Asynchronous Parallel Simulation of Parallel Programs*

Rajive Bagrodia
(rajive@cs.ucla.edu)
Computer Science Department
University of California
Los Angeles, CA 90024

Sundeep Prakash
(prakash@cs.ucla.edu)
Computer Science Department
University of California
Los Angeles, CA 90024

June 27, 1997

Abstract

Accurate simulations of parallel programs for large datasets can often be slow; parallel execution has been shown to offer significant potential in reducing the execution time of many discrete event models. This paper describes the design and implementation of PROSIM, a library for the execution driven parallel simulation of task and data parallel programs. PROSIM can be used to predict the performance of *existing* programs written in MPI, a message passing library, and UC, a data parallel language. The simulation models can be executed sequentially or in parallel. Parallel executions of the models are synchronized using a set of asynchronous conservative protocols that include optimizations to reduce the strength and frequency of barrier synchronizations by exploiting the communication characteristics of the simulated program. The paper presents validation and performance results for the simulator for a set of applications that include the NAS Parallel Benchmark suite. The techniques described in this paper yielded significant performance improvements in the simulation of parallel programs, and in some cases completely eliminated the synchronizations in the parallel execution of the simulation model.

1 Introduction

Simulators for parallel programs can be used to test, debug and predict the performance of the programs as a function of various architectural characteristics. Two primary classes of simulators have been developed: architectural simulators that typically use a detailed model of one or more component of the parallel architecture (e.g., the memory subsystem or interconnection network) to identify their impact on application behavior, and program simulators that typically use abstract models of the architectural components to evaluate program scalability or algorithmic alternatives. Many of the early architecture simulators were designed for sequential execution[BDCW91, DGH91, CDJ+91]. To reduce the large execution times for such models, the simulators used direct execution, where sequential code fragments were executed rather than simulated. However, even with the use of abstract models and direct execution, sequential program simulators tended to be slow with slowdown factors ranging from 2 to 35 for each process in the simulated program[BDCW91]. Several recent efforts have been exploring the use of parallel model execution[LW96, RHL+93, DHN94, PB95b, CH96] to reduce the model execution times, with varying degrees of success.

In this paper, we describe a parallel simulator for parallel programs using conservative synchronization algorithms. The primary contribution of this paper are as follows:

*This research was supported in part by an ARPA/CSTO Award (No. F30602-94-C-0273)

- This research addresses the problem of simulating data parallel and task parallel programs. Preceding efforts have primarily targeted task parallel programs. The global synchronizations that are implicit in a data parallel program allow a number of optimizations in their simulation.

- We use a novel conservative synchronization protocol that uses variations of existing null message[CM81] and conditional event[CS89] protocols together with communication properties of the model being simulated to significantly reduce the frequency and strength of synchronization in the parallel simulator.

- We present the results of an experimental study to simulate a number of programs including programs from the NAS Parallel Benchmark Suite (NPB 2).

The rest of the paper is organized as follows: In the next section, we present a model for the execution and simulation of message passing programs. We define the criterion for a simulation to be correct and use the model to compare the two commonly used simulation protocols for parallel programs: the synchronous (or quantum) protocol and the asynchronous protocol. This section also describes the PDES protocols used in the simulator. Section 3 presents a brief overview of MPI, and a simulation model for MPI programs. Section 4 describes the simulation of data parallel programs using the simulation model and algorithms described in the previous section. Section 5 contains results on the validation of the simulation model, and the performance of the simulation protocol. Section 6 contains a brief description of related work, and Section 7 contains a summary of the paper and directions for future work.

2 Parallel Program Models

We present a model for the execution and simulation of message passing programs that can be used to reason about the validity and accuracy of a simulation model. The following terms are used in this paper.

2.1 Definitions and Assumptions

- **Target Program**: The message passing program whose performance is to be predicted.

- **Target Machine**: The machine on which the target program executes.

- **Simulator**: The program that simulates execution of the target program.

- **Host Machine**: The machine on which the simulator executes.

2.2 Target Program Model

We assume that the target program uses a shared nothing programming model and contains three types of statements: *local code block* or LCB, send statements, and receive statements. An LCB is the sequence of local statements executed between two message passing statements. We assume that an LCB is deterministic. The send and receive statements provide a buffered communication capability where the send deposits a message in the message buffer at the named destination and the receive statement removes messages from its local buffer. We assume that the receive statemnt can select specific messages from its buffer based on message attributes like $sender-id$, *type*, or tag. We do not postulate a precise syntax or semantics with the preceding statement types in this section; in subsequent section we will apply this model to MPI.

2.3 Execution Model

We assume that the target program executes on the target machine as one process per processor with no dynamic process creation and termination. Messages from one process to another are assumed to arrive received in FIFO order. The input to the target program contains (a) the number of processes to be executed and, (b) the input for each process, i.e. values for the program's uninitialized variables.

At any point in its execution, a process in the target program may be in one of two states: *running* or *blocked*. A process is blocked if it has executed a receive statement and no matching message is available; otherwise it is said to be running. The model assumes that the execution times of local statements and the communication times of messages are non-deterministic. Note that each local statement is deterministic; however due to interrupts, cache behavior etc, the execution time may not be deterministic.

2.3.1 Program Trace

We define the state of a process to be the value of its local variables. The message queue of a process is not considered part of the program's data space. For each process in the program, we define a *process trace* as a sequence that captures the state changes of the process, from initialization to termination. We use PT to represent a process trace; each element of PT is a tuple t_i with three fields:

1. $t_i.seqno$ identifies the statement that is executed.
2. $t_i.state$ is the state of the process before the statement is executed; and
3. $t_i.msg\_rcvd$ is the accepted message if the statement executed is a receive statement. Otherwise, this field is null.

Upon the execution of a send statement, neither the state of the sender process nor the receiver process changes. Upon the execution of a receive statement, only the buffer, into which the accepted message is copied, gets changed. We restrict ourselves to terminating programs and hence process traces are finite sequences. The final term in process trace t_f, contains the final state of the process.

A process is *deterministic*, if it admits of a single process trace. Every execution of a deterministic process will produce the same trace. As the LCB and send statements are deterministic, a process is deterministic if each receive statement is deterministic. A receive statement is said to be deterministic, if the program contains a unique message that matches every execution of the receive statement. In general, the receive statements in a process are non-deterministic; consequently the process is non- deterministic and different executions yield different traces.

The execution of a program P for given input I can be characterized by its *program trace*, which is constructed by aggregating the process traces. We define the program trace $E(P, I)$ to be a tuple, $E = (PT_0, PT_1, \ldots, PT_{N-1})$, where PT_i is the process trace for process p_i in the execution of P, and N is the number of processes in the program. A program is *deterministic*, if each of its constituent process is deterministic; such a program must have a single program trace. Sequential programs are trivially deterministic under the assumptions of our model. In general, a parallel program is non-deterministic and its execution cannot be characterized by a single trace. We use $ES(P, I)$ to refer to the set of all possible program traces that may be generated by the execution of a program P with input I.

2.3.2 Timed Program Trace

The *process trace* captures the sequence of states traversed by the process from initialization to termination. We enhance the process trace to also include the physical time at which each statement in the trace is executed. Formally, let a *timed process trace* be represented by TPT; an element of TPT is a tuple s_i which consists of four fields: $\{s_i.seqno, s_i.state, s_i.msg\_rcvd,\ s_i.time\}$, where the first three elements were defined earlier and the last element $time_i$ is the physical time at which the corresponding statement was executed in some execution of the process. The time may be specified relative to the start time. The *termination time* of a process trace is the execution time of its final statement.

A given process trace can produce multiple timed process traces because the time to execute a given statement, in general, may differ in two different execution of the same program. A process trace can be derived from a timed process trace simply by projecting the latter on to all fields other than *time*. Two timed process traces are said to be *congruent*, if they yield an identical process trace. Note that all timed process traces for a deterministic process are congruent.

The *timed program trace*, TE, of a program is defined analogously to the program trace, where $TE = (TPT_0, TPT_1, \ldots, TPT_{N-1})$, where N is the number of processes and TPT_i is the timed process trace of process p_i. We assume all processes have the same start time. The *execution time* of a program trace is the maximum of the termination times of its constituent timed process traces. Finally, we will use *timed program trace set*, or $TES(P, M, I)$, to refer to the (possibly infinite) set of timed program traces of a program P on machine M for the given input I. We use our notions of program traces to reason about the correctness of a simulation model for parallel programs.

2.4 Simulation Model

The goal of the simulator is to predict the execution time of a target program on a target architecture for given program inputs. We assume that the simulation model defines a logical process (LP) for each process in the target program. Each LP, say lp_i, has a message queue $in - queue_i$ and simulation clock, $clock_i$. The message queue is an artifact of the simulator and is used to store incoming messages as they arrive at the corresponding LP. The simulation model of a parallel program contains three types of events: *local* events that correspond to execution of an LCB in the target program, and *send* and *receive* events that respectively correspond to the execution of a send or receive statement in the target program. Each of the preceding event is simulated as follows:

1. *local* event: The most common method for simulation of a local event is by direct execution: the LP, say lp_i executes the LCB on the host machine, measuring its duration, say t, and advancing $clock_i$ by t. For runtime measurement to have a reasonable degree of accuracy, the host and target processors must be the same (or an appropriate scaling factor must be determined). Even when the host and target processors are the same, other sources of error remain; these are discussed in section 5.4.

2. For a send statement, lp_i computes the communication latency, say l, for the message (or messages, if the send is a collective operation) using a model of the interconnection network. The message(s) are timestamped with the send time (which is simply the value of $clock_i$ when the send statement is executed) and its predicted receive time. We use a simple contention free model to predict the communication latency of a message. In this model, the latency of a message is a function only of its size. As shown in the results section, the simple model yielded fairly accurate results for the benchmarks considered in this paper.

3. For a receive statement, the LP uses some simulation protocol to remove messages from its $in-queue$ in their *simulation timestamp order* rather than the order in which messages are physically deposited in its $in-queue$. When lp_i accepts a message from its buffer, $clock_i$ is set to the larger of the simulation timestamp of the receive statement and the receive timestamp of the accepted message.

The simulator can also produce a timed trace for the target program and thus the predicted execution time of the target program on the specified target architecture. We refer to this as the *virtual* timed program trace.

2.5 Simulation Protocols

Let $ES_1 = ES(P, I)$ be the set of program traces that characterize the execution of a program P with input I. Let TES_1 be the corresponding set of timed program traces produced by executing the program on a machine M. In general, TES_1 will not contain a unique trace. For example, the time to execute an LCB may differ in two executions due to different cache effects or the transmission time for a given message may be different in two separate executions of the program. Even a deterministic program that has a single program trace may have infinite timed program traces, all of which are congruent, and each of which yields a different execution time. If the execution time for the target program on a target architecture is not constant, what does it mean for the simulation of such a (parallel) program to be correct? Consider the following definition of correctness.

Definition 1 : Given a program P that executes on a machine M with input I, the simulator is *correct* if the virtual timed program trace generated by the simulator is congruent with some program trace in $ES(P, I)$.

This property ensures that the simulator does not violate the event dependencies in the target program. In other words, the event order generated by the simulator is one that could be generated in some execution of the program. However, this definition permits the simulator to ignore the specific machine characteristics of the target platform. For instance, consider a program P that has two program traces E1 and E2. Assume that no execution of P on an architecture M can produce trace E1; however the preceding definition allows the simulator to generate a timed trace for P's execution on M that is congruent to $E1$. Thus, the preceding definition is likely to provide a relatively weak definition of correctness for a simulator.

To provide a framework in which we can reason about the accuracy of a parallel program simulator, we must identify a unique timed program trace that the simulator is required to reproduce. The non-determinism in the execution of the target program is due to variances in its execution environment which typically arise from sharing the computing resources with other (user and system) programs. If the execution environment for the program can be held constant, a unique timed trace can be used to characterize the execution of the program on a given architecture. We propose the following two properties for a simulator.

Reproducibility Assumption : Standalone execution of a program on a machine yields a unique timed program trace. [1] This trace is referred to as the *ideal target trace.*

[1] Of course, standalone execution of a program does not guarantee that two executions will yield exactly the same (timed) execution trace. However, our goal here is to select a reasonable execution environment that can be reproduced for the program and for its simulation; other execution environments (including synthetic ones) may also be used for this purpose.

Fidelity Assumption : The simulator can precisely predict the execution time of every LCB and the communication latency of every message in the target program.

Based on the preceding properties, we define what is meant by an *accurate* simulator as follows:

Definition 2 : A simulator is said to be *accurate* if it can reproduce the ideal target trace under the *reproducibility* and *fidelity* assumptions.

Almost all existing parallel program and hardware simulators use one of the following two types of protocols: the synchronous or *quantum* protocol, and the *asynchronous* protocols. In the synchronous protocol, each LP periodically simulates its corresponding process for a previously determined interval Q, termed the *simulation quantum*, and then executes a global barrier. The barriers are used to ensure that messages from remote LPs will be accepted in their correct timestamp order: an LP waiting at a receive will accept a matching message from its buffer only if *the receive timestamp on the message is less than the simulation time at which the current quantum terminates.* If more that one such message is present, the LP will select the one with the earliest timestamp; if no such messages are present, the LP remains blocked, and its simulation time is updated to the end of the current quantum.

Assuming that the simulator satisfies the *fidelity* property, the synchronous protocol will reproduce the ideal target trace only if $Q < L$, where L is the communication latency of the target architecture. However, the frequent global synchronizations necessitated by a small Q can cause the simulator to run extremely slowly such that its practical utility is limited[2]. Simulation efficiency can be improved by using a larger quantum; however with $Q > L$, it is no longer possible to guarantee that the simulator is accurate. Thus parallel simulators (e.g. SimOS[RBDH97]) that use this protocol offer two simulation modes: fast and inaccurate, or slow and accurate.

The alternative is to use asynchronous PDES protocols. Such protocols guarantee that each LP will (eventually) process all events in the strict order of their global timestamps. It follows that subject to the reproducibility and fidelity assumptions, such a simulator will be accurate. As all events are executed in th eorder of their timestamps, any inaccuracies in the predictions will be due only to the degree of accuracy to which a given component like the interconnection network is modeled in the simulation. The specific protocol used in this paper is a conservative protocol referred to as the Accelerated Null Message protocol or ANP: it combines the existing null message and conditional event algorithms together with optimizations that exploit the communication characteristics of an application to reduce the synchronization overheads. For a class of problems (including the NAS MPI benchmarks), they also offer excellent parallel performance. The next section describes this protocol.

2.6 ANP: An accelerated null message protocol

We define the following terms for each LP in a simulation model[JB93]:

Earliest Input Time(EIT): The EIT of an LP is defined to be the lower bound on the timestamp of all future messages that may be *received* by it.

Earliest Output Time(EOT): The EOT of an LP is defined to be the lower bound on the timestamp of all future messages that may be *sent* by the LP.

[2] If the host architecture provides an efficient hardware implementation of global synchronization (e.g., CM5), it might be feasible to obtain good performance even with a small value of Q

Earliest Conditional Output Time(ECOT): The ECOT of an LP is defined to be the lower bound on the timestamp of all future messages that may be *sent* by the LP, assuming that it will not receive any further messages; in other words ECOT is computed as its EOT, assuming its EIT is infinity.

The synchronization algorithm used for parallel simulations essentially computes the EIT for each LP in the model. We use the term *safe message* to refer to a message that is stored in the incoming message buffer, *in-queue*, of the LP and has a timestamp that is smaller than the EIT of the LP. In our simulation model, a blocked LP, say lp_i, can unblock itself, if $in - queue_i$ contains a matching message that is $safe$. The efficiency of a parallel simulation model is directly related to the efficiency with which it can advance its EIT. In the best case, if a message can be identified to be safe as soon as it arrives at an LP, the overheads of the parallel simulation will be small.

In the original null message protocol[Mis86], each LP periodically computes its EOT and sends this to all LPs in the model. The EOT may be transmitted aggressively or using a demand driven scheme. Further, they may either be piggy backed with regular messages or sent as special null messages. An LP updates its EIT to simply be the minimum of the incoming EOT. It is generally more efficient to have each LP maintain its communication topology[BtL94]: the *source set* of an LP is the set of LPs from which it can receive messages, and the *destination set* of an LP is the set of LPs to which an LP can send messages. An LP need only communicate with other LPs in its source and destination set to advance its EIT. The difference between the EOT and current simulation time of a process is referred to as its *lookahead*[Fuj88]. It follows that even if a single LP in the source set of a process has a low EOT, it can significantly hamper the ability of the receiver to identify safe messages.

In our model of a parallel program, each LP executes asynchronously until it reaches a receive statement. At this point, if it does not have any safe messages, it demands updated EOT from all LPs in its source set. On receiving a request, an LP must send a new EOT. In the worst case, the new EOT of an LP may simply be S+L, where S is the current simulation time of the LP and L is the minimum message communication latency for the target architecture. On receiving responses to its request, if the EIT of an LP advances sufficiently to identify a safe message, the LP proceeds with its simulation; if not it initiates another round of EOT updates. If communication is relatively infrequent in a target program, this algorithm may achieve reasonable performance. However, in general, this algorithm has been shown to have poor performance[LW96] because the lookahead of an LP is poor. Alternative schemes that have been proposed and used by other researchers to advance the lookahead of an LP that simulates each process in the target program are discussed subsequently in the related work section. We use a different solution to alleviate this problem - using the conditional event algorithm.

The conditional event protocol (CEP)[CS89] identifies the earliest event in a simulation model by using global information. Using this protocol, an LP computes its EIT as the minimum of the ECOTs of all LPs in the model and the receive timestamps of all messages in transit. In simpler terms, the CEP calculates a global lower bound on the receive timestamp of the next message that will be received by any LP. The primary drawback with this algorithm is that, in the worst case, a separate global computation may be required to identify each safe event.

The Aggressive Null Message protocol (ANP) used by our simulator uses the CEP to advance the EIT of an LP rapidly in situations where a low lookahead might otherwise require many rounds of null message transmissions[JB93]. Using this protocol, the EIT of an LP is calculated as the maximum of the EIT reported by the null message and conditional event algorithms. The ANP can lead to significant reductions in the number of synchronization messages as described in a subsequent section.

A blocked LP requests EOT or initiates minimum ECOT computation, when it does not have safe messages. However, if an LP, say $L1$ is blocked at a deterministic receive statement, it can identify safe messages locally. Recall that exactly one message in the program trace can match a deterministic receive. Thus it is only necessary for such a process to wait until it receives a matching message. As soon as the message is received, it is known to be safe; no null messages are necessary! Of course, if another LP in the model is blocked on a non-deterministic receive, $L1$ must still respond to requests for EOT and ECOT updates, but it will not initiate any such requests, thus reducing the overall traffic and blocking time in the model. Clearly, if no LP in the model is blocked on a non-deterministic receive statement, no synchronization messages will be generated in the model and the parallel simulation can be extremely efficient. Our goal is to automatically or manually identify as many deterministic receive statements in the simulation model as possible. In the next two sections, we explore the application of these protocols and optimization to the simulation of MPI and data parallel programs.

3 MPI Simulation Model

MPI[For93] is a message passing library which offers a host of point-to-point and collective inter-process communication functions to a set of single threaded processes executing in parallel. All communication is performed using a *communicator*, which is simply an identifier associated with a group of processes. Only member processes may use a given communicator. We have developed a simulator for MPI programs that can simulate a substantial subset of MPI programs including most of the commonly used MPI functions.

The simulator can be used to simulate *unmodified* MPI programs. Each program is first passed through a preprocessor that implement necessary transformations as explained next. The preprocessor also replaces all MPI calls by equivalent calls to corresponding routines in the simulator. The simulator does not directly simulate every MPI call. Rather, all collective communication functions are first translated by the simulator in terms of point-to-point communication functions, and all point-to-point communication functions are implemented using a set of core non-blocking MPI functions as depicted in Figure 1. Note that the translation of collective communication functions in the simulator must be identical to how they are implemented on the target architecture. The remainder of this section describes a library based facility to simulate MPI programs.

3.1 Preprocessing MPI Programs

MPI programs execute as a collection of single threaded processes, and, in general, the host machine will have fewer processors than the target machine (for sequential simulation, the host machine has only one processor). This requires that the simulator support multithreaded execution of MPI programs. We have developed MPI-LITE, a portable library for multithreaded execution of MPI programs, for this purpose.

Execution of an existing MPI program as a multithreaded program requires that the permanent variables, (global variables and static variables within functions) be handled separately. If the unmodified MPI program is executed as a multithreaded program, all threads on a given host process will access a single copy of each permanent variable. To prevent this, it is necessary to localize the permanent variable such that each thread has a separate copy. Each permanent variable is redeclared with an additional dimension whose size is equal to the maximum number of threads in a host process. Each reference to the permanent variable is also modified such that each thread uses its id to access its own copy of the permanent variable. This process of adding an additional dimension to the permanent variables is referred to as *privatization.*

A preprocessor is provided with PROSIM to support the automatic transformation of an MPI program to an equivalent program that is compatible with the simulator. The preprocessor privatizes permanent variables, converts each call to a MPI function to an equivalent MPI-LITE function, and implements miscellaneous transformations needed to link the target program with the simulator routines. Note that the programmer is not required to make *any* manual changes to the program to ensure compatibility.

3.2 Core MPI Functions

The prototypes of the four core functions are shown in Figure 2. Each of the two send functions compose a message consisting of *count* elements of type *datatype* from the data in the buffer pointed to by buf. The message is tagged with the value *tag* and sent to the process ranked *dest* in communicator *comm*. The argument *request* points to the id of the operation: a process can subsequently use this id to check on the status of the corresponding operation. The primary difference between the two functions is that whereas MPI_Issend is synchronous and completes only when the receiver has accepted the message using a matching receive, MPI_Ibsend completes as soon as the data has been copied to a local buffer. The buffer space is released only when the data has been transmitted to the receiver via a MPI_Issend. If no buffer space is available when the buffered send is initiated, the program is aborted.

MPI_Irecv starts a receive operation; as with the send commands, the corresponding process is not blocked. The operation completes if a message from the process ranked *source* in communicator *comm*, with tag *tag* is available in the buffer. The receive may use the wild card arguments MPI_ANY_SOURCE or MPI_ANY_TAG to respectively indicate that a message from any source process or with any value of *tag* is acceptable. As MPI_Irecv is non-blocking, a process may have several receives that are pending simultaneously. Incoming messages are matched with pending receives using the MPI in-order delivery rules: (a) a message cannot match a pending receive if an earlier matching receive is still pending and (b) a pending receive cannot match a message if an earlier matching message is still unclaimed. A message is said to be earlier if it has an earlier send time. MPI_Wait is simply a function which waits until the (send or receive) operation with id in the variable pointed to by *request* is completed.

3.3 Simulation Model for Core Functions

The simulation model is very similar to the one presented earlier for simple message passing programs. The only differences are (a) at each LP, there is a message queue for each communicator of which the LP is a member, and (b) at each LP, there is an ordered list (ordered by simulation timestamp) of the pending (send and receive) operations of the LP; this list is referred to as the *request list*. Each LP executes the target program, and takes the following actions upon encountering any of the four core calls:

1. **MPI_Issend**: The message (with source, destination, tag, communicator and data) is sent to the receiver LP. It is timestamped with the send timestamp, which is the current simulation time of the LP and the receive timestamp, which is the send timestamp plus the predicted message latency. A corresponding request is queued at the end of the LP's request list.

2. **MPI_Ibsend**: The same procedure is followed as for MPI_Issend, except for three differences: (a) Initially, a *buffer availability check* is performed. A buffer availability check reserves space for the message in the user provided buffer area. Note that in the simulation model, this space does not physically exist; a data structure is used to indicate the portion of the buffer

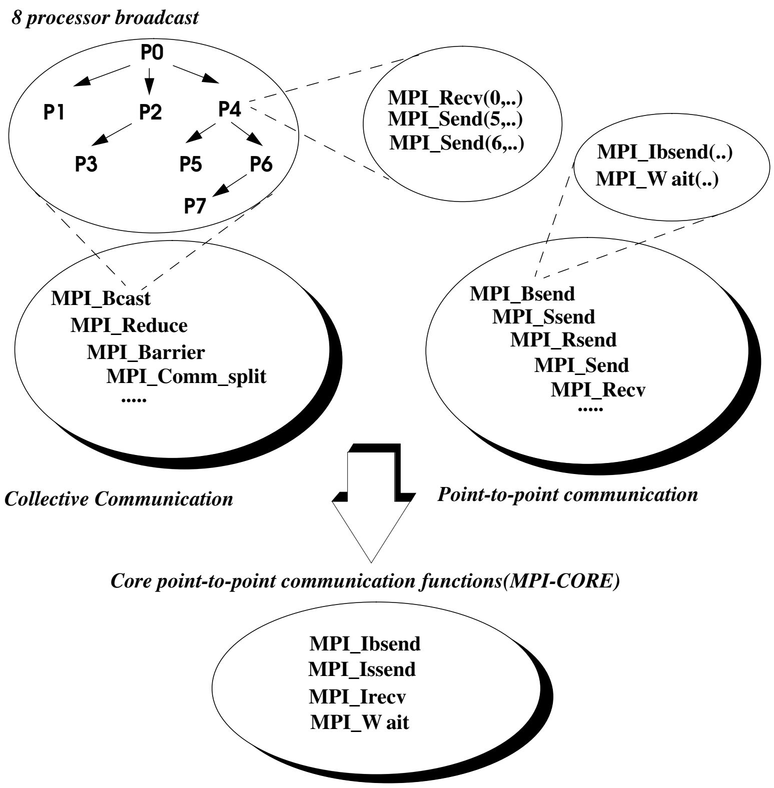

Figure 1: Transformation of MPI to Core Functions

```
int MPI_Issend(void *buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Ibsend(void *buf, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm, MPI_Request *request)

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
int source, int tag, MPI_Comm comm, MPI_Status *status,
MPI_Request *request)

int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Figure 2: Core Communication Functions of MPI

that is currently occupied. If no space is available at that simulation time, the simulation is completed with the report that the program would have aborted at that point due to lack of buffer space. (b) While calculating the receive timestamp of the message, the predicted message latency accounts for the additional copying that would occur in a buffered send. (c) The request queued at the request list is marked as a request from a buffered send.

3. **MPI_Irecv**: A request is queued at the request list. It is marked as a request to receive a message. The source, tag, and communicator of the receive are included in the request, as is the pointer to the buffer where the accepted message should be deposited.

4. **MPI_Wait**: The action taken depends on the type of operation, which is indicated the request id:

   (a) **MPI_Irecv request**: The LP is blocked until a matched message (as described above) is available. At this point, the LP's simulation clock is updated to the maximum of the current simulation time and the receive timestamp of the matching message, an acknowledgement is sent to the sender, and the LP is resumed.

   (b) **MPI_Issend request**: The synchronous send completes only when the corresponding acknowledgement has been received from the destination. At this time, the simulation time of the LP is updated to the maximum of the current simulation time and the receive timestamp of the acknowledgement. The LP is blocked until the send is completed.

   (c) **MPI_Ibsend request**: For a buffered send, the LP is not blocked. The corresponding request stays in the request list until it is satisfied, at which simulation time the corresponding buffer space in the user provided buffer area is released.

The simulation protocol used to synchronize the simulation models must ensure that MPI's in-order delivery rules are obeyed (in simulation time) while matching arriving messages with the request list of an LP. Matching acknowledgements with their corresponding requests requires no such effort by the simulation protocol, simply because there is only one matching request for each acknowledgement.

In the simulation model, each LP executes without synchronizing with other LPs until it gets blocked on a specific request. If the LP is in non-deterministic mode, it sends a request for EOT to all processes in its destination set as explained in the previous section. The simulator computes the EIT of a process on a per *communicator* basis to reduce synchronization costs; in other words, if an LP belongs to more than one communicator, it will define separate source sets and maintain separate EIT to identify safe messages within each communicator. An LP in the deterministic mode simply waits until a matching message is available in its *in-queue* at which point it accepts the message and continues asynchronously with its execution.

4 Data Parallel Programs

Data parallel programming is defined as single threaded, global name space, loosely synchronous parallel computation[Hig93]. We consider the execution of a data parallel program on a multicomputer. The data parallel program is first translated into a message passing SPMD program, perhaps in a language like MPI, which is then compiled and executed on the multicomputer. The operations in the data parallel program that lead to communication and synchronization in the corresponding SPMD program may be classified into the following categories:

- Data Distribution: These are operations which specify the placement and alignment of data (relative to other data or to some template) over a set of processors. Familiar HPF primitives which perform these functions are *align* and *realign* for relative alignment of data, and *distribute* and *redistribute* to partition aggregate data among processor memories.
- Parallel data assignment: These allow parallel operations on sections of arrays with the same shape. In HPF such operations occur in statements like the *forall*, *independent*, and *where* statements, and in array assignment statements.
- Parallel data combination: Data combination occurs in operations like *reductions* which generate a single value from aggregate data. Other operations include prefix, suffix and combining scatter. In HPF, these operations occur as intrinsic operations.

4.1 Compilation of Data Parallel Programs

It is generally possible to compile data parallel programs such that message communications in the resulting SPMD program is deterministic. The data distributions and reductions almost always generate deterministic communications and most commonly used forms of parallel assignments also generate deterministic communications. We illustrate the translation process with two examples using HPF syntax for the data parallel code. First, consider the fragment in figure 3. The declaration $PROCESSORS$ in line 2 describes the rectilinear processors arrangement, which is mapped onto the actual physical processors. For convenience, we assume that this is the same as the number of physical processors. The $DISTRIBUTE$ directive (line 3) specifies the mapping of the data on the processor arrangement. This directive distributes array a onto the processors using a simple block mapping, where $a[i]$ is stored on, or *owned* by, processor i. The $FORALL$ statement (line 4) describes a synchronous parallel assignment, where each element of array a is shifted to the left. The target code generated by this translation is shown using MPI syntax in the bottom of the figure. In the translated code, each process of the SPMD program will be assigned a single element of array a based on the specified data decomposition. We refer to this element as $a\_element$; for simplicity, we omit the code to implement the data decomposition. The MPI code in lines 3 and 4 specifies the simple communication that is necessary to implement the parallel assignment. Clearly the message communication in the translated program is deterministic and the form of determinism is such that it can easily be recognized automatically by the simulator. For brevity, we use * to represent MPI_ANY_SOURCE and MPI_ANY_TAG, and *world* to represent MPI_COMM_WORLD in the program fragments.

We now consider a different version of the parallel assignment as shown in line 4 of Figure 4. The primary difference in this type of assignment is that the *owner* process for a given element of array a cannot locally determine the destination processor that needs this value and hence a simple deterministic send and receive pair cannot be used to implement this fragment in the translated code. Instead, each process must first compute $b[i]$ which is the address of the source process, send a *request* message to the corresponding owner process, and wait to receive the value. In general, as the number of requests that can be generated for a given element of a, and hence the number and relative order of message arrivals at a given process, is not known, the resultant code will be non-deterministic[PB95a].

We present two possible compilations for this fragment. The first method is perhaps less efficient but produces deterministic code: broadcast the entire array and allow each process to select the specific element that it needs. To broadcast the entire array will require a sequence of broadcasts, where each process, in turn, transmits the elements that it owns to all other processes. This version is shown in lines 5-14 of figure 4. As a broadcast is implemented in MPI as a deterministic

Source Code:

```
        INTEGER a(0:7)
!HPF$   PROCESSORS procs(0:7)
!HPF$   DISTRIBUTE (BLOCK) ONTO procs:: a
        FORALL (i=0:7) a(i) = a(MOD(i+1,8));
```

SPMD code for FORALL statement for processor i (C syntax):

```
int a_element, temp;
MPI_STATUS status;
/* send local value of a to left neighbor */
MPI_Send(&a_element, 1, MPI_INT, (i-1)%8, 0, world);
/* wait to receive a value from right neighbor */
MPI_Recv(&temp, 1, MPI_INT,(i+1)%8, *, *, world, &status)
a_element=temp;
```

Figure 3: Deterministic Data Parallel Program Fragment

operation, the compiled code will be deterministic. A non-deterministic alternative is shown in lines 15-25 where each process asynchronously sends its local elements to all other processes, receive the elements in a non-deterministic order, and selects the one that it needs. MPI_RECEIVE provides an argument *status* that contains additional information for the incoming message, including the id of the message sender.

Other implementation alternatives are likely for the preceding program fragments and it is not appropriate to assume that compilers for all data parallel programs always generates deterministic code. However, data parallel code fragments with unpredictable communication patterns (like the program in Figure 4) are relatively uncommon and could, in any case, be compiled using deterministic communications for simplicity. Our simulator is designed to handle the non-deterministic case, and exploit the deterministic case when it can be detected.

4.2 Simulation of Data Parallel Programs

The compiled program can be simulated much as the MPI programs described in the previous section, where the simulator attempts to automatically detect deterministic code fragments and incur the larger synchronization overhead only when simulating non-deterministic receive statements.

A generalization to deterministic receives is also possible: assume that a given receive is not deterministic, in that more than one message in the system can potentially match the receive statement executed by the LP. For example the MPI receive statement in line 24 of the translated code in Figure 4 is not deterministic as it can match messages sent by multiple processes. Even for such fragments, it is possible to exploit knowledge of the communication patterns in the target application to avoid a global barrier. Again consider the forall statement in Figure 4 and its non-deterministic SPMD code. The compiler for the data parallel language can easily deduce that exactly 8 messages will be received by each process in the translated code; we refer to this as the *expected-message-set*.

An LP may use the expected-message-set to further reduce its synchronization overheads in

Source Code:

```
        INTEGER a(0:7), b(0:7)
!HPF$   PROCESSORS procs(0:7)
!HPF$   DISTRIBUTE (BLOCK) ONTO procs:: a,b
        FORALL (i=0:7) a(i) = a(MOD(b(i),8));
```

Deterministic SPMD code for FORALL statement for processor i (C syntax):

```
int a_element, b_element, i, source, temp, new_a;
source = b_element%8;
for (i=0;i<8;i++){
    /* myrank is the unique rank for this process */
    if (i==myrank) temp=a_element;
    MPI_BCAST(&temp, 1, MPI_INT, i, world);
    if (i==source)new_a=temp;
}
a_element=new_a;
```

Non-deterministic SPMD code for FORALL statement for processor i (C syntax):

```
int a_element, b_element, i, source,temp, myrank;
MPI_Status status;
source = b_element%8;
for (i=0;i<8;i++)
    /* send local a element to all other processes*/
    if (i != myrank)
        MPI_SEND(&a_element, 1, MPI_INT, i, 0,world);
/* Accept values from all sources in arrival order */
for (i=0;i<7;i++){
    MPI_RECV(&temp, 1, MPI_INT, *, *, world, &status )
    if (status.MPI_SOURCE==source) a_element = temp;
}
```

Figure 4: Data Parallel Program fragment with Potentially Non-deterministic communication

one of two ways. First, the LP need not initiate EIT computations to determine that a matching message is safe. Instead it may simply wait until all messages in the expected-message-set have been received, at which point it sets its EIT to the maximum of the timestamps of all messages in this set. A second alternative is to have the LP initiate the standard EIT computation phase by sending EOT requests to the LPs in its source set and identifying safe messages. At the same time it also monitors its expected-message-set and as soon as it is complete, advances its EIT as explained above. For data parallel programs in particular, the compiler may compute the expected message set for those occasions when it cannot generate deterministic object code. Note that the expected message set can always be defined for every LP: in the default case, it consists of a null message from every LP in the source set. Thus incorporating this optimization in the protocol need not introduce additional overhead.

5 Results

A 24-node IBM-SP2 at UCLA was selected as both the target and host architecture for the experiments reported in this paper. Each node of the IBM-SP2 is a POWER2 node with 128Kb of cache and 256Mb of main memory. Nodes are connected using a high performance switch which offers a point-to-point bandwidth of 40Mb/s, and has a hardware latency of 500ns. This section describes the benchmarks and presents the validation and performance results. The performance of the parallel simulators is presented using the following metrics: *slowdown*, *speedup*, and *synchronization overhead*.

Assuming that the host architecture has n processors, speedup refers to the ratio of the time taken by the simulator when executing on one processor to the time taken by the simulation using n processors. Note that the 1 processor implementation uses a sequential simulation algorithm to synchronize the multiple LPs. Slowdown refers to the ratio of the total elapsed time taken to execute the simulator on the host architecture to the elapsed time to execute the target program on the target architecture. Slowdown is a measure of the overhead of the simulator. Assuming the application being simulated has sufficient parallelism, speedup measures the ability of the simulator to exploit this parallelism. The synchronization overhead is typically represented by the number of synchronization messages required in a given execution of the simulator.

The execution time for each experiment reported in this paper was taken by executing the program in exclusive or standalone mode and taking the average of at least three different runs.

5.1 Benchmarks

5.1.1 MPI Programs

PROSIM was validated for the NAS (Numerical Aerodynamic Simulation) Parallel Benchmarks (NPB 2)[BHS+95]; to the best of our knowledge this is the only public-domain benchmark suite for MPI. The NPB 2 benchmarks are a set of programs designed at the NASA NAS program to evaluate supercomputers. These benchmarks are written in Fortran 77 and embedded with MPI calls for communication. Since MPI-LITE currently supports privatization for only C programs, it was necessary to convert the benchmarks to C. We were able to convert four out of the five benchmarks using f2c[SGMN90], a Fortran-to-C converter, which were then used for our experimental study. The specific configurations of the benchmarks that were used in the performance study were constrained primarily by their memory and cpu requirements.

Table 1 summarizes the relevant configuration information for the benchmarks. Each benchmark was executed for 3 target machine configurations. For example, LU was executed on 4, 8 and 16 processors. The column labeled "Priv." is the degree of privatization of the simulator, i.e. the maximum number of threads that could be mapped to each host process of the simulator (this was primarily constrained by the available memory on each host processor and the memory requirements of each target process). For each target machine configuration, the columns labeled "Host" lists the size of the executable code for the target program and the simulator. It also lists the number of processors of the host machine that were used to execute the simulator.

5.1.2 Data Parallel Programs

A set of three data parallel programs were selected: Gauss-Jordan Elimination, matrix multiplication, and FFT computation. As we did not have access to a public domain HPF compiler, the programs were written in a C-based data parallel language called UC[BKM] developed at UCLA. The data distribution, reduction operations, and parallel assignments used in the UC programs reported in this paper are similar to those supported by HPF. UC has been implemented on the IBM SP2 at UCLA. As we had ready access to the compiler it was possible to directly insert the simulator hooks into the object code generated by the compiler.

5.2 Synchronization Algorithms

Four different synchronization algorithms were used to synchronize the parallel program simulator: in the first three modes, termed the NMP, CEP, and ANP modes, the simulator respectively uses the three asynchronous simulation algorithms described in a previous section to advance the EIT of each LP. In the fourth version, called the Deterministic Synchronization mode (or DSP), the simulator tries to switch to the deterministic mode when feasible, based on an analysis of the most recent receive statement executed by the LP. If the receive statement is deterministic, the LP does not initiate any synchronization messages (but will continue to respond to EOT requests from other LPs in the model). If the receive statement is not deterministic, the LP uses the ANP protocol to advance its EIT as described earlier. The above simulator modes allow us to determine the contribution of each protocol and each optimization to the performance of the simulator.

5.3 Verification & Validation

5.3.1 MPI Programs

The target programs and the simulators were executed for all processor configurations listed in Table 1. For each target and host processor configuration, the simulator was executed in each of the four modes described in the previous section. The NPB 2 benchmarks are self-verifying, meaning that each benchmark after completion compares the computed results against precomputed results to ensure that it executed correctly. All target programs and simulators were found to verify correctly.

Figure 5 plots the target program execution time (solid line) and the execution time as predicted by the simulator (dashed lines) as a function of various target machine configurations; note that the simulator predicted times are plotted for each host configuration listed in Table 1. The plots were for the various simulator modes were very similar, and consequently the figure only displays the predicted time in the DSP mode. For each benchmark, the execution time predicted by the simulator for the corresponding target application were plotted for sequential execution of the simulator and parallel implementations using 4 and 16 processors. The measurements for other configurations were similar and have been omitted. In the best case the predicted and measured times differed by less than 5% and in the worst by 20% lending reasonable credibility to the simulations. We also found that the predicted times matched the measured time very closely for long running configurations of the simulator, lending additional credibility to the accuracy of the simulator.

5.3.2 Data Parallel Programs

The simulator for data parallel programs was validated for a number of small programs by measuring the actual execution time for the program, in standalone mode, with the predictions made by the sequential and parallel simulators. Figure 6 shows the accuracy of the prediction for a program that implements Gauss-Jordan Elimination. The curve labeled "Real Execution" shows the execution time of the target program executing on a target architecture with N processors; $N = 4, 8, 16$. The curves labeled K Processor Simulation are the execution times predicted by the simulator for the preceding target programs using K processors of the host architecture. As seen from the figure, the predicted times are in close agreement with the measured times, with the predicted times lying within 5% of the actual execution time.

5.4 Sources of Error

As discussed in section 2, the accuracy of a simulator depends on the degree to which it satisfies the fidelity assumption, namely that local code execution times and message latencies are predicted accurately. We briefly discuss the primary sources that contribute to modeling and measurement inaccuracies.

- The runtime measurements to predict the execution time of local code blocks may be inaccurate due to a number of factors: first, the simulator causes additional code to be inserted in the target program that is simulated. Even though the time to execute this additional code can easily be excluded form the measurements, the inclusion of this code can have indirect effects whose impact on the measurements may be hard to isolate. For instance, insertion of the simuator will typically affect the cache behavior and register allocation. The only way to exclude these indirect effects is to use detailed simulation model for local code blocks as implemented in simulators like SimOS.

- We use a simple contention free model of the communication protocol. The model assumes a fixed transmission delay based on the message size, for each message, and hence does not account for the software and hardware queueing delays.

5.5 Performance

5.5.1 Protocol Performance for MPI Programs

We compared the performance of the asynchronous protocols with the quantum protocol for the MPI benchmarks. The performance of the simulation protocol in each simulator mode is gauged by the number of rounds of protocol messages, R, sent for each processor. The performance of the quantum protocol is gauged by the number of global synchronizations it would have taken to simulate the same target program. A round of protocol messages is similar to a global synchronization, although it is frequently less expensive, since in many cases, a processor does not need to wait to receive protocol messages from all other processors in order to identify some safe message form its input queue.

Given a target processor configuration, we found that R decreases only modestly on increasing the number of host processors used to simulate the configuration. Figures 7, 8, 9 and 10 show

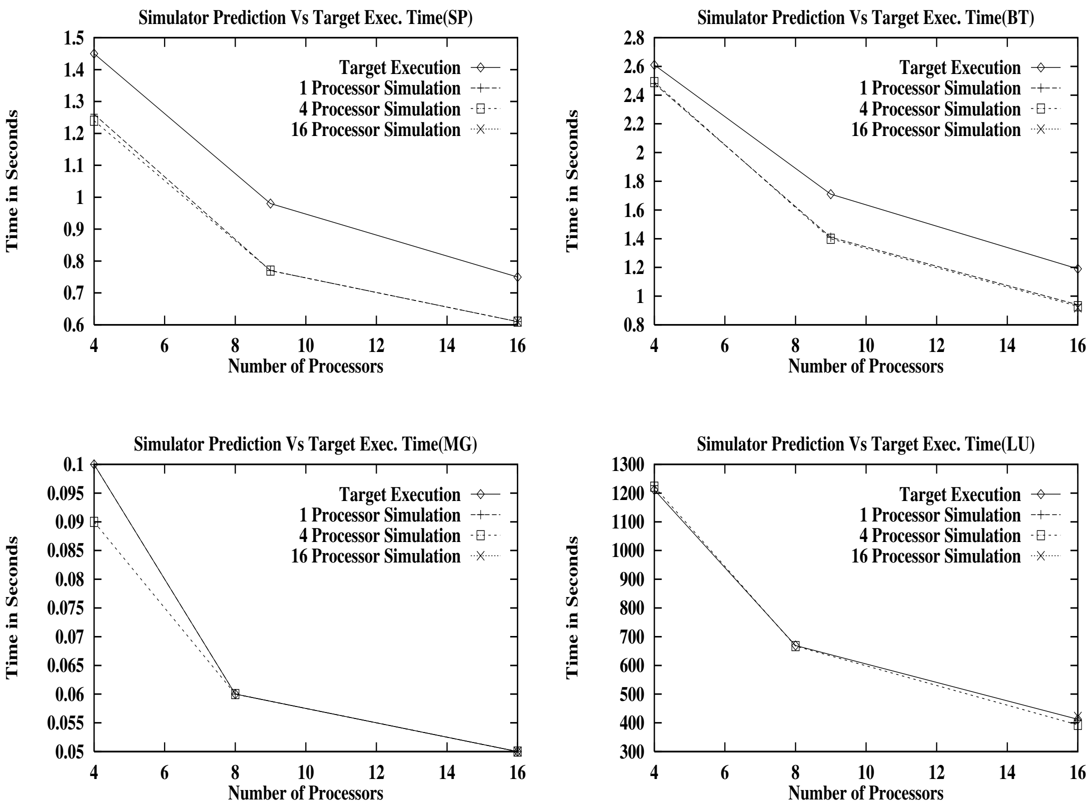

Figure 5: Target Execution Time Vs. Simulator Predictions for NAS benchmarks

Name	Lines	Class	Target Procs.	Priv.	Targ. 1	Targ. 2	Targ. 3
					Target Program Size/Simulator Size (Host Procs. for Simulator)		
LU	4623	A	4,8,16	4	14M/57M (1,2,4)	8M/32M (2,4,8)	5M/18M (4,8,16)
MG	2712	S	4,8,16	16	600K/8M (1,2,4)	400K/5M (1,2,4,8)	300K/3M (1,2,4,8,16)
BT	6290	S	4,9,16	16	2M/24M (1,2,4)	1M/15M (1,2,4,9)	1M/12M (1,2,4,9,16)
SP	5555	S	4,9,16	16	700K/7M (1,2,4)	500K/6M (1,2,4,9)	500K/5M (1,2,4,9,16)

Table 1: NAS Benchmarks

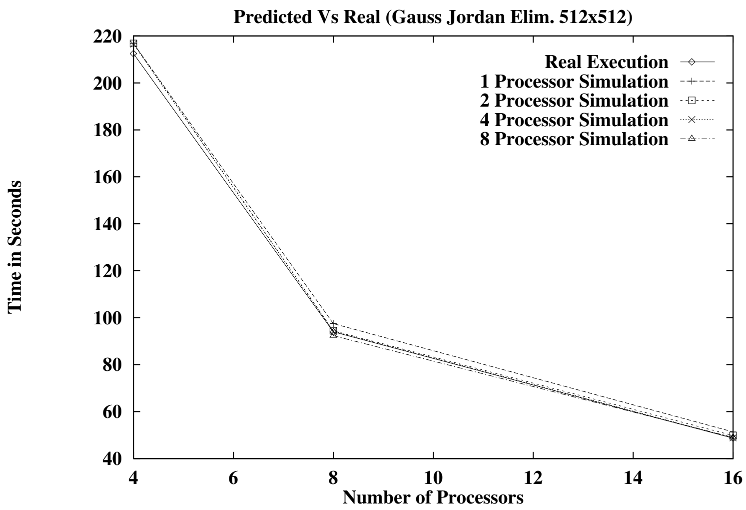

Figure 6: Simulator accuracy: Data Parallel Program for Gauss Jordan Elimination

the variation of R with the simulator modes for two representative target and host processor configurations of each benchmark. In each graph, the number of rounds of protocol messages is normalized against the number of global synchronizations of the quantum protocol.

Consider only the CEP mode: the amount of improvement over the quantum protocol is strongly dependent on the average duration for which an LP (i.e. thread) executes before getting blocked. Table 2 shows this average duration for each benchmark and each target program configuration, in terms of L, the minimum message latency of the target machine. The 9 processor BT benchmark has the largest average uninterrupted execution time per thread, and in the simulation, the ANP mode is able to eliminate more than 80% of the global synchronizations of the quantum protocol. The 16 processor MG benchmark has the smallest average uninterrupted execution time per thread, and the ANP mode is unable to significantly reduce the number of global synchronizations of the quantum protocol.

The performance of the CEP mode is significantly better than the NMP mode only for the 9 processor BT benchmark. The NMP mode eliminates 40% of the global synchronizations in the quantum protocol, and the CEP mode eliminates 80%. This is because the CEP significantly improves over the NMP only when some LPs are far ahead of the others in simulation time, requiring the other LPs to exchange many rounds of null messages to update their simulation times. This situation is more likely to occur when the average duration of uninterrupted execution is long, as in the 9 processor BT benchmark.

The NMP mode almost never performs better than the CEP mode, and the ANP mode is not significantly better than simply the CEP mode. This is because all the benchmarks predominantly use one communicator, and consequently, the null message protocol is unable to extract and use information on the communication topology.

Using the DSP mode,, we note that it is possible to eliminate *all global synchronizations* in the BT and SP benchmarks. However, the optimizations were not effective in significantly reducing the synchronizations from the MG and LU benchmarks as discussed in the next section.

5.5.2 Parallel Simulator Performance: MPI Programs

A number of techniques may be used to enforce deterministic communications in a parallel program. In a MPI program, two commonly used techniques to define determinstic receives are either by having a receive specify the source explicitly or if it specifies an explicit tag and each source uses unique tags. Although the first type of determinism can be detected automatically by the current simulator, we have not yet implemented the second mode. Out of the four NPB benchmarks considered in this study, SP and BT use determinism of the first type, and MG and LU benchmarks specify determinism of the second kind. As the simulator does not automatically detect the second type of determinism, we manually inserted the optimizations to evaluate the potential benefit that can be derived from exploiting this form of non-determinism. The final speedups obtained from the execution of all the benchmarks are presented in Figure 11. The speedup for the LU benchmarks are relative to the smallest host processor configuration that could be used to run the simulator. For example, the 8 target processor simulator could be executed on 2, 4 or 8 host processors. Hence, the reference execution time is of the 2 processor simulation. This *understates* the expected performance improvement for this application. Extrapolating from the speedup obtained by the 4 processor target program, we expect this application to also yield the excellent speedup that were obtained for the SP benchmark.

The performance of the NMP, CEP, and ANP protocols for the four benchmarks is presented in Figure 12. Each graph describes the performance of the three protocols for the target machine configuration with 16 processors. In general, the parallel performance of these protocols is poor primarily because of the relatively low computation granularity for the applications. An analysis of the parallel execution showed clearly that a very large number of null messages were used even with the ANP protocol, which effectively cancelled any benefits that accrued from using more processors.

5.5.3 Data Parallel Programs

The performance of the simulator for data parallel program is presented using both the *slowdown* and *speedup* metrics.

Figure 13, 14, and 15 respectively present the simulator slowdowns and speedups for the Gauss-Jordan Elimination, FFT and Matrix Multiplication programs. Clearly, the simulator speedups and slowdowns are very application dependent. For example, in Gauss-Jordan elimination, when we increase the number of processors in the simulation from 1 to 2, the simulation time goes down for all problem sizes. There are two conflicting factors here: The decrease in computation time due to the fact that the number of processors increase, and the increase in communication time for communications between threads that now lie on different processors. In Gauss-Jordan the benefit due to the first factor outweighs the loss due to the second factor. In both matrix multiplication and FFT, this is not the case for the finest grained problem executions (64 processors in FFT and 32 processors in matrix multiplication), where the second factor outweighs the first. So the slowdown increases from 100 to 160 in FFT, and from 80 to 110 in matrix multiplication, on increasing the processors from 1 to 2. Consider the comparative speedups obtained for target programs that use a varying number of processors(N) in the target architecture, for a constant data size and a constant number of processors (K) in the simulator: for Gauss-Jordan elimination, the speedup increases with N, but decreases for FFT and matrix multiplication. Closer analysis revealed that in the latter two applications, the communication in the target program increased substantially with N which also results in degrading the performance of the simulator.

6 Related Work

Most parallel program or architecture simulators use sequential or parallel implementations of the quantum protocol. Among these are Proteus[BDCW91], a parallel architecture simulation engine, Tango[DGH91], a shared memory architecture simulation engine, Wisconsin Wind Tunnel[RHL+93], a shared memory architecture simulation engine and SimOS[RBDH97], a complete system simulator (multiple programs plus operating system). Two simulation engines which use approaches similar to ours are Parallel Proteus[LW96] and LAPSE[DHN94].

In Proteus, the application to be simulated is written in a superset of C, and constructs are provided to control the placement of data. Library routines are provided for message passing, thread management, memory management and data collection. The target architecture is specified in terms of interconnection medium (bus, direct or indirect network), sizes of shared and private memory at each node, and other features e.g. interprocessor interrupts and handlers. An application- and architecture-specific simulator is created by the simulation engine, which upon execution produces a trace file that can be interpreted by Proteus tools. The simulator uses a custom lightweight threads package. It uses direct execution for most instructions of the application program, but must simulate message passing and shared memory access instructions. As these instructions are the costliest to simulate, the simulator provides low and high accuracy network and shared memory modules to allow the user to tradeoff speed and accuracy in model execution.

Tango has primarily been used to simulate the execution of programs written in typical shared-memory programming notations on shared memory computers. The target application is written in C or FORTRAN, using macros to emulate a variety of programming paradigms, such as locks, barriers, distributed loops and messages. These macros eventually expand into routines which implement equivalent primitives of the target architecture. The code is augmented for direct execution. The simulator executes as set of Unix processes. Target architecture primitives are implemented by interactions between the processes (using shared memory and semaphores), and interactions between the processes and the memory simulator, which can have varying degrees of accuracy.

As discussed in section 2, in general, simulators that use the quantum protocol must trade-off simulation accuracy with speed, with the exception of WWT which uses hardware implementations of barriers to improve efficiency. WWT is a simulator of cache-coherent, shared memory computers that runs on the Thinking Machines CM-5. It relies on direct execution of local code; shared memory of the target architecture is simulated by trapping on accesses to invalid blocks, where a block is marked invalid by manipulating the ECC bits in the CM-5 memory. This automatically causes a trap upon access, and a software protocol is used to obtain the required block, and charging the simulation only with the time the same operation would have taken on the target machine. A conservative distributed simulation protocol is executed to synchronize processors. The processors execute a (hardware) barrier after every Q simulation cycles, in order to ensure that all messages sent in the current quantum are received before the next quantum starts. $Q < T$ where T is the minimum message latency of the target machine.

Both LAPSE and Parallel Proteus use some form of program analysis to increase the simulation window. LAPSE is a parallel simulation engine for programs that use the message passing library of the Intel Paragon. It uses a quantum protocol called WHOA (Window-based Halting On Appointments) and uses runtime analysis to determine the size of the simulation quantum. Parallel Proteus is the parallelization of the Proteus simulation engine[BDCW91], which uses the quantum protocol. The synchronization overhead caused by frequent barriers is reduced using *predictive barriers* and *local barriers*. Predictive barriers is a method for safely increasing the simulation quantum beyond L, the minimum communication latency of the target machine. This method uses runtime and compile time analysis to determine, at the beginning of a simulation quantum, the earliest simulation time at which any LP will send a message to any other LP. Consequently, the simulation quantum can be extended until that time. Runtime analysis involves simply running an LP until it communicates. If it stops at the equivalent of a receive statement, analysis performed at compile time is used to predict when it would have sent a message if it were instantly resumed. The method of local barriers uses statically available communication topology information (i.e. groups of LPs that communicate only within the groups they belong to) to reduce the global synchronization at the end of a simulation quantum to local synchronizations between groups of LPs. In comparison, we use the equivalent of runtime analysis since we execute an LP until it reaches a receive statement. The benefits of compile time analysis are achieved using the conditional event protocol, which is portable and does not need target instruction set analysis. In addition, our implementation of the null message protocol adapts automatically to the dynamically changing communication topology specified by the target program. Perhaps most importantly, it automatically recognizes deterministic code fragments and switches off all synchronization while simulating the corresponding fragment. As seen in Section 5 this optimization helps us eliminate almost all synchronization overhead in simulating many real applications.

7 Conclusion

Parallel simulation appears to offer significant potential in reducing the execution time of simulation models of parallel programs and architectures. We have developed a parallel simulator that uses asynchronous conservative synchronization protocols together with optimizations that exploit the communication characteristics of the program being simulated. The simulator has been used to simulate a variety of task and data parallel programs including four of the five benchmarks that are defined in the NAS Parallel Benchmarks Suite (NPB 2). The results show that the optimizations suggested in this paper can significantly reduce the synchronization overheads for the simulator.

We are currently extending the simulator to incorporate detailed models of input output systems, parallel file systems, and IO data caching and placement algorithms.

8 Acknowledgements

All the data presented in this paper was collected on the IBM-SP2 at UCLA's Office of Academic Computing, granted to UCLA by IBM Corporation under their Shared University Research Program. We are thankful to the members of the SESAME project at UCLA for their comments and contributions to this work. Special thanks to M. Dhagat for implementing the data parallel programs and to P. Bhargave, S. Docy, and A.Kahn for their role in implementing the simulator.

References

[BDCW91] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, Cambridge, MA 02139, 1991.

[BHS+95] D. Bailey, T. Harris, W. Saphir, R. v.d. Wijngaart, A. Woo, and M. Yarrow. The nas parallel benchmarks 2.0. Technical report nas-95-020, NASA Ames Research Center, Moffet Field, CA 94035-1000, December 1995.

[BKM] R. Bagrodia, K.M.Chandy, and M.Dhagat. UC: a set-based language for data parallel programming. To appear.

[BtL94] R. Bagrodia and Wen toh Liao. Maisie: A language for design of efficient discrete-event simulations. *IEEE Transactions on Software Engineering*, April 1994.

[CDJ+91] R.G. Covington, S. Dwarkadas, J.R. Jump, J.B. Sinclair, and S. Madala. The efficient simulation of parallel computer systems. *IJCS*, 1:31–58, 1991.

[CH96] Sashikanth Chandrasekaran and Mark D. Hill. Optimistic simulation of parallel architectures using program executables. In *10th Workshop on Parallel and Distributed Simulation (PADS96)*, pages 143–150, May 1996.

[CM81] K.M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11):198–206, August 1981.

[CS89] K.M. Chandy and R. Sherman. The conditional event approach to distributed simulation. In *Distributed Simulation Conference*, Miami, 1989.

[DGH91] H. Davis, S. R. Goldschmidt, and Hennessey. Multiprocessor simulation and tracing using Tango. In *Proceedings of ICPP'91*, pages II99–II107, August 1991.

[DHN94] P. Dickens, P. Heidelberger, and D. Nicol. A distributed memory lapse: Parallel simulation of message-passing programs. In *Workshop on Parallel and Distributed Simulation*, pages 32–38, July 1994.

Benchmark	16 Targ. Proc.	8 or 9 Targ. Proc.
LU	8.74L	11.77L
MG	2.79L	4.03L
BT	12.33L	24.81L
SP	4.61L	9.29L

Table 2: Average Uninterrupted Execution Time

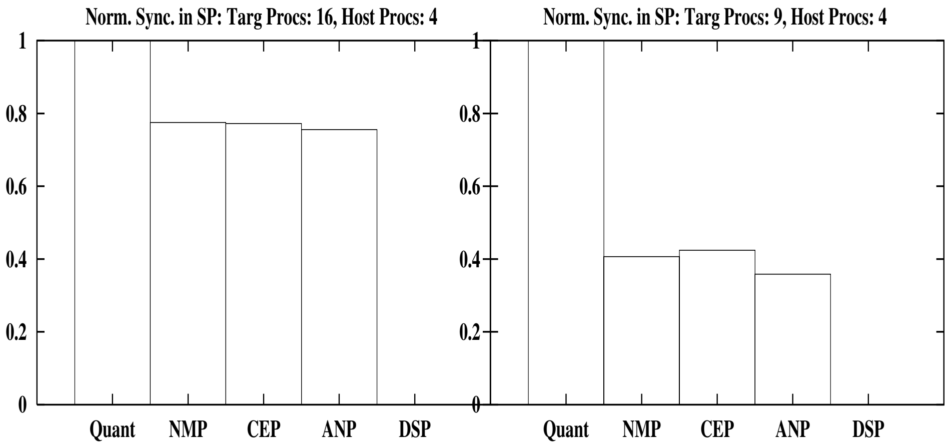

Figure 7: Performance of Simulators for SP

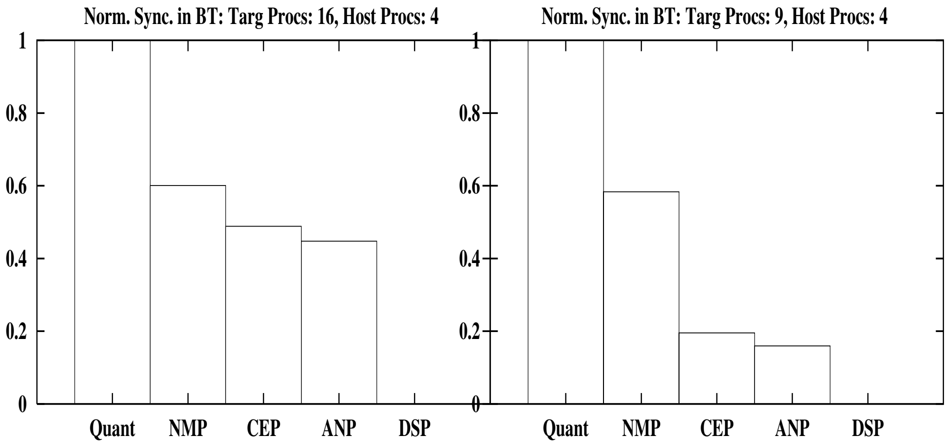

Figure 8: Performance of Simulators for BT

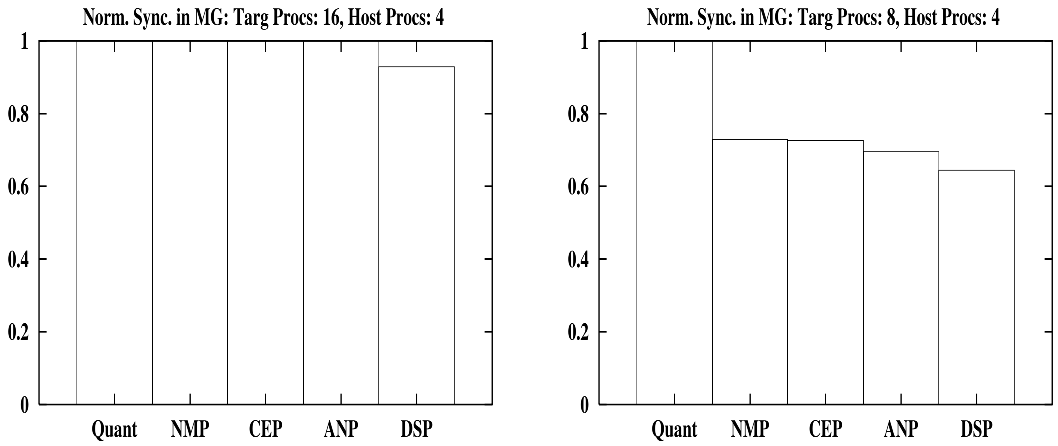

Figure 9: Performance of Simulators for MG

[For93] MPI Forum. Mpi: A message passing interface. In *Proceedings of 1993 Supercomputing Conference*, Portland, Washington, November 1993.

[Fuj88] R. Fujimoto. Lookahead in parallel discrete event simulation. In *International Conference on Parallel Processing*, August 1988.

[Hig93] High Performance Fortran Forum. High Performance Fortran Language Specification. Available by anonymous ftp from titan.cs.rice.edu, May 1993.

[JB93] Vikas Jha and Rajive Bagrodia. Transparent implementation of conservative algorithms in parallel simulation languages. In *Winter Simulation Conference*, December 1993.

[LW96] U. Legedza and W. E. Weihl. Reducing synchronization overhead in parallel simulation. In *10th Workshop on Parallel and Distributed Simulation (PADS96)*, May 1996.

[Mis86] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[PB95a] S. Prakash and R. Bagrodia. An adaptive synchronization method for unpredictable communication patterns in dataparallel programs. In *Proceedings of the International Parallel Processing Symposium*, 1995.

[PB95b] Sundeep Prakash and Rajive Bagrodia. Parallel Simulation of Data Parallel Programs. In *8th Workshop on Languages and Compilers for Parallel Computing*, August 1995.

[RBDH97] Mendel Rosenblum, Edouard Begnion, Scott Devine, and Stephen A. Herrod. Using the simos machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1), January 1997.

[RHL+93] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference*, May 1993.

[SGMN90] S.I.Feldman, D. M. Gay, Mark W. Maimone, and N.L.Schryer. A fortran-to-c converter. Technical report no. 149, AT&T Bell Laboratories, Murray Hill, NJ, May 1990.

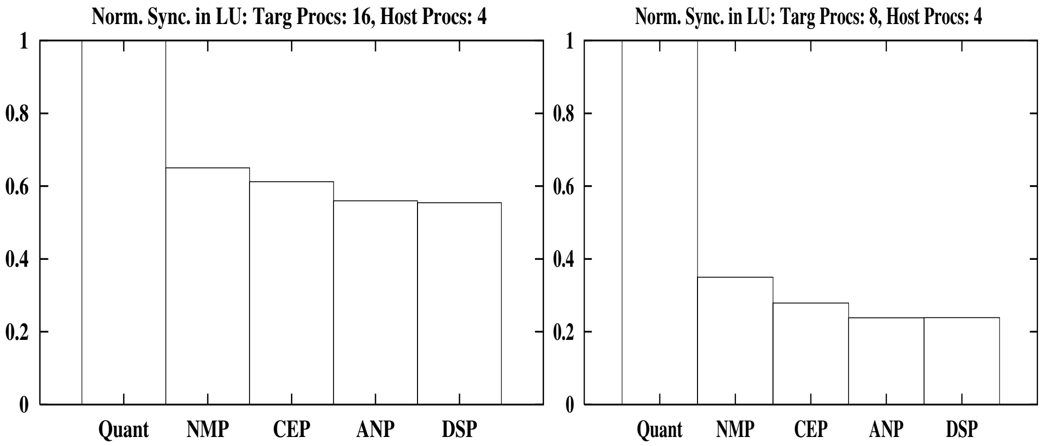

Figure 10: Performance of Simulators for LU

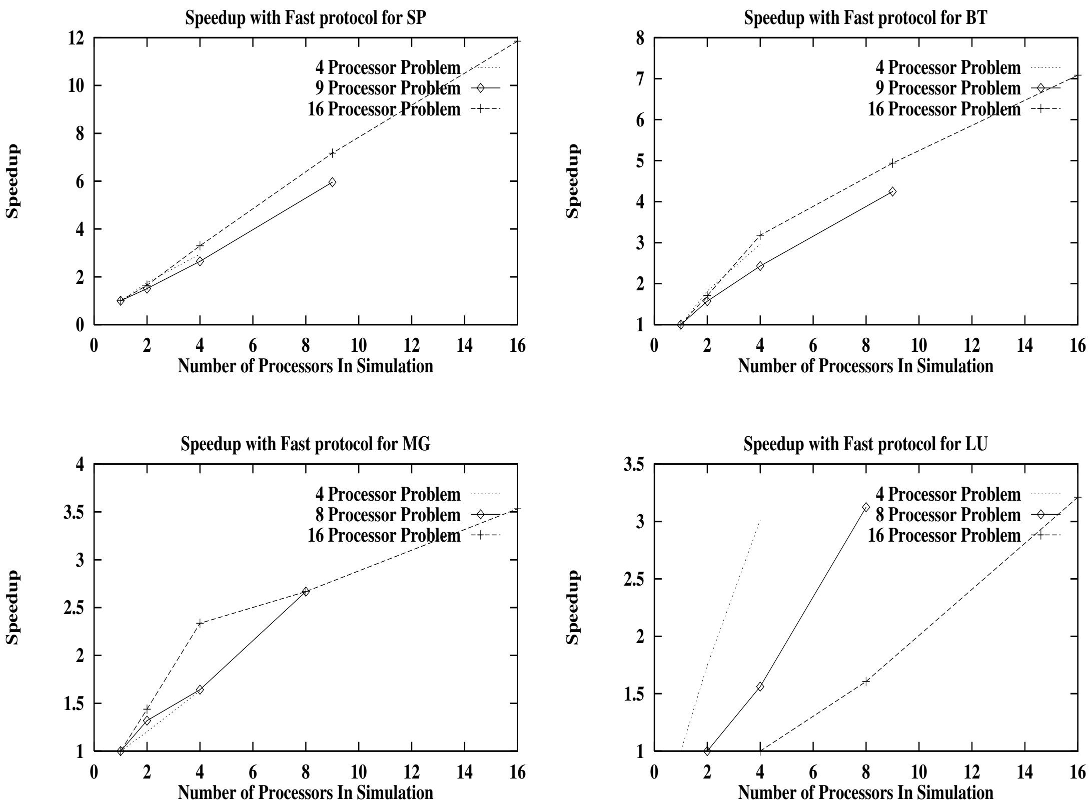

Figure 11: Deterministic Mode Simulator Speedups

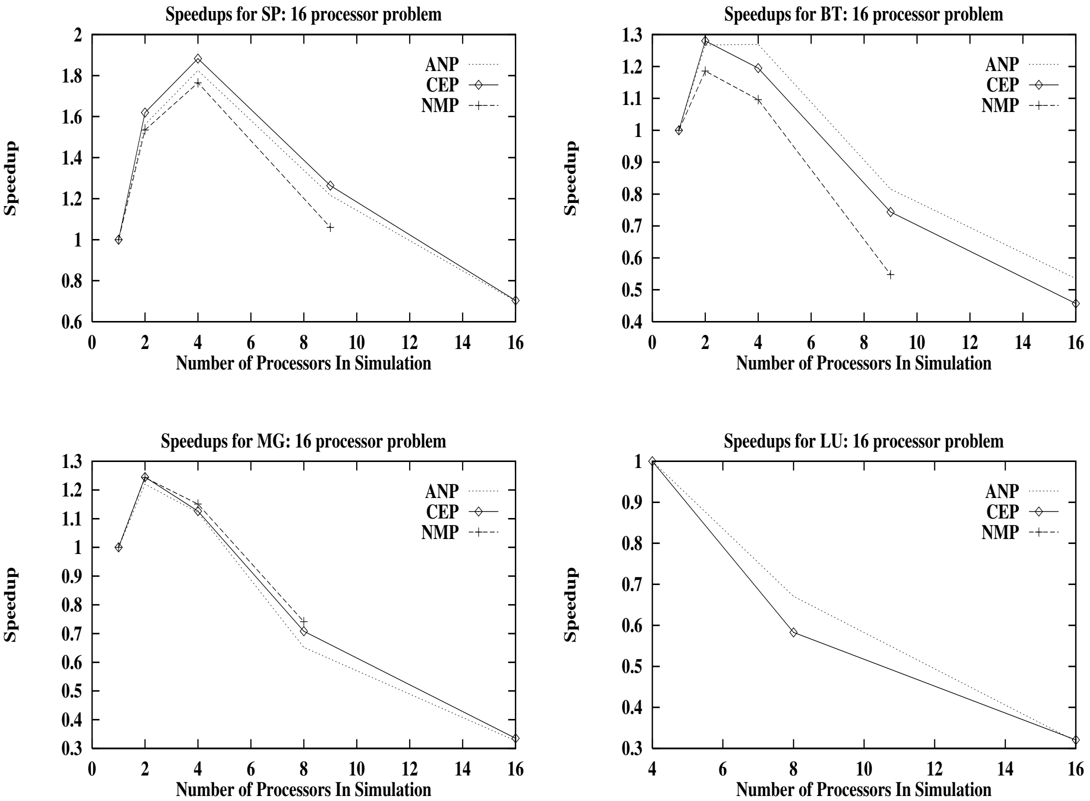

Figure 12: Simulator Speedups without Deterministic Mode Detection Optimizations

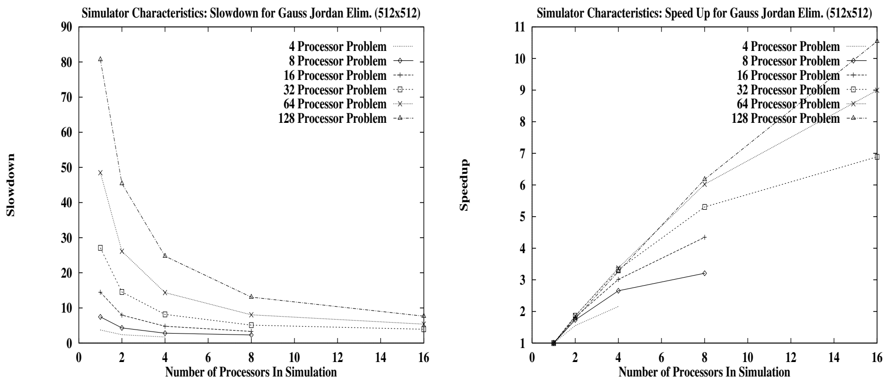

Figure 13: Simulator Characteristics for Gauss Jordan Elimination

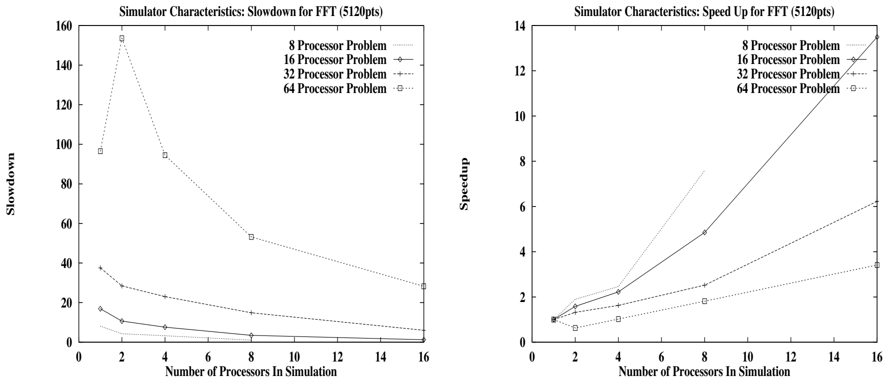

Figure 14: Simulator Characteristics for FFT

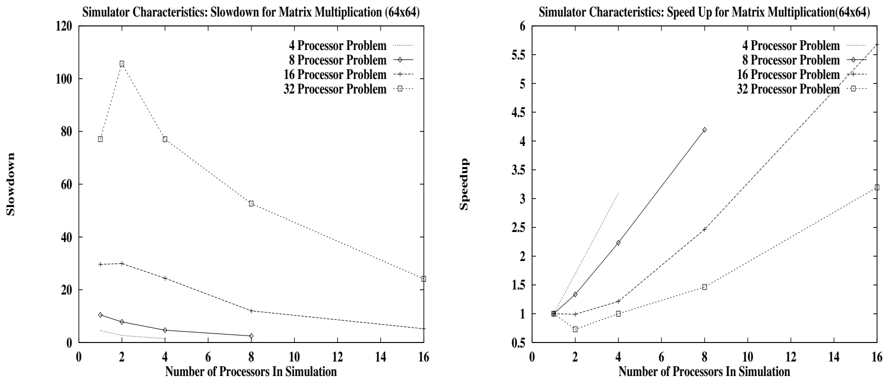

Figure 15: Simulator Characteristics for Matrix Multiply