interconnects. In most modern networks, to improve throughput, each physical link has multiple virtual channels multiplexed on it. Before a message can traverse a physical link, it needs to reserve one of the virtual channels corresponding to that physical link. Again, each virtual channel is modeled as a facility in the simulator to capture the above property.

In summary therefore, each host processor, NIC processor, injection channel, consumption channel, physical link and virtual channel is modeled as a facility in the simulator, and signals in the system are are modeled as events.

We now describe the method adopted for modeling the various components of message communication in the network, viz. the switching technique and the routing mechanism. In the following, we use the terms switch and router interchangeably.

3.2 Switching Techniques

WORMulSim models various forms of cut-through switching: wormhole switching, buffered wormhole switching, and virtual cut-through switching. Under cut-through switching, a router can begin forwarding an arriving packet once the (relevant portion of the) packet header has arrived. Once the packet routing decision is made and the (relevant portion of the) packet header is forwarded to the appropriate output port, the remaining packet flits are forwarded one after another to the same port. Thus, arriving packet flits can be routed in a continuous pipelined stream through the network.

The various forms of cut-through switching differ in the number of flits that can be buffered at the switch if a required output port cannot be obtained (because it is has been reserved by another packet, i.e. due to *link contention*). Under wormhole switching a single flit can be buffered at a switch, under buffered wormhole switching a few of the packet flits can be buffered at a switch (without the guarantee that the entire packet can be buffered), and under virtual cut-through switching, the entire packet is guaranteed to be buffered at the switch.

Although WORMulSim models all these forms of cut-through switching, in this paper we focus only on the methods adopted for modeling wormhole switching. The methods adopted for modeling buffered wormhole and virtual cut-through switching are variations of this.

Modeling Wormhole Switching: *Worms* (packets in transit through the network) are modeled as CSIM processes, one process representing the movement of the worm header and the other processes modeling the flow of the remaining worm flits.

The process modeling the worm header movement traverses the path from the source to the destination reserving virtual channels along the path. If system links have only one virtual channel, this has the effect of reserving all the links on the path from the source to the destination. For systems supporting *destination based* routing, the path from the source to the destination can be obtained at every router in the worm's path by calling a routing function (see Sec. 3.3 below). Delays are introduced to model such routing overhead at the routers/switches. Alternatively, for systems supporting *source based* routing, the path is pre-computed and carried with the header. The process modeling the worm header movement extracts information from this pre-computed route to choose an output link at the routers on its path.

Once the header reserves a link at a router and proceeds ahead, it creates a tail process for that router. Thus, one tail process is associated with every router on the worm's path. Each tail process takes care of transferring arriving data through the router and over the next link to the succeeding router whose tail process repeats a similar function. These tail processes collectively transfer the data flits of the worm over the reserved virtual channels on the path to the destination. They also have the function of releasing the virtual channels once all of the data flits have been transferred over them so that they may be used by other worms. Since flits are pipelined through the network, a tail process can begin transferring a flit as soon as the tail process corresponding to the previous router has delivered it. Thus, there is a coupling among the successive tail processes that has to be modeled. We model such coupling by using an event known as *xfer-done* associated with every router. Once a flit has been transferred by a tail process through its router and the next link, it sets the *xfer-done* event corresponding to the next router. The tail process corresponding to the next router can then begin transferring the arrived flit to the succeeding router and so on.

Every router can buffer a single flit at the input port under wormhole routing. When an output channel is acquired, the flit can be transferred to a similar one flit buffer at the next input port, thereby freeing the buffer at the input port for the next flit. When a worm header cannot acquire the channels that it requests, it remains blocked at the input port buffer and the remaining worm flits are also blocked in place since they find no available buffers at the input ports of the succeeding routers. To model this, we use an event called *buff-empty* associated with every router. The *buff-empty* event is set as soon as a flit is transferred from an input to the output buffer of a switch. A tail process transfers a flit from the input buffer to the output buffer only when the *xfer-done* event corresponding to its router has been set. Once this transfer is made, the tail sets the *buff-empty* signal corresponding to its router. It then transfers this flit from the output buffer to the input buffer of the next router when the *buff-empty* event corresponding to the next router has been set. The tail process then sets the *xfer-done* event corresponding to the next router Fig. 2 shows some of the above described events. If a worm header is blocked, a *buff-empty* event is not set by the process modeling the worm's header. This has a cascading effect causing all the tail processes to block, thereby modeling the blocking of the entire worm.

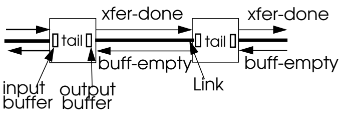

Figure 2: Interaction between the processes that model a worm.

3.3 Routing

As mentioned above, some mechanism is required for the header process to make its way to the destination. Many routing algorithms have been proposed for this. A routing algorithm provides the next link that a worm should take, given the worm header and the link on which the worm arrives at a router as input. Various routing algorithms have been proposed in the recent years (Duato et al., 1997) and WORMulSim supports a number of them (such as Dally's e-cube, Glass and Ni's turn model, Chien and Kim's planar adaptive, Duato's fully adaptive, and Autonet's up*/down* routing algorithms). Any (new) routing algorithm can be plugged in to work with WORMulSim because of the modular fashion in which the routing algorithm is integrated into the rest of the simulator.

Some machines, such as the IBM SP2, support source based routing where the worm header carries a pre-computed route. WORMulSim also supports such routing. In addition, some switch-based parallel systems like DEC Autonet use static routing tables within the switch to provide routing information. WORMulSim also provides flexibility to create such routing tables.

3.4 Topologies

Topologies of parallel systems can be broadly divided into two classes. The first class consists of router-based regular topologies such as k-ary n-mesh, k-ary n-cube, k-ary n-cube cluster-c, etc. The second class consists of switch-based topologies ranging from unidirectional/bidirectional multistage interconnection networks (MINs) to irregular networks. WORMulSim allows the modeling of any of these networks, as described below.

The simulator, during its initialization phase, runs a procedure which allocates the previously described facilities and events corresponding to the various network resources. Once this is done, a procedure is called to configure the links and routers into a regular or irregular topology. For regular topologies, routines can be called that set up most of the commonly used interconnections (including various forms of the k-ary n-cube such as the multidimensional meshes and tori, and various types of unidirectional and bidirectional MINs, such as the omega, butterfly, and cube). To model an irregular topology, data structures are set up to model the interconnection. These data structures can be set up with the use of a random number generator to give us randomly generated irregular topologies. Alternatively, values can be read from a file, thus providing the user with the flexibility to specify a particular topology.

Any new topology can be modeled by plugging in an appropriate module that sets up the interconnections among the routers and among the hosts and routers. This is again a result of the modular design of WORMulSim.

4 MODELING MESSAGING LAYER

In the previous section, the process of transmitting a message through the wormhole routed interconnection network is discussed. In this section, we describe how the overhead associated with preparing and transmitting messages to the network at the sender side is modeled. We also describe how a message is absorbed at the receiver side and delivered to the receiving host processor with the associated overhead. The details on the implementation of the two basic communication primitives (asynchronous nonblocking *send* and asynchronous blocking *recv*) are discussed. Further, we consider the effect of packetization in systems where there is a constraint on the maximum packet size being transmitted and indicate how we model this in WORMulSim.

4.1 Accounting for Messaging Overheads

A message being sent using the *send* primitive needs to be first transferred from the user space of the host processor to the kernel. The message is then transferred to the NIC and then to the network via an injection channel. The overhead of adding the appropriate header to the message, context switching and copying of the message from the user space to the kernel is modeled by holding the host processor resource for t_{host}^{send} units of time. The time required for processing the message at the NIC is modeled by holding the NIC resource for t_{nic}^{send} units of time. In addition to these, the time required for injecting a message flit to the network through an injection channel is modeled by t_{inj}.

Similarly, at the receiving end, the delay introduced by the consumption channel is modeled by t_{con} time units per flit. The processing times at the NIC and host processor are modeled by t_{nic}^{recv} and t_{host}^{recv}, respectively. All messages arriving at a processing node are posted to its mailbox and the host processor is signaled by setting an *mb_recv* event. It should be noted here that although we accurately model the transmission of a message from one node to another, we do not overload the simulator by sending actual data. Only a pointer to the data is passed to the receiving node.

A host processor receives a particular message by invoking the *recv* primitive. The receive mailbox is checked for the particular message. If the message is found in the mailbox, it is forwarded to the receiving host processor which continues with normal program execution. Otherwise, the host processor is blocked until the message is posted by the network into its mailbox because the *recv* primitive is a blocking receive. On arrival of each message the host processor is awakened (by *mb_recv* signal) to check if the desired message has arrived. On arrival of the specified message the host processor is allowed to continue normal execution.

4.2 Packetization

In most current systems the size of packets being transmitted through the network can not be arbitrarily large. Therefore, it is necessary to partition a message longer than the maximum packet size into a set of packets. WORMulSim allows the flexibility to set such maximum packet size parameter and to partition longer messages into smaller packets. Each packet has a sequence number and the number of packets for each message is recorded. On arrival of each packet the NIC processor is signaled (through an *mb_nic_recv* event). The NIC processor is responsible for reassembling the packets and signaling the host processor (through the *mb_recv* event) when all packets of a message arrive.

5 DISTRIBUTED SHARED MEMORY PROGRAMMING MODEL LAYER

So far, we have described how WORMulSim models the interconnection networks and basic communication primitives. Programs using distributed memory model can be directly executed by using these communication primitives. However, for systems supporting the distributed shared memory (DSM) programming model, more architectural components are needed. In this section, we briefly describe how the DSM model has been incorporated into WORMulSim. We focus on the modeling of two essential aspects of a DSM system: node architecture and cache-coherence protocol.

5.1 Node Architecture

In addition to the network, a key factor in modeling a DSM system is to capture the architecture and the operations of every processing node in sufficient detail. The node architecture modeled in WORMulSim, as shown in Fig. 3, is similar to the Stanford FLASH system. Each node in the system consists of a processor, a cache, a portion of global memory, a node controller, and a network interface. Transaction buffers for resolving outstanding memory requests are also modeled to support various relaxed memory consistency semantics.

The processor executes the instructions of a user program and directs all shared memory accesses to the cache. When there is a cache miss, a request is sent to the local node controller. Depending on the current state of the cache/memory block and the underlying cache coherence protocol, different actions are taken by the node controller to satisfy the request (see following subsection). While an outstanding request is being resolved, the processor at the requesting node can either proceed or suspend its execution depending on the memory consistency model supported, data dependency and the depth of the transaction buffers.

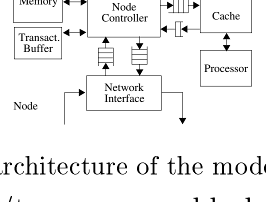

Figure 3: Node architecture of the modeled DSM system.

All types of storage (caches lines/tags, memory blocks/directories, transaction buffers, and transmitting/receiving buffers at the network interface) are implemented as arrays in WOR-

MulSim. Potentially overlapped operations via multiple ports of these storage units (like dual ports in the cache) are modeled by fixing the maximum simultaneous requesters allowed, associating a separate lock at a finer granularity of the storage (like a cache line), and assigning different priorities to the requesters. The functional units such as the cache controller, the node controller, and the network interface are simulated as concurrent CSIM processes. Intra-nodal communication among these units is simulated using queues coupled with event signals.

5.2 Cache-Coherence Protocol

The actions taken by a node in dealing with a shared memory access are dictated by the cache coherence protocol used. Specifically, the cache/node controllers manipulate the corresponding tags/directories of the memory blocks and exchange messages between the nodes to enforce the designated memory consistency semantics. Current generation DSM systems support a range of coherency protocols (such as fully-mapped, limited) (Stenstrom, 1990). WORMulSim models these protocols.

In WORMulSim, a cache coherence protocol is enforced by the processes of the cache controller and the node controller via dispatching tables and a set of transaction handlers. These handlers are invoked based on the combination of the current states of cache lines or memory blocks and the types of messages being processed. These techniques provide us with the desired flexibility with respect to the selection of coherence protocols.

6 MODELING APPLICATION LAYER

WORMulSim can be driven by either synthetic or real application programs. It is commonly known that the overall execution time of any parallel application contains two portions: computation time and communication time. To estimate the communication time, we use the message passing primitives in distributed memory programs and global memory accesses and synchronization operations in DSM programs. The time delays for these operations are determined by WORMulSim in a dynamic and integrated manner during execution, as described in earlier sections.

The computation time of each application is estimated as follows. We first compile the source code of each application by the *DLXcc* compiler (Hennessy, 1996) using the optimization level equivalent to O2 of *gcc*. Next, we examine the generated object code instruction by instruction on each processor and instrument the program (by inserting instructions in each basic block of the code) to count the instructions executed between two successive communication operations. The computation time between two successive communication operations is now estimated as a product of the instruction count and the CPI (cycles per instruction) of the processors of the target system. CSIM *hold* primitives with these computation times are now inserted into the basic blocks of the code. Using such an approach, WORMulSim maintains the temporal relationships between computation-communication steps and models the behavior of an application accurately.

7 PERFORMANCE MEASURES OF WORMulSim

In this section we present some sample performance evaluation results for WORMulSim. Two types of results are presented: communication patterns for distributed memory systems (*one-*

to-all, *all-to-one*, and *multiple-multicast*) and application-driven evaluation for two DSM applications. The pattern-based evaluations use layers 1-2 of the simulator where as the application-driven evaluations consider all four layers.

The results were obtained by running the simulator on an 99 MHz HP 735 with HP-UX 9.05 operating systems. Table 1 shows the execution time of the simulator to implement *all-to-one* and *one-to-all* communication in an 8x8 distributed memory system. The patterns are implemented with point-to-point *send* and *recv* primitives. In the *all-to-one* pattern 63 nodes send individual messages to node 0. Communication takes place in the reverse order for the *one-to-all* pattern.

Table 1: Simulation times for *one-to-all* and *all-to-one* communication patterns on an 8x8 system.

	One-to-all	*All-to-one*
Message Length (in bytes)	Time (in seconds)	Time (in seconds)
4	0.17	0.24
64	2.16	2.71
256	8.30	10.30
1024	33.42	43.27

Table 2 presents simulation times for evaluating *multiple-multicast* pattern. A set of sources participate in this pattern where each source sends a message to a set of destinations. Number of destinations and message size were fixed at 32 and 4 bytes, respectively in this experiment. A binomial tree-based algorithm was used for this pattern. Based on these results as well as the results in Table 1, it can be observed that various communication patterns in distributed memory systems can be evaluated extremely fast using WORMulSim.

Table 2: Simulation times for *multiple-multicast*.

No. of Sources	1	16	32	64
Time (in seconds)	0.40	0.57	1.30	2.81

The next set of results show the time needed to evaluate the execution of complete applications (at an instruction-level) on an 8x8 DSM system using WORMulSim. Two applications (*Barnes* and *Radix*) were considered. *Barnes* is an application, representative of the class of hierarchical N-body methods used in astrophysics, electrostatics, and plasmaphysics, among others. The communication pattern in Barnes is hierarchical and irregular. *Radix* is the integer radix sort kernel, a major task in data base applications. Table 3 shows the simulation times for these two applications. These data points indicate that integrated modeling of parallel systems by considering all four layers is feasible by using current generation workstations.

Table 3: Simulation times for two DSM applications.

Application	Size	Time (in seconds)
Barnes	8K bodies	22560.43
Radix	1M Keys	49597.49

8 CONCLUSIONS AND FUTURE WORK

In this paper we have presented an integrated modeling approach for evaluating parallel systems design. This approach considers four layers of a parallel system: interconnection network, messaging layer, programming model, and application. An overview of our simulator testbed, implementing the integrated modeling approach, together with a set of techniques for modeling different components in CSIM are presented. Sample execution times for evaluating communication patterns and applications on our simulator are reported. These measures demonstrate that simulation-based integrated modeling and evaluation of parallel systems design is feasible on current generation workstations.

Many new developments are currently taking place in parallel system design with respect to interconnection networks (pipelined flit transfer on links, central-buffer organization in a switch, stop-and-go flow control), messaging layer (active messages, get/put), and programming model (multithreading). We plan to enhance WORMulSim so that parallel systems design with these new features can also be modeled and evaluated.

REFERENCES

Duato, J., S. Yalamanchili, and L. Ni. 1997. Interconnection Networks: An Engineering Approach. *The IEEE Computer Society Press.*

Hennessy, J. and D. Patterson. 1996. Computer Architecture: A Quantitative Approach. 2nd Ed. *Morgan Kaufmann.*

MPI: A Message-Passing Interface Standard. Mar 1994. *Message Passing Interface Forum.*

Schwetman H. D. 1998. Using CSIM to Model Complex Systems. In *Proceedings of the 1988 Winter Simulation Conference*, 246–253.

Stenstrom P. June 1990. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23(6):12–24.

ACKNOWLEDGMENTS

This work was supported in part by NSF Grant MIP-9309627, NSF Career Award MIP-9502294, an Ohio State University Presidential Fellowship, and an IBM Cooperative Fellowship. Thanks to several other students who participated in developing different components of this simulator testbed over a period of time. They are Shobana Balakrishnan, Vibha Dixit-Radiya, Sandeep Gupta, I-Lang Hour, Scott King, T.-H. Lin, Raghu Machiraju, Pradeep Prabhakaran, Po-Wen Shi, Sanjay Singhal, Shifang Sun, and Yu-Chee Tseng.

AUTHOR BIOGRAPHIES

DHABALESWAR K. PANDA is an Associate Professor in the Department of Computer and Information Science at The Ohio State University. He received his Ph. D. in computer engineering from the University of Southern California in 1991. His research interests include parallel computer architecture, wormhole-routing, interprocessor communication and synchronization, networks of workstations, clustered systems, and high-performance computing. Dr. Panda is a 1995 recipient of the NSF Faculty Early CAREER Development Award. He can be contacted by e-mail at `panda@cis.ohio-state.edu`.

DEBASHIS BASAK received the B.Tech degree in Computer Science and Engineering from the Indian Institute of Technology, N. Delhi, India, in 1991 and the M.S. and Ph.D. in Computer Science from The Ohio State University, Columbus, USA, in 1992 and 1996, respectively. He is currently part of the ATM Switch Design Group at FORE Systems, Pittsburgh, USA. His research interests include inter-processor communication, interconnection networks, and ATM switch design. Email: `dbasak@fore.com`.

DONGLAI DAI received an M.S. degree in Computer Science from The Ohio State University in 1994. He is currently a Ph.D. candidate at OSU. His research interests include design and performance evaluation of DSM systems with emphasis on communication, synchronization, and coherence protocols. Email: `dai@cis.ohio-state.edu`.

RAM KESAVAN received a B.Tech. degree in Computer Science & Engineering from the Indian Institute of Technology, Madras in 1993, and an M.S. degree in Computer Science from The Ohio State University in 1994. He is currently a Ph.D. candidate at OSU. Email: `kesavan@cis.ohio-state.edu`.

RAJEEV SIVARAM received a B.Tech. degree in Computer Science & Engineering from IT-BHU, Varanasi in 1993 and an M.S. degree in Computer Science from The Ohio State University (OSU) in 1994. He is currently a Ph.D. candidate at OSU. Email: `sivaram@cis.ohio-state.edu`.

MOHAMMAD BANIKAZEMI is a Ph.D. student in the Department of Computer and Information Science at The Ohio State University. He received a B.S. degree in Electrical Engineering from Isfahan University of Technology, Iran and an M.S. degree in Electrical Engineering from The Ohio State University in 1996. Email: `banikaze@cis.ohio-state.edu`.

VIJAY MOORTHY received a B.E. degree in Computer Science & Engineering from Regional Engineering College, Trichy, India in 1996. He is currently an M.S. student at The Ohio State University. Email: `moorthy@cis.ohio-state.edu`.