TRANSPARENT IMPLEMENTATION OF CONSERVATIVE ALGORITHMS IN PARALLEL SIMULATION LANGUAGES

Vikas Jha
Rajive L. Bagrodia

Computer Science Department
University of California at Los Angeles
Los Angeles, California 90024, U.S.A.

ABSTRACT

Parallel discrete event simulation offers significant speedup over the traditional sequential event list algorithm. A number of conservative and optimistic algorithms have been proposed and studied for parallel simulation. We examine the problem of transparent execution of a simulation model using conservative algorithms, and present experimental results on the performance of these transparent implementations. The conservative algorithms implemented and compared include the null message algorithm, the conditional-event algorithm, and a new algorithm which is a combination of these. We describe how dynamic topology can be supported by conservative algorithms. Language constructs to express lookahead are discussed. Finally, performance measurements on a variety of benchmarks are presented, along with a study of the relationship between model characteristics like lookahead, communication topology and the performance of conservative algorithms.

1 INTRODUCTION

Distributed(or parallel) simulation refers to the execution of a (discrete event) simulation program on parallel computers. A potential for a significant speedup has led to the design of several algorithms for distributed simulation, which are broadly classified into two categories - optimistic and conservative. Performance of these algorithms has been studied on various benchmarks. A survey of most of the existing simulation protocols and their performance studies on various benchmarks appears in Fujimoto (1990).

Bagrodia and Liao (1992a) describes a distributed simulation language called Maisie which attempts to separate the development and representation of the simulation model from the specific simulation algorithm which is used to execute it. It also provides constructs using which the user might optimize the

execution of the model under a particular simulation algorithm. Efficient sequential and parallel optimistic implementations of Maisie have been described in Bagrodia and Liao (1992a), Bagrodia and Liao (1992b). In this paper, we examine the problem of transparent implementation of a conservative algorithm in a simulation language. We use Maisie as a specific example to present our ideas. We show how special constructs can be added to the language to improve the performance under a conservative protocol. We also present a performance study of the implementation using various queuing networks and synthetic benchmarks.

The contributions of this paper are as follows:

- Performance studies of conservative algorithms have primarily used a hardcoding of the simulation protocol into the application, for example, Fujimoto (1987), Nicol (1988), Chandy and Sherman (1989). We show how a simulation model described in an algorithm independent simulation language can be executed using various conservative methods.

- We describe conservative implementations using three different algorithms— null message algorithm (Misra 1986), conditional-event algorithm (Chandy and Sherman 1989), and a new conservative algorithm that combines the preceding approaches. Although, the performance of null message algorithm is generally better than that of conditional-event algorithm, the latter has the nice property of not *requiring* lookahead for progress(under the assumption that events with the same timestamp can be processed in an arbitrary order). A combination of the two has almost the same performance as the null message algorithm and would in addition, also not *require* lookahead for progress. On certain kind of applications, the combination could potentially perform better than the null message algorithm.

- Knowledge of communication topology plays an important role in controlling the null message overhead. Most of the existing work on conservative algorithms assumes a static communication topology. In fact, it is widely believed that the null message algorithm *can not* be used in a dynamically changing topology. However, dynamic process and channel creation can potentially improve the performance of conservative algorithms (Lin 1992). Maisie allows dynamic process and channel creation. We describe how these constructs can be supported with conservative algorithms.

- Lookahead (Fujimoto 1987), which is defined as the ability of a process to predict its future outputs, plays an important role in the performance of a conservative algorithm. We present a more general formulation of lookahead than presented before. We discuss this formulation in the context of Maisie and describe how information can be extracted transparently from Maisie programs to improve the lookahead. We also describe language level features that are provided to the user to further improve the value of lookahead.

- We study the performance of the conservative implementations using a variety of benchmarks. The effect of varying different parameters like lookahead is studied.

The rest of the paper is organized as follows: Section 2 discusses related work. Section 3 describes the various conservative algorithms used. Section 4 briefly describes Maisie. Section 5 describes some of the optimizations for the conservative implementation. Section 6 describes the benchmarks used in the experiments. Results are explained in section 7 , and section 8 gives the conclusions.

2   RELATED WORK

Languages/systems that support conservative simulation protocols include Yaddes (Preiss 1989), SIMA (Rajaei and Ayani 1992), and OLPS (Abrams 1988). Yaddes requires user to use system calls to send null messages, and therefore the simulation protocol is not completely transparent to the user. SIMA, on the other hand, uses synchronous protocols which are radically different from the algorithms used by us. OLPS requires the user to choose different types of processes for different simulation protocols, and hence, is not algorithm independent. Most of these languages don't provide language level constructs to express lookahead and dynamic topology.

Performance of the null message deadlock avoidance algorithm (Misra 1986) using queuing networks and synthetic benchmarks has been studied by Fujimoto (1987). Chandy and Sherman (1989) describe the conditional event algorithm and study its performance using queuing networks. They use null messages in the conditional event algorithm too, but, since their implementation is synchronous (i.e. all LPs carry out local computations followed by a global computation), its performance is quite sensitive to load balancing.

Effect of lookahead on the performance of conservative protocols was studied by Fujimoto (1987). Nicol (1988) introduced the idea of precomputing the service time in order to improve the lookahead. Cota and Sargent (1990) have described the use of graphical representation of a process in automatically computing its lookahead.

3   CONSERVATIVE ALGORITHMS

For the correct execution of a (process based) discrete-event simulation, the underlying system has to ensure that all messages to a Logical Process(LP) are processed in an increasing timestamp order. Distributed simulation algorithms are broadly classified into *conservative* and *optimistic* based on how they ensure this. Conservative algorithms, in general, achieve this by not delivering a message of timestamp t (and hence blocking the process if it can't proceed without the message) until it can ensure that the process will not receive any other message with a timestamp lower than t. Optimistic algorithms, on the other hand, allow events to be processed (possibly) out of timestamp order. The causality errors are corrected by rollback and recomputations. In this section, we describe three conservative algorithms. We assume that the communication channels are FIFO, and messages with the same timestamp can be processed in an arbitrary order.

At any simulation instant, let n be the next message, with timestamp t_n, to be processed by an LP. In conservative protocols, n will have to wait for some time after its arrival, until the LP can make sure that there won't be any messages with smaller timestamps, before it can be processed. This waiting period, which is the main overhead in conservative protocols, can be reduced by estimating t_n in advance. **Earliest Input Time(EIT)** for an LP, at a given simulation instant, is a lower bound on t_n. Under conservative protocols, therefore, an LP can not *process* any messages with timestamp greater than EIT. Different protocols compute the value of EIT differently. In general, efficiency of a protocol is deter-

mined by how close the value of EIT is to the actual t_n. In the ideal case, if EIT is always equal to t_n, the waiting period would be zero for every message, and the simulation protocol would be optimal. We now describe how EIT is computed in the three conservative protocols that we have studied.

3.1 Null Message Algorithm

Earliest Output Time(EOT), for an LP, at a given simulation instant, is a lower bound on the timestamp of the next message sent by the LP. It is equal to EIT plus the value of **lookahead**(described in detail in section 5) for the process at that simulation instant. Every LP uses *null* messages to inform the LPs, corresponding to all its *output* channels, of the value of EOT whenever it changes. The EIT of a process is simply equal to the *minimum of the last EOTs received on every input channel.* Note, therefore, that the knowledge of **communication topology** is crucial for the performance of null message based algorithms. Null message overhead can be reduced by piggybacking null messages with regular messages, and by requiring that the entities send null messages only when they have no regular messages to process. A non zero lookahead is required (Misra 1986) in every cycle of entities to ensure that the simulation model doesn't deadlock(i.e. EIT keeps advancing).

3.2 Conditional-Event Algorithm

Consider an instantaneous global snapshot of the system. We define **Earliest Conditional Output Time(ECOT)** for an LP to be the timestamp of its earliest *unprocessed* input plus the minimum timestamp increment(lookahead), if any. The minimum over the values of ECOT of all the LPs and the timestamps of all the messages in transit is the (Globally) Earliest Conditional Event Time in the system, and gives an estimate for the EIT of every LP in the system. Note that the computation of Earliest Conditional Event Time is similar to GVT calculation in optimistic algorithms. Hence, any of the the GVT computation algorithms can be used.

3.3 A New Algorithm

Assuming that messages with same timestamp may be processed in an arbitrary order, the conditional-event algorithm doesn't *require* lookahead for progress. However, in presence of good lookahead, the null message algorithm performs much better than the conditional-event algorithm(which requires frequent global computations to ensure progress).

We superimpose the null message protocol on top of the conditional event algorithm. The conditional event algorithm uses a GVT algorithm that doesn't require freezing of normal computation in order to calculate the Earliest Conditional Event Time(hence allows the null message protocol to perform unhindered). The EIT for any process is, therefore, the maximum of the estimates computed by the two algorithms. This method has the potential of combining the efficiency of the null message algorithm in presence of good lookahead with the ability of the conditional event algorithm to execute even without lookahead(a scenario in which null message algorithm alone will deadlock).

4 MAISIE

Maisie (Bagrodia and Liao 1992a) is a C based distributed simulation language. The central construct introduced by the language is that of an entity. A Maisie entity-type models physical objects (or a collection of objects) of a given type. An entity-instance, henceforth referred to simply as an entity, represents a specific object. Interactions among the physical objects in the system are modeled by message exchanges among the corresponding entities.

An entity may be created and destroyed dynamically. An entity is created on a specific processor and cannot be migrated subsequently. Message-communication among the entities is based on buffered message-passing. An entity-type specifies the types of messages that may be received by it. A message-type consists of a name and a list of parameters. Every entity has a unique message-buffer. A message is deposited in the message-buffer of an entity on the execution of an *invoke* statement. Each message carries a timestamp, which corresponds to the simulation time at which the corresponding invoke statement was executed. Messages sent by one entity to another are delivered to the destination buffer in FIFO order.

An entity accepts messages from its message-buffer by executing a *wait* statement. The wait statement has two components: an integer value called waittime (t_c) and a Maisie statement called a resume block – a (non-empty) sequence of resume statements. A resume statement is like a guarded command, where the guard consists of a message-type (say m_t) and an optional boolean expression(say b_i). A resume statement is said to be *enabled* if the message-buffer contains a message of type m_t, which if delivered to the entity would cause b_i to evaluate to *true*; the corresponding message is called an *enabling* message. If the buffer contains one or more enabling message, in

the most commonly used form of the wait statement, the message with the earliest timestamp is removed from the buffer and delivered to the entity. If two enabling messages have the same timestamp, they are processed in an arbitrary order. By selecting the guards appropriately, the wait statement may be used to ensure that an entity accepts a message from its input buffer only when it is ready to process the message.

If the buffer does not contain any enabling messages, the entity is suspended for a *maximum* duration equal to its wait-time t_c; if omitted, t_c is set to an arbitrarily large value. If no enabling message is received in the interval t_c, the entity is sent a special message called a *timeout* message. An entity must accept a timeout message that is sent to it. A non-blocking form of receive may be implemented by specifying $t_c=0$.

If a wait statement contains exactly one resume statement and its guard specifies timeout as the message-type, the entity will resume execution only when it receives a timeout message after the wait-time specified in the statement has elapsed. As this timeout message cannot be cancelled, it is referred to as an unconditional timeout message. Wait statements that schedule an unconditional timeout message are used frequently and are often abbreviated by a *hold* statement. The example at the end of the section illustrates their use in a simulation. If the wait statement contains multiple resume statements, only one of whose guards include timeout as the message-type, the entity may resume execution on the receipt of a message other than timeout. Thus, the timeout message scheduled by such statements is referred to as a conditional timeout message.

As a simple example, consider the simulation of a pre-emptible priority server in Maisie. In the physical system, the server receives two types of requests, respectively referred to as *high* and *low*, where the requests of the first type have a higher priority and can interrupt the server if it is currently serving a request of type *low*. Figure 1 describes the Maisie model of the server. In the interest of brevity, the program ignores issues concerned with the initiation and termination of the simulation. The maisie code for the source entities is also omitted.

5 OPTIMIZATIONS

Two factors which affect the performance of conservative algorithms most are the **knowledge of the exact communication topology**, and **lookahead**. Since the conditional event algorithm finds the earliest conditional event over the *entire system*, knowl-

```
1   entity server { mean }
2      int mean;
3   { message high { ename hisid; } ;
4     message low { ename hisid; } ;
5     ename jobid;
6     int rem_time=MAXINT, dep_time, busy=0;
7     for(;;)
8     { wait rem_time for
9        { mtype(high)
10          { if(busy)
11               rem_time=dep_time − sclock();
12            hold(expon(mean));
13          { if(busy)
14               dep_time=rem_time + sclock();
15            invoke msg.high.hisid with done; }
16        or mtype(low) st(!busy)
17          { busy=1; jobid=msg.low.hisid;
18            rem_time=expon(mean);
19            dep_time=sclock() + rem_time; }
20        or mtype(timeout)
21          { busy=0; rem_time=MAXINT;
22            invoke jobid with done; }
23     }
24   }
25 }
```

Figure 1: Maisie Model of Priority Server

edge of communication topology affects only the null message based algorithms. In this section, we discuss the language level constructs provided in Maisie to support these optimizations.

5.1 Dynamic Communication Topology

Any conservative method that uses null messages requires the knowledge of the communication topology. In absence of this knowledge, the null messages would have to be broadcast which would severely degrade the performance. Since, typically, the communication pattern keeps changing over the course of the simulation, having a static communication topology, which would necessarily have to encompass all the channels that exist at any point during the simulation, would mean that each LP, at any given time, might be synchronizing (using null messages) with a large number of LPs that its not going to be interacting with in the near future. Allowing dynamic process and channel creation(and destruction), therefore, can improve the performance considerably (Lin 1992). However, it is widely believed that null message based algorithms can't support these constructs.

The main problem in allowing dynamic channel creation in conservative schemes is illustrated by the following example: In Figure 2, there already exists a
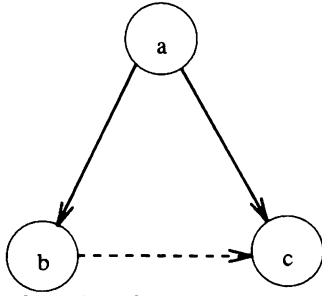
Figure 2: Creating Channels Dynamically

channel from a to b and from a to c. A channel is to be created from b to c at time t(i.e. the first message on that channel will have a timestamp equal to t). If the information to add b to its *source set* reaches c after c's local simulation clock is past time t, then, it could result in a violation of causality(i.e. the message from b to c might arrive in the past of c). Also, if entity b didn't add c to its *destination set* until after simulation time t, it could lead to a deadlock, since, b would inform(through null messages) only the entities currently in its *destination set* about the value of its EOT, whereas, c would start waiting for b's EOT at or before time t.

At the time of process creation, Maisie automatically creates a channel from the creator to the created process. Any other channels have to be created or destroyed explicitly by the entities by (locally) adding or deleting entities from their source or destination sets. Four constructs, namely, *add_source*, *add_destination*, *del_source*, and *del_destination* are available to an entity for this purpose. In order to avoid the potential problem of causality violation as described above, if the earliest message on a channel from entity b to c has a timestamp t, then, the user has to ensure that the following conditions are satisfied:

1. b should add c to its destination set before or at simulation time t.

2. c should add b to its source set before or at simulation time t.

First condition is easily satisfied, since, b can simply execute an *add_destination(c)* just before it sends a message to c . In order to satisfy the second condition, c needs to be informed about the ename of the entity b before or at time t(normally, in Maisie, the destination doesn't need to know source's name). In most applications, b and c are created by the same entity, say, a(typically the driver entity), and the channel from b to c is created at the simulation time t, when the two entities are created (see Figure 2). In such a case, a can send the ename of b to c right after creating the two entities, thus ensuring that it reaches c at time t.

The only responsibility of the user is to satisfy conditions (1) and (2). The actual synchronization with the source-set and destination-set is a part of the algorithm used, and hence is transparent to the user.

5.2 Lookahead

Informally, lookahead is defined as the ability of a process to *look ahead into the future*. Quantitatively, we define $lookahead(t)$ for a process, at simulation time t, to be the value of $EOT - t$ after *all and only* the inputs to the process with timestamp *less* than t have been processed by the process (for simplicity, we assume that the EOT, and hence lookahead, is same on all output channels). Note that the value of lookahead depends on the semantics of process behavior(local factor), and the message arrival pattern(global factor). The above definition is similar to the one used by Fujimoto (1987). They define the lookahead for a process to be t', if upon having processed all messages with timestamp t or less, it can predict all future messages with timestamp $t + t'$ or less. However, they assume the lookahead to be fixed throughout the simulation which, we believe, is inadequate to explain the lookahead characteristics of most of the applications.

An eager server (Fujimoto 1987) is defined to be one in which the departure event is scheduled(i.e. the corresponding output message is sent) as soon as the arrival event for a job is processed (only possible for FCFS servers). A lazy server , on the other hand, waits until the simulation time advances past the departure time before sending the output message. Consider the lookahead of an eager FCFS server in a Closed Queueing Network. If the message arrival pattern and the service time distribution is such that the number of messages received with timestamp less than t is n and the server is never idle during the time interval $[0,t]$, then, the value of $lookahead(t)$, for the eager server, is equal to $\sum_{i=0}^{n-1} serv\_time_i - t$. If, in addition, the server also precomputes the service time of the next job (Nicol 1988), the value of $lookahead(t)$ is equal to $\sum_{i=0}^{n-1} serv\_time_i + serv\_time_n - t$. The $lookahead(t)$ of the lazy server , irrespective of the message arrival pattern, is zero, for all t. For a lazy server which precomputes the service time of the next job, $serv\_time_n$, lookahead depends on the message arrival pattern. If the message arrival pattern and the service time distribution is such that the server is idle at simulation time t, $lookahead(t)$ is equal to $serv\_time_n$. If the server is busy with a job that has a remaining service time left of r_time, then $lookahead(t)$ is equal to r_time.

Clearly, In order to be compared across applica-

tions, the absolute value of lookahead has to be normalized with respect to the service time(timestamp increment) (Fujimoto 1987).

Now, we discuss how the value of EOT is calculated for Maisie entities (which determines the value of *lookahead* at any instant). Every Maisie entitiy has a *Clock* variable associated with it. Whenever an input message is *processed* by an entity, the value of its *Clock* is updated to the maximum of its current value and the timestamp of the message. In Maisie, the timestamp of a message is equal to the *Clock* value of the sender entity. Since the value of *Clock* increases monotonically, an obvious estimate of EOT, at any simulation instant, is equal to *Clock*. Therefore, $lookahead(t)$ is equal to $Clock_t - t$, where $Clock_t$ is the value of *Clock* when all and only the inputs with timestamp less than t have been *processed*(or are ineligible to be *processed* by the current selective receive command) and the entity is waiting for the next input. The following subsections outline how this estimate of EOT can be further improved upon.

5.2.1 Transparent Extraction of Lookahead

$hold(t_c)$ statement is frequently used in Maisie programs to model servicing of jobs. Semantically, $hold(t_c)$ is equivalent to a $wait(t_c)$ statement with the only resume condition being *timeout*. Therefore, upon processing a $hold(t_c)$ statement, the *Clock* can be incremented by t_c time units. It is easy to see how in applications which frequently use *hold* statement, for example, the code for an eager FCFS server, the value of $Clock_t$ can progress far beyond the value of t, thereby improving the lookahead estimate $Clock_t - t$.

5.2.2 User Specified Lookahead

If the user is able to guarantee that the minimum increment to *Clock* between *processing* the next input and sending the corresponding output is equal to δ, then the estimate of EOT can be improved to $Clock + \delta$. Maisie provides a special function call, **lookahead**, to allow the user to express this minimum timestamp increment in form of an expression consisting of local variables and the function call **sclock()** which gives the current value of the *Clock* for the entity. This expression is evaluated whenever its value is used by the underlying system. In the simple case of an FCFS server the expression could simply be *ntime*, where the variable *ntime* contains the precomputed service time of the next job. The expression for the pre-emptible priority server is more complicated. and is shown in Figure 3. In presence of the user defined lookahead, therefore, the estimate for $lookahead(t)$

```
1   #define MIN(a,b)    ((a < b)?a : b)
2   entity server { mean }
3       int mean;
4   {   message high { ename hisid; } ;
5       message low { ename hisid; } ;
6       ename jobid;
7       int rem_time=MAXINT, dep_time,
                next_time, next_next_time, busy=0;
8       lookahead(busy ? MIN(next_time,dep_time
            −sclock())  :  MIN(next_time,next_next_time));
9       next_time=expon(mean);
10      next_next_time=expon(mean);
11      for(;;)
12      {   wait rem_time for
13          { mtype(high)
14              { if(busy)
15                  {   rem_time=dep_time − sclock();
16                      dep_time=dep_time + next_time; }
17                  hold(next_time);
18                  next_time=next_next_time;
19                  next_next_time=expon(mean);
20                  invoke msg.high.hisid with done; }
21              or mtype(low) st(!busy)
22              { busy=1; jobid=msg.low.hisid;
23                  rem_time=next_time;
24                  next_time=next_next_time;
25                  next_next_time=expon(mean);
26                  dep_time=sclock() + rem_time; }
27              or mtype(timeout)
28              { busy=0; rem_time=MAXINT;
29                  invoke jobid with done; }
30          }
31      }
32  }
```
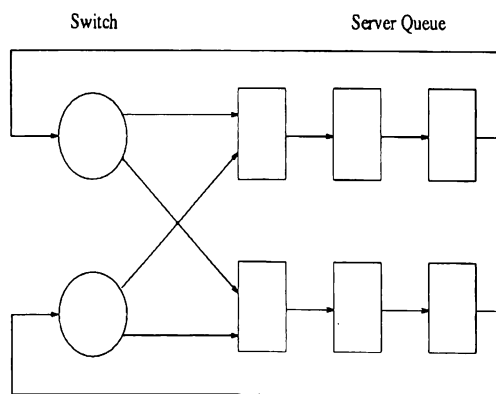
Figure 3: Maisie Code for Pre-emptible Priority Server Incorporating User Defined Lookahead

improves to $Clock_t + \delta_t - t$, where δ_t is the value of the lookahead expression at t.

6   EXPERIMENTS

Two sets of experiments, one consisting of queuing network simulations and the other using synthetic benchmarks, were carried out to evaluate the performance of the conservative implementations.

The Closed Queueing Networks(CQN), used in our experiments, consist of N switches. Each switch has a tandem queue of Q servers(note that the server process includes a queue where the incoming jobs are stored before being processed) associated with it. Each switch routes the jobs to the first server in any one of the tandem queues, with

Figure 4: Closed Queueing Network(N=2, Q=3)

7.1 Closed Queueing Network Experiments

The maisie model of the CQNF network(Figure 4), called CQNF1, models each FIFO server by a separate Maisie entity. Similarly, each switch is modeled by a separate entity. For the parallel implementation, each switch entity and the associated queue entities are allocated to one processor. Here, we summarize some of the main results. A more detailed description can be found in Jha and Bagrodia (1993).

Figure 5 shows the variation of speedup, using 16 processors, with the number of jobs initially at each switch, for the three algorithm(null message, conditional event, and the combination of the two).

equal probability. Each server services the job, with a shifted-exponential service time distribution(a shifted-exponential distribution is chosen so that the minimum lookahead for every entity is non-zero, thus preventing a potential deadlock situation in the null message protocol) and sends it to the next server in the queue, the last server in the queue sending it back to the unique switch it is associated with. The topology of the network, for 2 switches, is shown in Figure 4. Each switch has J jobs initially. The simulation is carried out up to simulation time H. Two variations of the above CQN model are considered - CQNF, where every server is First-come-first-serve, and CQNP, where every server is a Pre-emptible priority server. In the CQNP model, a fixed fraction of jobs are HIGH priority and the rest are LOW priority. The second set of experiments used synthetic benchmarks. These benchmarks consist of closed networks of processes with fixed number of messages circulating between them. Each process in the network processes the messages it receives in the FCFS order with a shifted-exponential service time. Different topologies can be selected by varying benchmark parameters.

7 RESULTS

All the experiments were carried out on an implementation of Maisie on Symult 2010 hypercube where each node uses a Motorola 68020 cpu and has 4MB of main memory. All the programs were written in Maisie. The programs used for the parallel implementations were the same as the ones used for sequential implementation, except for (a). explicit assignment of Maisie entities to specific nodes of the multicomputer, (b). code to create the source and destination sets for each entity, and (c). specification of lookaheads. The speedups were calculated with respect to the sequential version(using the Global Event List algorithm implemented using splay trees) running on one node of the multicomputer.
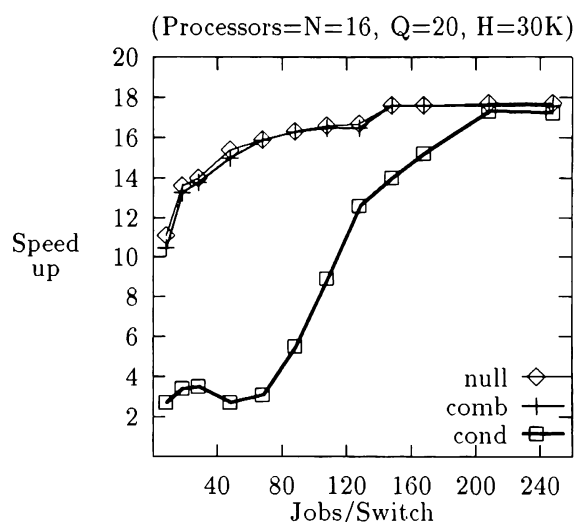

Figure 5: CQNF1: Speedup vs. Jobs/Switch

As shown in the figure, the performance of the null message algorithm is much superior to the conditional event algorithm for both the experiments. This can be attributed to the high overhead of the global communication required to compute the next event time in case of the conditional event algorithm. However, the performance gap between the two narrows considerably for higher values of Jobs/Switch, since, processes have more jobs to process between successive global computations in case of conditional event algorithm resulting in a better computation to overhead ratio. The combination of null message and conditional event algorithms performs almost as well as the null message algorithm in both the cases.

Note that 1 node execution of any of the three algorithms is faster than the global event list algorithm(which is used as the basis to calculate the all the speedups reported in this paper) and higher than linear speedup is observed in many cases. This is because the global event list algorithm executes

events in strictly timestamp order across all processes, whereas in case of conservative algorithms, for good lookahead processes, a number of events may be executed on the same entity before other events with lower timestamp are executed on a different process. This results in fewer context switches. Also, since the context switching overhead is not linear in terms of number of processes, the total overhead decreases when they are divided over many processors.
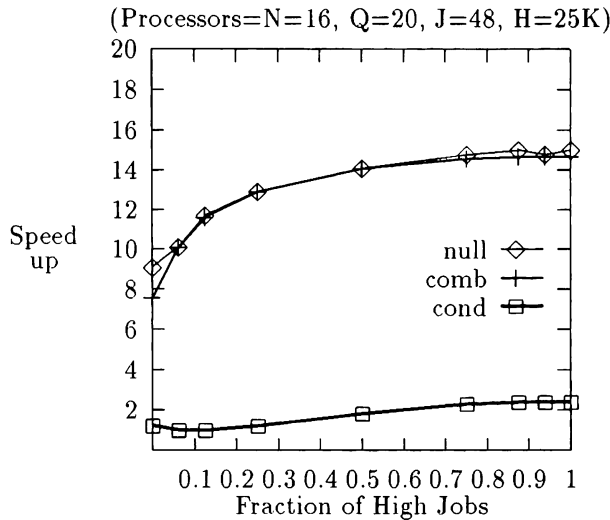

Figure 6: CQNP: Speedup vs. Fraction of HIGH Jobs

Figure 6 plots the speedup with respect to the fraction of HIGH priority jobs in the CQNP experiment(same as CQNF1 with the FIFO servers replaced by priority servers). While processing the high priority jobs, the code uses *hold(service_time)* instruction to model the servicing of the job because the HIGH priority jobs can't be pre-empted. This allows the transparent extraction of lookahead to take place. Hence, increasing the fraction of HIGH jobs should improve the performance. This expected behavior is confirmed by the figure. When all the jobs in the system are HIGH, performance is similar to that of CQNF1, since, the priority servers behave like FIFO servers in such a case.

In order to study the effect of the user defined lookahead, we repeat the CQNP experiment without the user defined lookahead, and the results are shown in Figure 7. As explained before, the *null message algorithm deadlocks in absence of the user defined lookahead*(the transparent lookahead is not guaranteed to break the deadlock in general). As predicted, the new(combination) algorithm is able to execute even in absence of a lookahead guarantee in every cycle, and is able to utilize (transparent) looka-
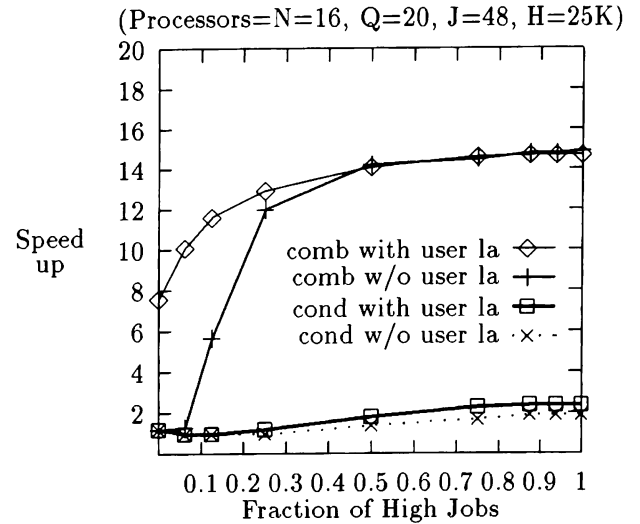

Figure 7: Effect of User Defined Lookahead(CQNP): Speedup vs. Fraction of HIGH Jobs

head where its available(when the fraction of HIGH jobs is high. Note that the lookahead in every cycle is still not guaranteed to be non zero). Comparison of Figure 6 and Figure 7 reveals that presence of user defined lookahead improves the performance dramatically when the transparent lookahead is minimal(i.e. low fraction of HIGH jobs), but, the improvement is negligible when the transparent lookahead is high. Presence of user defined lookahead marginally improves the performance of conditional event algorithm too. This is because we utilize the user defined lookahead in computing a better estimate of the globally earliest conditional event.

7.2 Synthetic Benchmark Experiments

In order to study the effect of specific network characteristics like lookahead, communication topology, and processes per node on the performance of the simulation, we used synthetic benchmarks. Here we only report the results of the lookahead. Rest of the results are summarized in Jha and Bagrodia (1993).

7.2.1 Lookahead

We study the effect of improving the lookahead of a system on the performance in two ways: one in which lookahead characteristics of all the entities in a simulation are the same and are improved across different simulations (lookahead in homogeneous networks), and the other in which some of the entities in the simulation have good lookahead characteristics and others have poor lookahead characteristics,

with the proportion of each type being varied across different simulations (lookahead in non-homogeneous networks). The effect of changing lookahead in a network is closely related to its communication topology. We choose a simple topology, namely, a ring of entities. Each entity is an FCFS server.

Lookahead in homogeneous networks: As noted before, an FCFS server can be programmed as a lazy server or an eager one, and with or without precomputed service time as the lookahead. In order to further vary the degrees of lookahead in the synthetic workload, we express only a fraction, called LAF, of the precomputed service time as lookahead(using the Maisie constructs to specify lookahead). Thus, although, the application knows the amount of timestamp increment on the next message that it would process, it expresses only a fraction of it. In the studies done by Fujimoto (1987), the process knows(and expresses as lookahead) only the minimum possible value of the timestamp increment. The ratio of mean timestamp increment and the minimum possible timestamp increment is defined as the Lookahead Ratio(LAR). Therefore, LAF, as defined above, corresponds to the inverse of LAR. Fujimoto varies LAR by changing the service time distribution(hence the ratio of mean to minimum service time), whereas in our case LAF is specified directly by the user(and is independent of the service time distribution).

Figure 8 shows how the speedup(on 16 processors) varies with the value of LAF. For the case of lazy server, the speedup improves dramatically as we increase LAF from 0.1 to 1.0. Note that we choose the minimum value of LAF to be non-zero since a zero value for LAF might lead to a deadlock in case of the null message algorithm. The speedup of the eager server is not affected much because of an increase in LAF. This is because the lookahead of an eager server is very good even without the precomputed service time (as explained before) and presence of precomputed service time as lookahead doesn't help appreciably. In fact, in some cases, the performance might even degrade slightly because of increased null message overhead.

Lookahead in non-homogeneous networks: We use an eager server with an LAF of 1.0 to represent a good lookahead entity, whereas a lazy server with LAF of 0.1 represents a bad lookahead entity.

Figure 9 shows how the speedup(on 16 processors) varies as the number of consecutive bad lookahead nodes are increased(a connectivity of n implies that each process is connected, via an output channel, to the next n processes in the ring). All the curves show a gradual degradation in performance as bad lookahead nodes are introduced, instead of a sharp decline.
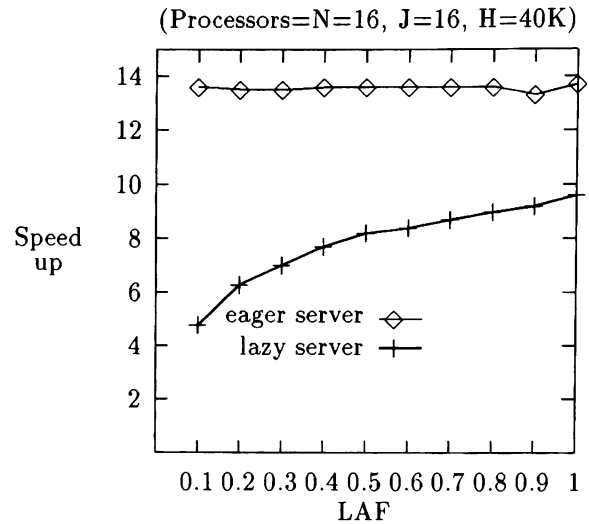

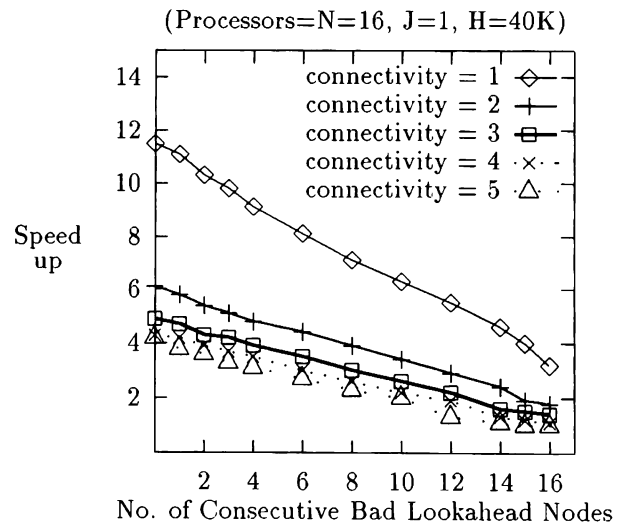Figure 8: Homogeneous Lookahead(Synthetic): Speedup vs. LAF

Figure 9: Non-homogeneous Lookahead(Synthetic): Speedup vs. No. of Bad Lookahead nodes

8 CONCLUSION

An important goal of parallel simulation research is to facilitate its use by the discrete-event simulation community. We have designed a simulation language called Maisie which separates the simulation model from the specific algorithm (sequential or parallel) that is used to execute the model. This paper addressed the problem of transparent implementation

of conservative algorithms for parallel simulation languages. In particular, it describes how three different conservative algorithms can be implemented transparently under the Maisie simulation language.

The paper also described how conservative methods can be implemented to handle dynamic communication topologies, in a restricted way. Previous studies of conservative implementations have used a static communication topology. If the communication pattern in the model varies dynamically, this assumption leads to sub-optimal performance. We describe language constructs to ensure that topological changes are made consistently by the run-time system. Lastly, the paper describes how certain types of lookahead behavior can be extracted transparently by the simulation system. It also introduces language constructs that can be used by a programmer to specify the lookahead behavior of a specific object.

The three algorithms that were studied include the null message algorithm, the conditional event algorithm, and a new algorithm that combines the preceding approaches. Maisie models were developed for standard queuing network benchmarks. Various configurations of the model were executed using the three different algorithms. The implementations were optimized to exploit the lookahead properties of the models. The benchmarks were used to compare the performance of the three algorithms and were also used to evaluate the effect of variations in lookahead characteristics on the performance of the algorithms.

ACKNOWLEDGMENTS

REFERENCES

Abrams, M. 1988. The object library for parallel simulation (OLPS). In *Proceedings of the 1988 Winter Simulation Conference*, 210-219, December 1988.

Bagrodia, R. L., and W. T. Liao. 1992a. A Language for Iterative Design of Efficient Simulations. Technical Report UCLA-CSD-920044, CSD, UCLA, Los Angeles, October 1992.

Bagrodia, R. L., and W. T. Liao. 1992b. Transparent Optimizations of Overheads in Optimistic Simulations. In *Proceedings of the 1992 Winter Simulation Conference*, 637-645, December 1992.

Chandy, K. M., and R. Sherman. 1989. The Conditional Event Approach to Distributed Simulation. In *Proceedings of the SCS Simulation Multiconference on Distributed Simulation*, 93-99, March 1989.

Cota, B. A., and R. G. Sargent. 1990. A Framework for Automatic Lookahead Computation in Conservative Distributed Simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, 56-59, January 1990.

Fujimoto, R. M. 1987. Performance Measurements of Distributed Simulation Strategies. Technical Report UUCS-87-026a, Computer Science Department, University of Utah, Salt Lake City.

Fujimoto, R. 1990. Parallel Discrete Event Simulation. *CACM*, Vol. 33, No. 10, 30-53, Oct. 1990.

Jha, V., and R. L. Bagrodia. 1993. Transparent Implementation of Conservative Algorithms in Parallel Simulation Languages. Technical Report UCLA-CSD-93007, Computer Science Department, UCLA, Los Angeles, May 1993.

Lin, J. M. 1992. Exploiting Dynamic Topological Information to Speed Up Concurrent Multicomputer Simulation. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, Jan. 1992.

Misra, J. 1986. Distributed Discrete-event Simulation. *ACM Computing Surveys*, Vol. 18, No. 1, 39-65, March 1986.

Nicol, D. M. 1988. Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks. *SIGPLAN Not. 23, 9*, 124-137, September 1988.

Preiss, B. R. 1989. The Yaddes distributed discrete event simulation specification language and execution environments. In *Proceedings of 1989 SCS Multiconference on Distributed Simulation*, 139-144, March 1989.

Rajaei, H., and R. Ayani. 1992. Language Support For Parallel Simulation. In *Proceedings of the PADS 92*, 191-192, January 1992.

AUTHOR BIOGRAPHIES

VIKAS JHA is a Ph.D. student in the Computer Science Department at UCLA. His research interests include parallel simulation, parallel languages, and distributed algorithms.

RAJIVE L. BAGRODIA received the B.Tech. degree in Electrical Engineering from the Indian Institute of Technology, Bombay in 1981 and the M.A. and Ph.D. degrees in Computer Science from the University of Texas at Austin in 1983 and 1987 respectively. He is currently an Associate Professor in the Computer Science Dept. at UCLA. His research interests include parallel languages, parallel simulation, distributed algorithms, and software design methodologies. He was selected as a 1991 Presidential Young Investigator by NSF.