# SIMULATING MESSAGE-DRIVEN PROGRAMS *

Attila Gürsoy
Department of Computer Science
University of Illinois, Urbana IL
email: gursoy@cs.uiuc.edu

Laxmikant V. Kalé
Department of Computer Science
University of Illinois, Urbana IL 61801
email: kale@cs.uiuc.edu

*Abstract – Simulation studies are quite useful for performance prediction on new architectures and for systematic analysis of performance perturbations caused by variations in the machine parameters, such as communication latencies. Trace-driven simulation is necessary to avoid large computational costs over multiple simulation runs. However, trace-driven simulation of nondeterministic programs has turned out to be almost impossible. Simulation of message-driven programs is particularly challenging in this context because they are inherently nondeterministic. Yet message-driven execution is a very effective technique for enhancing performance, particularly in the presence of large or unpredictable communication latencies. We present a methodology for simulating message-driven programs. The information that is necessary to carry out such simulations is identified, and a method for extracting such information from program executions is described.*

## 1 INTRODUCTION

An accurate performance prediction of parallel computations plays an important role in designing parallel algorithms and evaluating machines. Many computational complexity models have been derived for parallel algorithms. Although these analytical approaches are very useful to determine the fundamental performance limits of parallel algorithms, they are often inadequate to analyze the parallel computations and the interactions between computations and parallel architectures. Particularly, the load imbalance, scheduling, synchronization, and the dynamic properties of the parallel computations make the analytical approaches difficult, if not impossible, for this purpose.

Simulation techniques offer a more realistic analysis in this regard, and they have been used extensively to predict the performance of parallel programs on specific computers. There are various simulation techniques that are suitable for different purposes depending on the accuracy of the prediction desired and the complexity of the simulator itself. For example, one can emulate the user code instruction by instruction in the simulated environment. Although this method provides very accurate results, the simulator itself is very costly in terms of both the development and the computation (for instance, emulating every instruction of a parallel code on a 1000 processors would be very time-consuming). Another approach which makes simulations more affordable is to use an abstract model of the computation instead of emulating a specific computation. The model may contain some statistical properties such as average number of messages sent, average computation size etc. As no user computation is executed during the simulation, the simulation time (and the computation power required) is small. However, this approach too may not capture adequate details of the computation and is not useful to predict the performance of a specific program.

Trace-driven simulation is another approach which combines the advantages of both. Its complexity is in between the above two, and yet it is powerful enough to capture the details of the computation. A trace is a time ordered sequence of significant events that happened during the execution of a program. The traces are collected from an actual execution of the program, and it is fed to the simulator. The simulator, then, executes these traces on a model of the new system. Trace-driven simulation has been successfully used in studies of uniprocessor systems such as memory designs, cache performance etc [11]. In such studies, the order of events in the system to be simulated (such as the sequence of addresses accessed) was assumed to be deterministic. Therefore, the changes in the simulated environment affected only the length of the time interval between the events, not the actual sequence of events.

The trace-driven simulation has been extended to

*This research was supported in part by the National Science Foundation grants CCR-91-06608 and ASC-93-18159.

study parallel computations also [4, 9, 5]. However, application of the trace-driven simulation to parallel systems poses a problem. The behavior of a parallel computation may change under a new environment which invalidates the traces. Some of the trace-driven simulations of parallel computations were limited to programs written in traditional message-passing style (i.e., single process per processor and *blocking* message receives that ask a specific message with a given tag and source processor). In these cases, the behavior of the programs remains the same despite the changes in the environment [9]. In general, message-passing programs may contain nonblocking message passing primitives which introduce nondeterminism. With nonblocking receives, the behavior of the program may depend on the arrival order of messages. For example, the program may check for a particular message, if the message is not there, the program may choose a different action. In a simpler context, the problem posed by this sort of non-determinism was addressed in some trace-driven simulation studies [8]. These studies used a hybrid method combining trace-driven simulations and real execution of user code to study the performance of shared memory systems. This approach is difficult and expensive since it requires execution of a part of user code during simulation. The execution-driven simulation [3] or hybrid simulation of message passing programs whose behavior depends on message arrival order would be more difficult and impractical.

Message-driven execution, explained in Section 2, is based on the ability to run computations in different orders. Therefore, the simulation of message-driven computations inherently involves dealing with this difficulty. In a message-driven computation, the messages may arrive in a different order due to numerous reasons in the new simulated environment. The execution trace of the program becomes invalid at that point because the rest of the computation is different from the traces. In order to achieve accurate simulation, it is necessary to reconstruct remaining sequence of computation steps. This is impossible, in general, without rerunning or interpreting the program instruction by instruction. In some special cases, using the knowledge about the algorithm/computation model, the sequence of computations can be reconstructed.

In this paper, we will describe a method to simulate message-driven programs written in the Dagger language [7]. The Dagger language, in addition to helping the expression of message-driven programs, also turns out to expose sufficient parts of dependence structure of each parallel object so as to render accurate trace-driven simulations feasible. Our approach depends on extracting some semantic information from the Dagger programs satisfying certain conditions (which will be described later), and using this information together with the execution traces to conduct the simulation. The technique is not limited to the Dagger language. It can be extended to other parallel programming languages such as CC++ [1], OCCAM with some modification.

The rest of the paper is organized as follows: Section 2 describes the message-driven execution and the Dagger coordination language. Section 3 discusses the simulation of Dagger programs. The abstract machine model that is used by the simulator is explained in Section 4. The design of the trace-driven simulator is discussed in Section 5. Some examples of illustrating the usage of the simulator are presented in Section 6, and the conclusion in Section 7.

## 2 MESSAGE-DRIVEN EXECUTION AND DAGGER

The traditional method of programming distributed memory parallel computers involves a traditional message-passing style of programming. This style involves one process per processor. The processes may send messages to each other and issue blocking system calls to receive a specific message. They may also invoke global operations such as reductions and scans which act as barriers, i.e. all processors must invoke these operations in identical sequence and each processor must wait until all processors have arrived at each barrier. Despite its simplicity, this style of parallel programming often leads to severe performance impediments, because it requires the programmers to commit to a particular sequence in which the messages must be processed. Although, one may use nonblocking communication primitives to overlap communication and computation in traditional message-passing style, its usage is limited to a single module: it is difficult to overlap communication latencies across multiple modules [6] without loosing modularity.

In message-driven execution, there are typically many processes per processor. A process does not block the processor it is running on while trying to receive a message. Instead, processes are scheduled for execution depending on the availability of the messages for them. Processes typically provide code in the form of entry functions or continuations and a way of associating them with specific incoming messages. With this information, the runtime system can invoke the appropriate code in the appropriate process to handle a particular incoming message. There-

fore, Message-driven execution provides the ability to overlap computation and communication, and tolerates communication latencies. It helps latency tolerance in two ways: First, when one process is waiting for data from a remote process, another ready process may be scheduled for execution. Secondly, even a single process may wait for multiple data items simultaneously, and continue execution whenever any of the expected items arrive. Message-driven style also supports the use of parallel libraries without loss of efficiency. In traditional message-passing programs, the program has to yield control completely to the parallel library. Thus, the idle times in the library computation cannot be utilized effectively. Message-driven execution on the other hand, allows the control to switch between multiple concurrent library computations. Details of performance benefits of message-driven execution can be found in [6].

Charm [10] is one of the first systems to embody message-driven execution in a portable parallel programming system running on stock multicomputers. A Charm program/computation consists of potentially small-grained processes or objects, called chares. A chare consists of local data, entry-point functions, and private and public functions. Public functions can be called by any object on the same processor. Entry functions are invoked asynchronously by an object on any processor. Invoking an entry function in a remote object can also be thought of as sending a message to it.

Despite its performance benefits, the expression of programs in a pure message-driven language, such as Charm, is difficult due to the split-phase style that it requires and the nondeterministic arrival of messages. Consider a concurrent object (listed in Figure 1) which performs the following calculations. The object executes `C0` when it is created (assume entry `e0` is invoked by the creation message). Then, it can perform either `C1` or `C2` in any order whenever their corresponding entry is invoked due to message arrival. After both `C1` and `C2` have been completed, then it performs `C3`. Since `e1` or `e2` can be invoked in any order, each entry must keep track of whether the other one is already done, so that the later one invokes `C3`. To achieve this, a counter is used as shown in the code. The counter is set to 2 at the beginning (number of subcomputations before `C3`). Whenever the counter reaches zero, then `C3` is called. In more complex computations, the expression of such cases becomes quite difficult (in addition to being difficult to simulate as we will show later). In order to simplify the expression of message-driven programs, a new notation, Dagger [7], was developed to express dependences between subcomputations and messages within a single object on top of Charm language.

```
chare G {
  int count;

  entry e0 : {   C0(); count=2;}
  entry e1 : {   C1();
                 if(--count==0) C3()};
  entry e2 : {   C2();
                 if(--count==0) C3();}
}
```

Figure 1: The Charm code for `chare G`

A Dagger program includes *dag-chares* as a special form of concurrent objects in addition to regular chares. The dag-chare for the previous Charm code is listed in Figure 2. The message receiving points are specified by entry declarations. The subcomputations within a dag-chare are called *when-blocks*. The when-block `when e0` is executed when a message has been received at the entry `e0`. The execution of a when-block is completed without interruption. The when-block code may contain some sequential computation as well as some specific Dagger statements for synchronization such as `Expect`, and `Ready`. The instruction `Expect(`$e_i$`)` tells the Dagger that the message for the entry $e_i$ can be made available to the dependent when-blocks. In other words, reception of message is not sufficient to trigger a subcomputation, it must be expected also. `Ready` is equivalent to sending a message to an entry-point within the same dag-chare and issuing an `Expect` for this message. Since the message is local to the dag-chare, it can be implemented more efficiently than actually sending the message. The efficient implementation is achieved by conditional variables — a special synchronization variable that is local to the dag-chare. In addition to simplifying the expression of message-driven programs, it turns out that Dagger also provides necessary information to simulate message-driven programs by allowing us to trace additional events as discussed in the next section.

```
dag chare G {

  local-variable-declarations

  entry e0 :  (message MSG *m0);
  entry e1 :  (message MSG *m1);
  entry e2 :  (message MSG *m2);

  CONDVAR c1;
  CONDVAR c2;

  when e0:{C0();expect(e1);expect(e2);}
  when e1:{C1();ready(c1);}
  when e2:{C2();ready(c2);}
  when c1,c2:  {C3();}
}
```

Figure 2: The Dagger code for dag-chare G

## 3 SIMULATION OF DAGGER PROGRAMS

An accurate trace-driven simulation of message-driven programs is not possible without a complex dependence analysis of various paths through the each entry-point. Traces from one instance of execution may not cover all the possible execution paths which may depend on message arrival order.

We will explain why simulation is not possible without the dependence information for the program given in Figure 1. Traces from an instance of execution of the program consist of the duration of execution of each entry-point (and relative timings of any message sent during it). Assume that an instance of the chare G is created when a message for the entry e0 is received. Then, G awaits two messages concurrently, one for e1 and one for e2. In a particular execution, assume that the message for e1 arrives first. This causes the execution of C1. Then, when the message for e2 arrives, C2 is executed followed by C3. However, if the messages arrived in the reverse order, the code at e2 would only execute C2 leaving the execution of C3 to the other entry-point. If traces are obtained with the former sequence with A time units for the execution of e1, and B time units for e2, and during simulation the machine conditions lead to the latter sequence, it is not possible to reconstruct the times of e2 and e1 from A and B.

If the individual times for computations C1, C2 and C3 were recorded, one would be able to reconstruct the timings in presence of the new sequence. It may seem simple then to record these times. However, note that C1, C2 and C3 need not be function calls as shown here. The if statements as well as the computation blocks might be deeply buried inside complex control structures. Therefore, in general, it is not easy to retrieve the timings of the individual blocks. Furthermore, the connection between the value of counter becoming zero and arrival of messages may not be easy for the compiler to deduce.

In order to accurately simulate a message-driven program via trace-driven simulation, the simulator needs to reconstruct the execution sequence under the new runtime conditions. Dagger facilitates this reconstruction by

- tracing the execution at the level of basic blocks rather than only messages, and
- providing information about dependences among messages and computations.

Dagger statically captures the dependences, and its runtime is able to trace the beginning and end of each individual when-block. The dependencies among the blocks and messages forms a partial ordering. The actual execution sequence in a given run will depend upon message arrival sequence, but must be consistent with the partial order. The simulator knows the message arrival order from its runtime environment and the dependence structure from the Dagger translator, and thus, can mimic the Dagger runtime to reconstruct the new sequence correctly without re-executing the user code.

For the example in Figure 2, let's assume that message arrival order in the real execution is (e0,e1,e2) and the order of subcomputations is (C0,C1,C2,C3). During simulation in a new environment, assume that messages arrive in the order (e2,e0,e1). When a message has been received at e2, the subcomputation C2 cannot be executed because C0 is not completed yet. Later, when the message for e0 arrives, we can execute C0. Now the subcomputation C2 is ready for execution. However, in the execution trace, the next one is the subcomputation C1. In the partial order, C1 and C2 are incomparable, that is, they can be executed in any order. So we can execute C2. When e1 receives a message, then we can execute C1 and then C3. So the simulator executes the blocks in the order (C0,C2,C1,C3) without violating the partial order.

For the above reconstruction to be valid, an additional condition, which is quite natural, must be satisfied. To see the need for this condition, consider the same example again. If the code inside the block `C1` contains sections whose execution time and data values depend on variables set in `C2`, the execution under the new arrival order will not match the traces (i.e., block `C1` may take more or less time or even may send out different number of messages). Such uncaptured dependences constitute a bad programming style and occur very rarely in parallel programs. The condition required for accurate simulations can be stated concisely as follows:

> A variable *used* in a when-block `W` must not be modified by any other when-block that is incomparable[a] to `W` in the partial order defined by the DAG (dependency graph).

The Dagger compiler extracts the dependency information from the user program to be used by the simulator. The compiler also inserts the necessary code to produce the traces during execution. The traces from an execution are gathered and combined with the static dependency information. Since we know the basic blocks and their dependencies, we do not need to trace every instruction that is executed during the run. Only a small number of events have to be traced, and only a small amount of data needs to be stored for each event. This reduces the computational cost of the simulations significantly. The events to be traced that are sufficient for the simulation are:

1. beginning of a when-block
2. send message
3. broadcast message
4. expect and ready,
5. initialization of condition variables, and
6. end of a when-block.

## 4 ABSTRACT PARALLEL MACHINE MODEL

In this section, we will define a model for the parallel machine to be simulated. Despite the large diversities among the parallel machines, they have a common property: to access remote data takes longer than to access local data. The machine model emphasizes this property. In this model, a parallel machine is a collection of processing elements (PE) interconnected by a communication network. A processing element consists of a processor, a local memory, and possibly a communication processor. The communication processor interfaces the processor to the network. It can access to the local memory and interact with the network without blocking the processor, therefore it releases the processor from most of the communication related tasks. The network provides communication among PEs. In reality, there exist various communication network structures with different topologies and communication protocols. From the point of view of our simulator, the network provides data transfer with a latency that may depend on the network load in an arbitrary fashion, and it has a finite capacity.

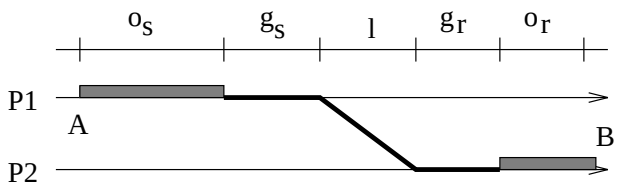


Figure 3: Sending A Message

Communication between two processors involves a number of steps. Each step requires a distinct time interval which must be charged to appropriate component of the system. We will explain these steps by an example which is depicted in Figure 3. Processor $P_1$ starts sending a message to $P_2$ at time $A$. $P_1$ spends $o_s$ time units for the send operation. Then, the communication processor interacts with the network and spends $g_s$ time units. After $l$ time units, the message arrives at the destination node. The communication processor on the destination node receives the message. The message becomes available to the processor after $g_r$ time units, and to the user program after an additional $o_r$ time units. The total delay that the message experiences, or the time between the user program issues a send operation and the message becomes available to the user program at the destination processor is the sum of these delays:

$$o_s + g_s + l + g_r + o_r$$

The sending processor is blocked only during the $o_s$ time units (similarly the receiving processor is blocked $o_r$ time units), and duration of the other parts of the delay, the processor is free to perform computation. Similarly, the communication processor is blocked by

[a] Two blocks are incomparable if neither is a successor or predecessor of the other

$g_s$ (or $g_r$) time units. (This limits the amount of and the number of messages a processor can inject and receive from the network per unit time.) Therefore, the $g_s + l + g_r$ part of the remote information access delay, can be potentially overlapped with useful computation.

Each of these parameters has a fixed part and a variable part that depends on the size of messages. As in many studies, we have chosen to model each of these parameters as

$$\alpha + \beta n$$

where $\alpha$ is the startup cost, $\beta$ is the time per data item and $n$ is the number of data items in the message. The time spent in the network, $l$, in our model is also affected by the finite capacity of the network. The capacity limitations are similar to those described by [2]. The finite capacity of the network is modeled by blocking the sender communication processor if the volume of messages traveling in the network is above a threshold. The communication processor has a finite buffer to hold messages deposited by the processor also. If it runs out of buffer space, then the processor is blocked. This model subsumes the LogP model presented in [2].

## 5 SIMULATOR

The simulator consists of three major components: the preprocessor, the parallel machine simulator, and the trace interpreter. The traces may be obtained from a run on a parallel machine or on a uniprocessor emulating a parallel machine.

A simulation session starts with the preprocessing of the execution traces. The output of this stage then is interpreted by the the trace interpreter on the simulated parallel machine model.

### Preprocessor

The Charm/Dagger programming system allows multiple module compilation, i.e., independently compiled Dagger programs can be linked at run time. The Dagger translator produces a separate dependence information for each module. Therefore, dependence information from individual modules and the runtime trace information are reconciled and a single consistent dependence graph and trace information are produced. The preprocessor also converts all timing information to relative times. The traces from Dagger programs contain absolute times. For example, a when-block trace with absolute times might look like this:

```
when-block instance A started at time t1,
sent message B at t2,
when-block instance A ends at t3
```

The simulator uses relative timings:

```
when-block instance A elapsed time t3-t1,
sent message B at t2-t1,
when-block instance A ends
```

After the preprocessing, a when-block record in the trace information forms one entity. The simulator reads a when-block record at a time and processes it. The instances of when-blocks are identified by a quadruple `<p,b,i,r>` where `p` is the processor number, `b` is the static identification of the when-block, `i` is the instance of the dag-chare to which the when-block belongs, and `r` is the reference number. A program may contain many instances of a particular chare (and dag-chare), and a particular instance is identified by this dynamic component, `i`. The reference number is a feature in the Dagger language that has not been discussed in this paper. Within the context of the simulator, it is sufficient to assume that a when-block is completely identified with this quadruple.

### Parallel Machine Simulator

The simulator uses an event-list based approach to simulate the machine model. An event contains the event-time, event-type, and other information depending on the event type. The events are kept in a heap. There is one entry for each processor and communication processor in the heap. Each entry contains a sorted list of events that are to happen on that processor or communication processor. The time stamp of the heap entry is that of the earliest time event in its list. The simulator removes the next event from the heap and processes it until the heap becomes empty. The communication processor events handle network level the message transfers. They contain the necessary information about the communication including message destination, length, priority etc. Processor events are either user events (a when-block execution) or system events such as send or receive a message.

### Interpreting the Traces

Interpreting the traces requires modeling of the Dagger runtime. The simulator has to schedule when-blocks based on the arrival order of messages without violating the dependencies in the same way the Dagger does. The simulator models the Dagger by maintaining three major queues: a scheduling queue, a when-

block-wait-queue, and a when-block-ready-queue. All the incoming messages are buffered in the scheduling-queue. The management of this queue is FIFO by default. It may use other queuing strategies. The simulator supports LIFO (stack), prioritized FIFO, and prioritized LIFO strategies. The prioritized message scheduling can be used in certain applications to decrease the execution time. By adding this feature to the simulator, it is possible to experiment with such applications also. The wait-queue is a list of when-block instances waiting for some messages or completion of other when-blocks. This queue is necessary because a when-block may depend on multiple messages or when-blocks. When all the dependences of a when-block are satisfied, it is moved from the wait-queue to the ready-queue.

The processor continuously fetches the messages from its incoming message queue (where the communication processor puts the messages received from the network) and puts into the scheduling queue. If the incoming message queue is empty, the next message from the scheduling queue is retrieved and the simulator emulates the Dagger runtime to process the message. It checks the dependency graph to determine if any when-block instance depends on this message. If so, then it checks the wait-queue to see if the when-block instance has already been created, otherwise it creates the when-block instance and puts it in the wait-queue. If the arrival of the message causes additional when-block instances to become eligible for execution, those when-blocks are put in the ready-queue for execution. If there are more than one when-block instances in the ready-queue, the first when-block instance from the ready-queue is interpreted by default (however, this queue can be managed differently similar to the scheduling queue). The trace record of the when-block instance is then is read from the trace files and the events that are recorded in the when-block trace are executed in the same sequence. Note that the trace contains only the events that denote sending of messages, synchronization, and the elapsed time between these events. The local time of the processor is incremented by the appropriate delay of each action including blockings due to network load. At the end of the execution of the when-block, the dependence graph is inspected again to see if any other when-block instance is waiting for the completion of the currently executing when-block. The cost of management of when-block-queues and cost of moving messages between various queues are also reflected in the simulation time by user supplied various cost parameters.

## 6 SOME SIMULATION RESULTS

We will present some of simulation studies taken from a larger study [6] to illustrate the impact of a single machine parameter on the performance. The broader study demonstrated that message-driven execution often leads to better performance compared to the traditional message-passing style. In this paper, we only present some of the data from that study to illustrate the utility of the simulation framework.

These studies are intended to analyze and project the trends in a somewhat qualitative manner. For full-fledged performance prediction (e.g., performance of a new, yet to be available, machine), such trend analysis is not adequate. Repeated calibration and validation studies with accurate machine parameters are necessary. This is a topic for future research.

The example code that is used in this simulation study is abstracted and modified from a real application — a communication module in a parallelized version of a molecular dynamics simulation code. Each processor has an array of size $n$ elements. The computation consists of many iterations. Every iteration involves computation of all the elements in each processor locally and then the global sum of each element across all processors, i.e., each processor gets the sum of the first elements, the sum of the second elements etc. The computation of each element and its global sum is independent of other elements with the iteration.

In the traditional message-passing implementation, each processor first computes all the elements of its array, then calls a single global reduction operation (of size $n$) which is a blocking library call usually provided by the message-passing library. The message-driven version, on the other hand, exploits the the fact that global sum of the elements can be done concurrently. Each processor divides its array into $k$ partitions. Then, the global sum of elements in a partition is computed with a non-blocking reduction operation. Thus, it pipelines the global sum operation of $k$ partitions of size $\frac{n}{k}$ each. The traditional message-passing version could divide the arrays into $k$ partitions but it would experience performance loss due to the blocking nature of the reduction operation it invokes.

We gathered traces from both traditional message-passing (which will be referred as traditional-spmd) and message-driven programs on 64 processors, and conducted simulations by changing various machine parameters. Figure 4, illustrates the impact on performance of the two programs as we increased the net-

work latency. The time is reported in terms of simulation time units. In these examples, one simulation time unit corresponds 100 nanoseconds. The graph (a) plots the completion time versus the network latency. That is, $o_s$, $g_r$, $o_r$, and $g_r$ is kept fixed, but the network latency, $\alpha_{net}$, (i.e., $\alpha$ in $l = \alpha + \beta n$) is varied. For $k = 1$, the elapsed time of traditional-spmd and message-driven programs are the same, and as $\alpha_{net}$ is increased, the elapsed time for both programs increases slightly since the communication takes place in one reduction operation only. For pipelined cases (i.e, $k > 1$), the performance of the traditional-spmd program shown by dotted lines deteriorates rapidly with increasing communication latency, whereas the performance of the message-driven program does not deteriorate indicating that it is tolerating the latency. Also notice that, the message-driven performance for $k = 8$ and $k = 64$ is better than the case $k = 1$ (note that the problem size per processor, $n$, is fixed). However, the $k = 64$ case is worse than the one for $k = 8$. This is due to the overhead per message incurred by the message-driven execution. For $k = 64$, the total overhead exceeds the performance benefits achieved for $k = 8$.

To illustrate some other features of the simulator, the second plot in Figure 4 shows the result obtained by varying the the network latency in a random fashion. Randomized variation can be found in ethernet connected workstations for instance. In this experiment, additional random delays (exponentially distributed) were introduced to the network latency. The message-driven program appears to tolerate the unpredictability of communication latencies better than traditional-spmd program as indicated by the slope of the curves.

Figure 5 illustrates how a communication processor differentially impacts the performance of the two programs. The horizontal axis shows the fraction of message passing overhead taken over by the communication processor. Again it was seen that, the message-driven program can exploit the presence of a communication processor better than the traditional-spmd program. As the communication processor handles more of the message delays, the total elapsed time of the two programs decreases. However, the elapsed time for the message-driven program is less than the traditional one, and the rate of decrease in elapsed time is better also. It should be noted that multi-threaded programming style will yield similar performance benefits as message-driven execution.

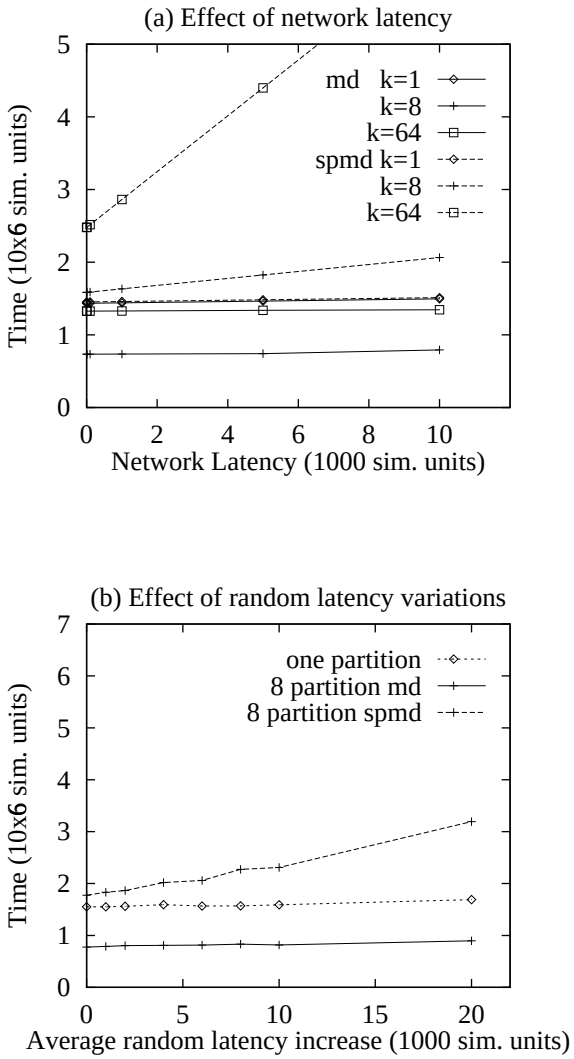


Figure 4: Impact of latency on message-driven and traditional message-passing programs (64 processors)

## 7 SUMMARY

We presented a technique to conduct trace-driven simulation of message-driven programs. The message-driven execution has performance advantages by providing the ability to overlap the latency with computation. However, the simulation of such programs poses problems. The difficulties in carrying out simulation of such programs has been identified. For a specific class of programs using the expressions of the Dagger language, it has been shown that the compile time information and execution traces are enough to achieve accurate simulation of message-driven programs even the runtime conditions changes. The design of the simulator has been presented, and some preliminary performance studies conducted using the simulator were

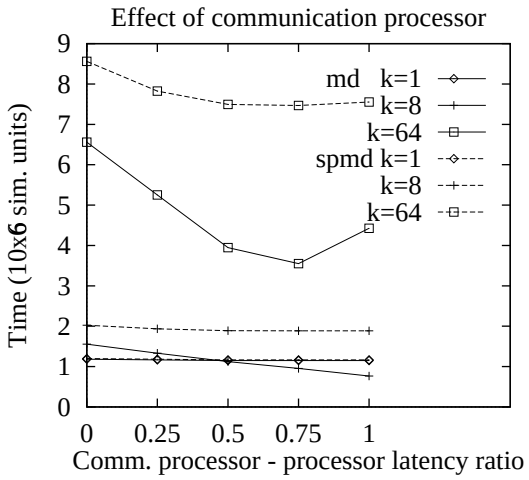


Figure 5: Impact of communication processor on message-driven and traditional message-passing programs

discussed. Results of a larger study conducted using this simulator are presented in [6]. Although the simulator is limited to the programs written in Dagger language, the technique can be applied to programs written in other languages with parallel constructs (such as `parbegin parend`) provided that the programs satisfy the condition described in Section 3 and the compiler produces dependence graph for subcomputations.

### 8 ACKNOWLEDGEMENTS

The authors would like to thank to Sandia National Laboratories for providing access to the nCUBE/2 and Intel Paragon computers.

## References

[1] Chandy, K.M. and Kesselman, C., "CC++: A Declarative Concurrent Object-oriented Programming Notation", Editors Agha, G. et al., *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, 1993, pp281-313.

[2] D.E. Culler et al, "LogP: Towards a Realistic Model of Parallel Computation", *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, May 1993, pp1-12.

[3] H. Davis, S. Goldschmidt, and J. Hennesy, "Multiprocessor Simulation and Tracing using Tango", *Proceedings of the International Conference on Parallel Processing*, Vol II, Aug 1991, pp99-107.

[4] M. Dubois, F.A. Briggs, I. Patil, M. Balakrishnan, "Trace-Driven Simulation of Parallel and Distributed Algorithms in Multiprocessors", *Proceedings of the International Conference on Parallel Processing*, Aug 1986, pp909-915.

[5] C. Eric Wu, et al, "The Design of A Timing Simulator for Distributed Applications", *Proceedings of 1992 International Conference on Parallel and Distributed Systems, Taiwan* Dec 1992, pp50-57.

[6] A. Gursoy, "Simplified expression of message-driven programs and its impact on performance", Ph.D. thesis, University of Illinois at Urbana-Champaign, May 1994.

[7] A. Gursoy, L.V. Kale, "Dagger: combining the benefits of synchronous and asynchronous communication styles", *Proceedings of the International Parallel Processing Symposium*, Cancun, Mexico, Apr 1994, pp590-596.

[8] M.A. Holliday, C.S. Ellis, "Accuracy of Memory Reference Traces of Parallel Computations in Trace-Driven Simulation", *IEEE Trans. TPDS,* Vol.3, No.1, Jan 1992, pp97-109.

[9] J.M. Hsu, P. Banerjee, "Performance Measurement and Trace Driven Simulation of Parallel CAD and Numeric Applications on a Hypercube Multicomputer", *IEEE Trans. TPDS,* Vol.3, No.4, Jul 1992, pp398-412.

[10] L.V. Kale, "The Chare Kernel parallel programming language and system", *Proceedings of the International Conference on Parallel Processing*, Vol II, Aug 1990, pp17-25.

[11] A. Smith, "Cache Memories", ACM Comput. Surveys, Vol.14, No.3, Sep 1992, pp473-530.