Implementation and Performance Evaluation of Conservative Algorithms in Parallel Simulation Languages

Rajive L. Bagrodia

Vikas Jha

Abstract

We examine the problem of efficiently executing a simulation model written in an algorithm-independent simulation language, using a variety of parallel conservative algorithms. 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 a restricted form of dynamic topology can be supported by conservative algorithms. Language constructs to express lookahead are also discussed. We evaluate the performance of these algorithms using two kinds of benchmarks. A queueing network simulation which is commonly used to evaluate parallel simulation protocols, and a set of synthetic benchmarks that help us study relationships between the performance of conservative protocols and application characteristics such as lookahead, communication topology, message population, and computation granularity.

Keywords: Discrete-event Simulation, Parallel and Distributed Simulation, Conservative Algorithms, Parallel Simulation Languages, Lookahead, Parallel Computing.

1 Introduction

Distributed(or parallel) simulation refers to the execution of a (discrete event) simulation program on parallel computers. A potential for 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 many existing simulation protocols and their performance studies on various benchmarks appears in [14, 16].

A number of languages and environments have also been defined for parallel discrete-event simulations; [6] is a recent survey of available tools. Most existing software for parallel simulation use a single (class of) simulation protocol(s). A parallel language called Maisie[4] has been developed to separate the design of a discrete-event simulation model from the underlying synchronization protocol, sequential or parallel, that is used for its execution. Efficient sequential and parallel optimistic execution of Maisie models have been described previously [2]. In this paper, we examine the problem of using parallel conservative protocols to execute a parallel discrete-event simulation model written in an algorithm-independent simulation language. We describe an implementation that uses three different conservative algorithms, namely, null message algorithm [10], conditional-event algorithm [11], and a new conservative algorithm that combines the preceding approaches. The implementation runs on three different architectures : Symult 2010 – an older MPP, IBM SP – a state of the art MPP, and SPARC 1000 – a state of the art SMP.

The primary contributions of this paper are as follows:

- To the best of our knowledge, this is the first comparative study of conservative protocols at the level of a simulation language, as opposed to an evaluation where the synchronization protocol is directly embedded in the application[15, 22, 11]. This separation permits a fairer comparison of the different synchronization protocols for a given application.

- Although performance of the null message schemes has been studied exhaustively in the past, this is the first comprehensive study of the conditional event algorithm. We found that over much of the parameter space, the null message algorithm generally performed better than the conditional-event algorithm, although the latter has the nice property of not *requiring* lookahead for progress. The combined algorithm was found to have almost the same performance as the null message algorithm. As this algorithm does not *require* lookahead for progress, it is an ideal candidate to execute models whose lookahead properties are unknown or hard to determine.

- We describe language level constructs to express lookahead and communication topology. Knowledge of communication topology of the application can play an important role in controlling the null message overhead. Most existing work on conservative algorithms assumes a static communication topology. However, dynamic process and channel creation can potentially improve the performance of conservative algorithms[19]. We describe how to support dynamic process creation and a restricted form of dynamic topology with conservative algorithms. We also provide experimental results that illustrate their utility.

- We discuss the various issues involved in the implementation of a parallel conservative runtime system. Different alternatives are discussed, along with their performance implications.

- We present a comprehensive performance evaluation of the system using queueing network simulations and parameterized synthetic benchmarks. The latter are used to evaluate the effect of varying different parameters like lookahead, computation granularity, message population, and communication topology on the performance. In addition to these benchmarks, the implementation has also been used to execute gate-level and switch-level VLSI simulations [7, 3, 12] and simulation of network protocols [5]. To the best of our knowledge, this is the only language level conservative simulation system that has demonstrated speedup across a variety of applications, architectures, and protocols.

The remainder of the paper is organized as follows: Section 2 describes the conservative algorithms used. Section 3 briefly describes Maisie. Section 4 describes some of the optimizations for the conservative protocols. The details of our implementation are discussed in section 5. Section 6 describes the experiments and the results. Section 7 discusses related work, and section 8 gives the conclusions.

2 Conservative Algorithms

In the message-based approach to discrete event simulation, each Physical Process (PP) is modeled by a Logical Process (LP). LPs communicate with each other using messages which correspond to the events in the physical system. For the null message protocol, we assume that each LP knows the identity of the LPs that it can communicate with. For any LP p, we use the term $dest\text{-}set_p$ ($source\text{-}set_p$) to refer to the set of LPs to which p can send messages (receive messages from).

For the correct execution of a 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 enforce this. Conservative algorithms achieve this by not allowing an LP to process a message with timestamp t until it can ensure that the LP will not receive any other message with a timestamp lower than t. Optimistic algorithms, on the other hand, allow events to be potentially processed out of timestamp order. Causality errors are corrected by rollbacks 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.

In order to simplify the description of the algorithms, we define the following terms. All of them are defined for an LP p at real time r.

- **Earliest Input Time**($EIT_p(r)$): Lower bound on the (logical) timestamp of any message that p may *receive* in the interval (r, ∞).
- **Earliest Output Time**($EOT_p(r)$): Lower bound on the timestamp of any message that p may *send* in the interval (r, ∞).
- **Earliest Conditional Output Time**($ECOT_p(r)$): Lower bound on the timestamp of any

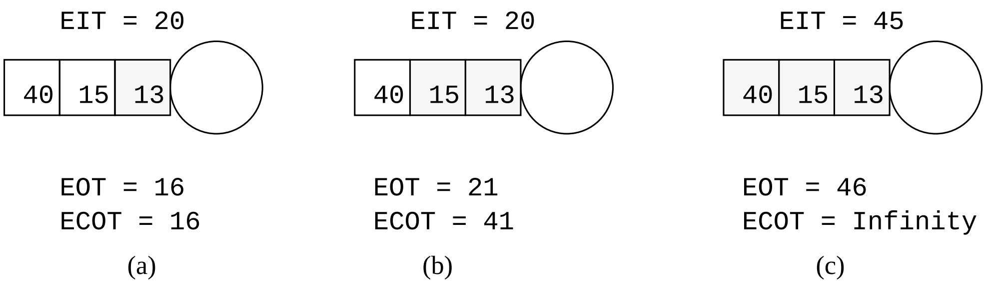

Figure 1: The computation of EOT and ECOT: Shaded areas represent processed messages. LP is modeling an FCFS server with a minimum service time of 1 time unit.

message that p may *send* in the interval (r, ∞) *assuming that p will not receive any more messages in that interval.*

For any given LP, the value of EOT and $ECOT$ depends on the value of EIT, the unprocessed input queue, and the semantics of the process that the LP is modeling. Figure 1 illustrates how EOT and $ECOT$ is computed for an FCFS server with a minimum service time of 1 time unit. Let T refer to the earliest unprocessed input timestamp. In (a), since $T < EIT$, both EIT and $ECOT$ are equal to T plus the minimum service time. In (b), EOT is equal to EIT plus the minimum service time because the LP may still receive inputs with timestamp less than T, but no smaller than EIT. However, the $ECOT$ is equal to T plus minimum service time because if the LP does not receive any more messages, its earliest next output will be in response to processing the message with timestamp T. In (c), $T = \infty$ since there are no unprocessed messages. Therefore, $ECOT$ is equal to ∞.

Various conservative protocols differ in how they compute the value of EIT for each LP in the system. By definition, in a conservative protocol, at real time r, p can only process messages with timestamp $\leq EIT_p(r)$. Therefore, efficiency of a conservative protocol depends on how efficiently and accurately each LP can compute the value of its EIT. In the following sections, we discuss how the three protocols that we consider compute EIT.

2.1 Null Message Algorithm

An LP uses *null* messages [10] to inform every LP in its *dest_set* of the value of its EOT whenever it changes. The EIT of a process is simply equal to the *minimum of the last EOTs received* from all the LPs in its *source-set*. Null message overhead can be reduced by piggybacking null messages with regular messages whenever possible. The frequency of sending null messages can also be a tunable parameter of the protocol. A positive timestamp increment between the input and the output messages is required [21] for at least one LP in every cycle of LPs to ensure that the simulation model does not deadlock.

2.2 Conditional-Event Algorithm

In the conditional event protocol [11], the LPs alternate between an *EIT computation phase*, and an *event processing* phase. For the sake of simplicity, we first consider a synchronous version which ensures that there are no messages in transit when the LPs reach the *EIT computation phase*. In such a state, the value of EIT for an LP p is equal to the minimum of ECOT over all the LPs in the transitive closure[1] of $\textit{source-set}_p$.

The protocol can be made asynchronous by defining the EIT to be the minimum of the values of ECOT of all the LPs in the transitive closure of the source-set and the timestamps of all the messages in transit from these LPs. Note that this definition of EIT is the same as the definition of Global Virtual Time (GVT) in optimistic algorithms. Hence, any of the GVT computation algorithms [8] can be used. Details of one such algorithm are described in section 5.1.5.

2.3 A New Algorithm

Assuming that messages with same timestamp may be processed in an arbitrary order, the conditional-event algorithm does not *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 an asynchronous conditional event algorithm which allows the null message protocol to perform unhindered. The EIT for any process is 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 conditional event algorithm to execute even without lookahead – a scenario in which null message algorithm alone will deadlock.

3 Maisie

Maisie [4] is a C-based distributed simulation language. A Maisie program is a collection of entity definitions and C functions. An entity definition (or an entity type) describes a class of objects. An entity instance, henceforth referred to simply as an entity, represents a specific LP in the model;

[1] Transitive closure of source-set in many applications is almost the same as the set of *all* LPs in the system

```
entity server { mean }
    int mean;
{   message high { ename hiid; } ;
    message low { ename loid; } ;
    ename loid;
    int rem_time=MAXINT, dep_time, busy=0;
    for(;;)
    {   wait rem_time for
        {   mtype(high)
            { if(busy)
                    rem_time=dep_time − sclock();
              hold(expon(mean));
              invoke msg.high.hiid with done;
              if(busy)
                    dep_time=rem_time + sclock(); }
            or mtype(low) st(!busy)
            { busy=1; loid=msg.low.loid;
              rem_time=expon(mean);
              dep_time=sclock() + rem_time; }
            or mtype(timeout)
            { busy=0; rem_time=MAXINT;
              invoke loid with done; }
        }
    }
}
```

Figure 2: Maisie Model of Priority Server

entities may be created dynamically and recursively. Entities communicate with each other using buffered message passing. Every entity has a unique message buffer; asynchronous send and receive primitives are provided to, respectively, deposit and remove messages from the buffer.

Specific simulation constructs provided by Maisie are similar to those provided by other process-oriented simulation languages. However, Maisie extends these constructs to allow the enabling condition for an event to be described succinctly. Innovative features of Maisie include the ability of an entity to inspect specific messages from its message buffer and the use of compound resume conditions. These constructs allow an entity to remove a message from its buffer only when the entity is ready to process the message. Appropriate use of the wait statement leads to succinct programs and reduces program development time.

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 2 describes the Maisie model of the server.

The *server* entity defines two types of messages, *high* and *low* to represent the two types of requests that may be received by it. Upon servicing the request, the server sends a *done* signal back to the entity that requested the service (whose identity is included in the request message).

Henceforth, we will use *high* to mean a message of type *high*; similarly for *low*. The body of the entity consists of an infinite number of executions of the **wait** statement (line 9) that is used to receive jobs and to schedule future events. When idle, the entity accepts the next message from the buffer. If the message is of type *high*(line 10), the entity executes a hold statement to schedule an unconditional timeout message for the future time at which the service of the request will be completed. If the entity services a *low* message (line 17), it schedules a conditional **timeout** message; this message will be automatically cancelled if the entity receives a *high* message in the interim. In this case, the *high* message is again processed by executing a hold statement (line 13), after which service of the *low* message is resumed (line 9).

4 Optimizations

Two factors which most affect the performance of null message based conservative algorithms are *lookahead* and the *knowledge of the communication topology*. In this section, we discuss the Maisie constructs to support these optimizations.

4.1 Lookahead

Lookahead of an LP, at a given real time, is defined to be the difference between its EOT and EIT. Clearly, in the null message protocol, the greater the lookahead for an LP, the farther successor's simulation clock can advance. For a given EIT, the value of the EOT (and lookahead) depends on LP semantics , the message timestamp distribution, and the messages execution speed. In this section, we discuss how semantic information from a Maisie entity is used to compute better estimates of its EOT.

Every Maisie entity has a *Clock* variable associated with it which contains the timestamp of the *latest* message processed by the entity. Any message generated by an entity contains a timestamp greater than or equal to the value of the *Clock* for the entity when the message is generated. Therefore, an obvious estimate of EOT is equal to the current value of the *Clock* variable.

4.1.1 Transparent extraction of Lookahead

The $hold(t_c)$ 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. Consequently, in applications which frequently use *hold* statement (for example, the code for an eager FCFS server[15]), the value of *Clock* can progress far beyond the value of EIT, thereby improving the lookahead.

4.1.2 User specified Lookahead

If the user is able to guarantee that the minimum timestamp increment 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 the 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 needed by the underlying system. In the case of an FCFS server, the expression could simply be *ntime*, 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 (line 8). If the server is not processing any job, the user specified lookahead is equal to the minimum of the next two service times (instead of just the next service time) because it is possible that the next job processed is a low priority job that is immediately pre-empted by a high priority job. If the server is currently processing a job, the user specified lookahead is equal to the minimum of the next service time and the remaining service time of the current job (given by dep_time – **sclock()**).

4.2 Communication Topology

Any conservative method that uses null messages can benefit from maintaining precise knowledge of the communication topology. In absence of this knowledge, the null messages would have to be broadcast to every entity in the system, which would severely degrade performance. As the communication pattern typically changes during the execution of a model, maintaining a static communication topology implies that each LP, at any given time, might be synchronizing (using null messages) with a larger number of LPs than is necessary. Allowing dynamic process and channel creation(and destruction), therefore, can improve the performance considerably [19]. However, most existing conservative simulation systems assume a static topology.

The two problems in allowing arbitrarily dynamic channel creation are as follows:

- If the first message that an LP b sends to another LP c is timestamped t, and if c does not add b to its *source-set* before or at simulation time t, then there is a possibility that c may receive the message when its clock is past simulation time t – clearly a causality violation.

- If c adds b to its *source-set* at simulation time $t' \leq t$, but b does not add c to its *dest-set* until *after* t', it is possible that simulation may deadlock because c may be waiting for a null message from b which b may never send.

```
#define MIN(a,b)    ((a < b)?a : b)
entity server { mean }
    int mean;
{   message high { ename hiid; } ;
    message low { ename loid; } ;
    ename loid;
    int rem_time=MAXINT, dep_time,
        next_time, next_next_time, busy=0;
    lookahead(busy ? MIN(next_time,dep_time
    −sclock()) : MIN(next_time,next_next_time));
    next_time=expon(mean);
    next_next_time=expon(mean);
    for(;;)
    {   wait rem_time for
        {   mtype(high)
            { if(busy)
              {   rem_time=dep_time − sclock();
                  dep_time=dep_time + next_time; }
              hold(next_time);
              next_time=next_next_time;
              next_next_time=expon(mean);
              invoke msg.high.hiid with done; }
            or mtype(low) st(!busy)
            { busy=1; loid=msg.low.loid;
              rem_time=next_time;
              next_time=next_next_time;
              next_next_time=expon(mean);
              dep_time=sclock() + rem_time; }
            or mtype(timeout)
            { busy=0; rem_time=MAXINT;
              invoke loid with done; }
        }
    }
}
```

Figure 3: Maisie code for Pre-emptible Priority server with user defined lookahead

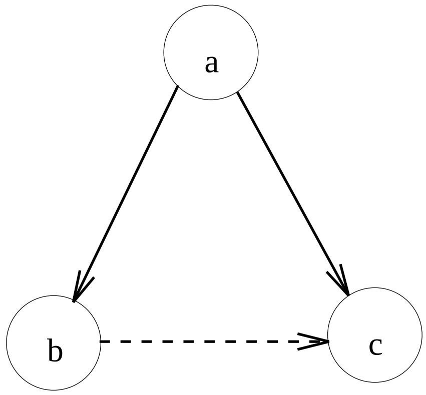

Figure 4: Creating channels dynamically

Maisie provides following constructs that can be executed inside any entity: *add_source*(e), *add_dest*(e), *del_source*(e), and *del_dest*(e). Using these constructs, an entity can dynamically maintain its *source-set* and *dest-set* as long as it is *safe* to do so. The sufficient conditions for safe channel creation are given below. Similar conditions exist for safe channel destruction.

1. If the first message c receives from b is at simulation time t, c should add b to its *source-set* before or at simulation time t.

2. If c adds b to its *source-set* at simulation time t', b should add c to its *dest-set* before or at t'.

In many applications, safe creation (and destruction) of channels is easy. For example, at the time of dynamic entity creation, establishing a channel between the creator and the created entity is safe. Figure 4 shows another situation that occurs in many applications where channel creation is safe: entities b and c are created by an entity a at the same simulation time, and a channel needs to be established between them at the time of creation. In section 6.3, we describe a simulation example where the maintenance of a dynamic topology can lead to better performance than a static topology.

5 Implementation

In this section, we describe the design and implementation of the conservative runtime system for Maisie. Figure 5 describes the system architecture of the conservative implementation. The Maisie simulation program is translated using the Maisie compiler[18] into an intermediate code. The intermediate code consists of C statements and calls to the Simulation Virtual Machine (SVM). SVM is a simulation algorithm-independent library that provides the following functionality:

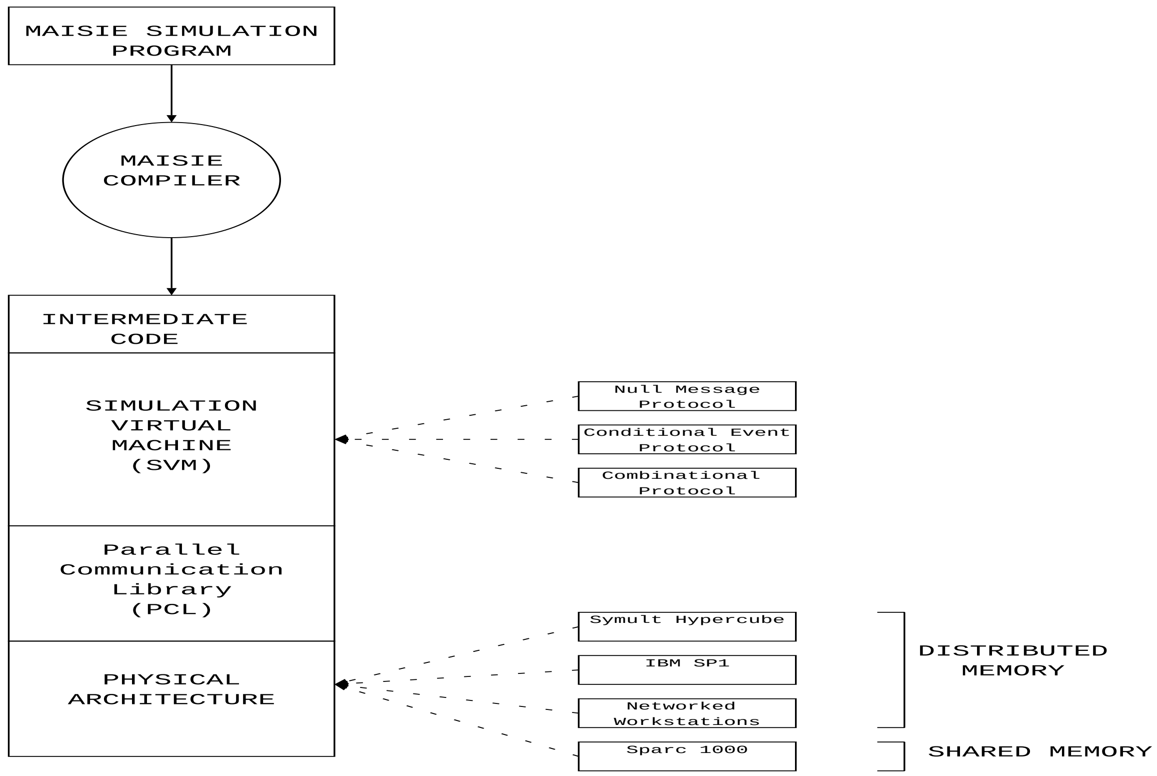

Figure 5: System architecture of the conservative runtime implementations

- Creation and termination of LPs.
- Scheduling of LPs (if more than one LP are assigned to the same processor).
- Message communication between LPs.
- Processing of messages by the LPs in the correct timestamp order.

SVM has been implemented using many parallel simulation protocols. Its implementation using global event list and optimistic protocols is described in [18]. In this section, we describe the implementation using 3 conservative protocols (null message, conditional event, and a combination of the two).

The parallel implementations of SVM use a message passing library called PCL (Parallel Communication Library) whose functionality, for historical reasons, is very similar to that of cosmic environment[25]. Currently, PCL runs on both, distributed memory parallel architectures (Symult, Network of Workstations, and IBM SP), and a shared memory architecture (SPARC 1000) [3].

5.1 Conservative Runtime System

In this section, we describe some of the main issues involved in the design of conservative runtime system for a parallel simulation language, compare the various alternatives available, and describe the choices we have made.

5.1.1 Memory management

The runtime system must manage dynamic memory used for entity creation and message communications. Relying on OS level constructs to do this can be expensive [18]. Therefore, the runtime system does its own management of the memory used for entity states and in messages communication. The latter includes user messages, and protocol messages (e.g. null messages, conditional event messages etc). Since different free lists are maintained for each message type, and each type has a fixed size, there is no overhead of memory fragmentation typically associated with memory recycling.

5.1.2 Message communication frequency

Message sending frequency: For a message intended for an entity on a remote processor, the runtime can either send the message as soon as it is generated, or aggregate the message with subsequent messages intended for the same processor, then send them as a bigger packet.

The aggregation has the advantage of reducing the number of actual messages in the system, thus reducing the queue management overheads, but it can increase the blocking (idle) time of the receiver. In our runtime, we choose to send the messages as soon as they are generated.

Message receiving frequency: Local messages, *i.e.* messages sent to an entity mapped to the same processor are deposited into the receiver's message queue as soon as they are generated. However, the runtime system must periodically check for any remote messages. The frequency of doing this can be a tunable parameter of the system.

If the system polls the remote message buffer frequently, it may significantly increase the message processing overhead as the polling typically requires an (expensive) OS function call. On the other hand, the runtime system may poll for remote messages only when there is a local deadlock. We choose the latter alternative in our runtime system. Although this reduces the communication overhead, it has the potential of causing the system buffers to overflow. The MPL communication library on IBM SP allows us to implement flow control by preventing the processors from sending any more messages to a processor whose system buffer is full.

5.1.3 Scheduling

If multiple LPs are mapped to a processor, a scheduling strategy must be implemented to choose one among a number of LPs that may be eligible for execution at a given time. An LP is eligible to be executed if it has an unprocessed message with timestamp $\leq$ current value of EIT for the LP. In our runtime, once an LP is scheduled, it is allowed to execute as long as it remains eligible. This helps reduce the overall context switching overhead since more messages are executed each time the LP is scheduled, but it may result in the *starvation* of other LPs on the processor. This, in turn, may cause blocking of other processors that might be waiting for messages from these LPs, resulting in sub-optimal performance. Note, however, that without prior knowledge of the exact event dependencies, no scheduling scheme can be optimal.

An alternative scheduling strategy, used in the Global Event List protocol, is to schedule events across all LPs mapped to a processor in the order of their timestamps. In Figure 7, we compare the performance of the two scheduling schemes. The table shows the average number of messages processed per context switch using the two scheduling schemes for a simulation of Closed Queueing Network of FCFS servers (CQNF) running on one processor (see section 6 for a more detailed description of the CQNF benchmark). The particular example used here consists of 16 routers and 20 FCFS servers per router. All the simulations were carried out for the same simulation interval, but with different number of initial jobs in the system. As seen in the table, for high job density, our scheduling scheme executes far more messages per context switch than the GEL scheduling scheme. This can result in a substantial reduction in the total context switching overhead. As seen from Figure 6, for a high job density (96 jobs/router), the uni-processor NM protocol is 1.63 times faster than the GEL protocol. We believe that the main cause for the reduction in execution time is the lower context switching overhead (the other factor being the smaller queue management overhead due to smaller queues, since, in the NM protocol, event queues are maintained at each LP rather than for the whole system).

5.1.4 Message queues

A number of alternatives exist to implement message queue of an entity. Some simulators use a single global queue to store messages for all LPs mapped to a processor. At the other extreme, it is possible for each LP to have a set of message queues, one for each message type that it can receive. Further, the queues may be implemented using a variety of data structures like linear lists, trees, heaps, or priority queues. A detailed study comparing the performance of different data structures for the simulation event queues has been reported in [9].

In our runtime system, a separate message queue is maintained for each entity. A Maisie entity

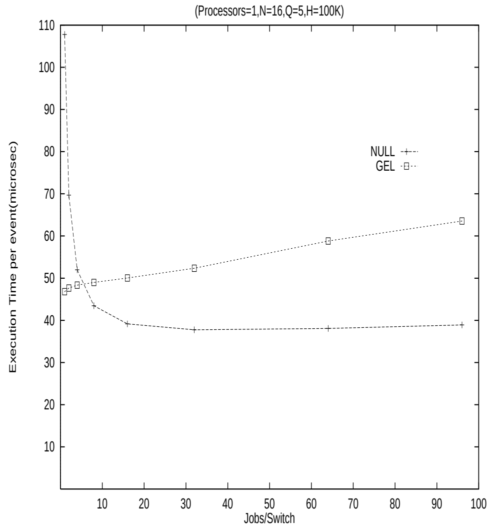

Figure 6: Execution Time per event for GEL and Uniprocesor-NM

Jobs per Router	Total Msgs Processed Per Ent.	Msgs Processed Per Context Switch (Uniprocessor GEL)	Msgs Processed Per Context Switch (Uniprocessor NM)
8	1615	1.16	9.37
18	2667	1.30	16.57
28	3282	1.41	21.43
48	3960	1.54	32.33
88	4533	1.67	52.67

Figure 7: Context Switching Overheads for CQNF

does not necessarily remove messages from its queue in strict order of their timestamps; it may selectively remove messages based on their type and parameters. Therefore, if the average message selectivity, *i.e.* ratio of the total number of message types that an entity can receive to the average number of message types that it receives in a given resume condition is high, it may be possible to reduce the queue management overheads by maintaining independent queues for different message types. However, in most of the applications we considered, the message selectivity was low.

The messages in the queue are ordered by timestamp. We use doubly-linked linear lists to maintain the message queues. Since typically, most of the insertions to the queue take place either near the end of the queue (regular messages) or near the beginning of the queue (timeout messages), we compare the timestamp of the message to be inserted with the timestamps at the two ends of the queues in order to determine at which side to start trying to insert the message. Splay trees, which are used in the sequential implementation of Maisie[18], or other data structures like calendar queues[9], can possibly reduce the queueing overheads further.

The queue insertion overheads for the linear list implementation of the message queue described above were profiled for many applications, and were found to be relatively small. For instance, Table 1 shows the part (percentage) of the event queue traversed at each insertion in our runtime for the CQNF (# switches = 16, # servers/switch = 20) experiment. As seen from the table, even when the message queue is large, on average, only 4 to 7% of the queue is traversed. It is unlikely, therefore, that the use of splay trees will reduce the queue management overhead significantly.

Jobs/Router	Msgs Processed per Ent.	Average Q Length	Percentage of Q Traversed at Each Insertion
8	1615	3.95	6.96%
18	2667	6.80	4.07%
28	3282	9.17	3.67%
48	3960	14.02	3.61%
88	4533	23.0	3.96%

Table 1: Queueing Overhead in NM Protocol for CQNF Experiment

5.1.5 Global Virtual Time (GVT) computation

For the conditional event and the combination protocol, some kind of GVT computation needs to be implemented. As discussed in section 2.3, we want this GVT protocol to be asynchronous so that it does not require the freezing of the computation at each node in order to calculate the GVT.

In our runtime system, GVT computation takes place in phases. At the start of a new phase p, every processor broadcasts the value of *next time*, equal to $min(E_p, M_p)$, where E_p is the smallest ECOT among all the LPs on the processor, and M_p is the smallest timestamp among all the messages sent by the processor in the phase $p-1$. M_p accounts for the messages in transit; in the worst case none of the messages sent by the processor in the phase $p-1$ would have been received and accounted for in E_p of other processors. Messages sent during phases $p-2$ and earlier do not need to be considered because a new phase starts only when all the *next time* messages from the previous phase have been received. This, along with the FIFO communication assumption, implies that E_p of each processor takes into account all the messages sent to it by any processor during phase $p-2$. Thus, during phase p, a processor can compute the GVT to be the minimum of the phase p *next time* values received from each processor.

Note that each *next time* message needs to carry the phase identifier with it. But, since a new phase does not start until the previous one has finished, *next times* from a maximum of two phases can be in circulation at any given time. Therefore, a boolean flag is sufficient to store the phase identifier.

5.1.6 Null message frequency

One of the tunable parameters of any null message scheme is when to send null messages. Different alternatives include eager null messages – an LP sends null messages to all successors as soon as its EOT changes, lazy null messages – an LP sends null messages to all successors only when it is idle, or demand driven – an LP sends null message to a destination only when the destination demands to know the value of its EOT.

We use lazy null message scheme in our experiments. This has the advantage of reducing the number of null messages because several null messages may be replaced by one, the latest, null message. However, the delay in sending null messages may delay the processing of real messages at the receiver's end.

[26] discusses experiments with eager, lazy, demand driven, and some other variants of the null message scheme. For the limited set of experiments they report the results on, demand driven schemes perform poorly, whereas the lazy null message scheme combined with eager event sending, *i.e.* regular events are sent as soon as they are generated (as is the case in our runtime), marginally outperforms other schemes.

6 Experiments and Results

In this section, we describe the results of the experiments performed to evaluate the performance of the three conservative protocols using various types of benchmarks. The experiments reported in this paper were carried out using an implementation of Maisie on the following architectures:

- An older distributed memory MPP, Symult 2010 hypercube, where each node uses a Motorola 68020 CPU and has 4MB of main memory. The reported latency is $\approx 200\mu s$ or more, depending on the length of the message.
- A state of the art distributed memory MPP, IBM SP, where each node uses an RS/6000 processor with 128MB of main memory. The reported latency is $\approx 50\mu s$, depending on the length of the messages.

In addition to the above architectures, Maisie has also been implemented on an SMP machine – a SPARCServer 1000 with 8 SuperSPARC processor, and 128M of shared 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 (GEL) protocol implemented using splay trees) running on one node of the multiprocessor.

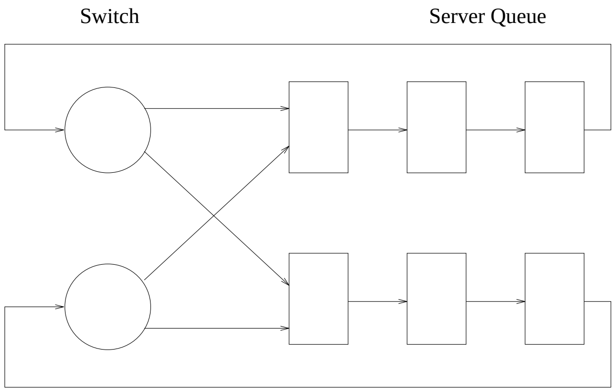

Figure 8: Closed Queueing Network(2 switches and 3 servers per switch)

6.1 Closed Queueing Network Experiments

The Closed Queueing Networks(CQN) used in our experiments consist of N switches. Each switch has a tandem queue of Q servers associated with it. Each switch routes the jobs to the first server in any one of the tandem queues, with 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 sends the jobs back to the unique switch it is associated with. The topology of the network, for N=2 and Q=3, is shown in Figure 8. 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. All the experiments in this section were performed on Symult multi-computer. Each one of the switches and servers in the CQN experiment (Figure 8) is mapped to a different entity instance. The entities corresponding to a switch and all the servers associated with it are mapped onto the same processor.

Figure 9 shows the speedup on the CQNF experiment using 16 processors. As shown, 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 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 for all values of Jobs/Switch.

As seen in the graph, for high job densities, all three protocols show superlinear speedup. There are two reasons for this. Firstly, the conservative protocols have fewer context switches because of better scheduling (as discussed in section 5.1.3). Secondly, since the event queues are maintained on a per entity basis in the parallel protocols, they are smaller, thus resulting in lower queue management overheads.

Figure 12 plots the speedup with respect to the fraction of HIGH priority jobs in the CQNP experiment (same as CQNF 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 cannot be pre-empted. This allows the **transparent** extraction of lookahead, as discussed in section 4.1.1, to take place. Hence, increasing the fraction of HIGH jobs should be expected to improve the performance. This is confirmed by the figure. When all the jobs in the system are HIGH, performance is similar to that of CQNF, since, the priority servers behave like FIFO servers in such a case.

The figure also shows the effect of user defined lookahead, as explained in section 4.1.2, on the performance of simulation protocols. For each protocol, the CQNP experiment was carried out with and without user defined lookahead. As explained before, the *null message algorithm deadlocks in absence of the user defined lookahead.* However, the combination algorithm is able to execute even in absence of a lookahead guarantee, and is able to utilize transparent lookahead where it is available. Figure 12 reveals that for the combination protocol, 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 the conditional event algorithm too. This is because we utilize the user defined lookahead in computing a better estimate of the globally earliest conditional event.

Figure 10 compares the speedups obtained for the same CQNF experiments on Symult and the IBM SP. Because each node of IBM SP is much faster than that of the Symult, a longer simulation interval had to be used for the SP in order to keep the total execution time comparable on the two machines. In addition to the IBM SP being faster in terms of absolute execution times, the figure shows that the speedup figures are also better. This could be because of a more efficient communication subsystem, allowing smaller latencies and better computation – communication overlap. Further, the graph shows that our implementations are stable across substantially different architectures.

We conclude this section with a comparison of the speedups obtained, using 32 nodes, on the same CQNF configuration using the null message protocol and the optimistic space-time implementation described in [2] (Figure 13). As seen in the figure, the optimistic implementation performs slightly better for low job densities, but null message performs better at higher job densities because of lower blocking overhead and a relatively smaller null message overhead.

6.2 Synthetic Benchmark Experiments

In this section, we study the effect of changing several application characteristics on the performance of the null message protocol. The four application characteristics that we study are: lookahead, communication topology, computation granularity, and message population.

We use synthetic benchmarks to carry out these experiments. These benchmarks consist of a ring of N processes with a fixed number of messages circulating between them. In all the experiments reported in this paper, each processes was mapped to a unique processor (*i.e.* the number of entities on any processor, E, is equal to 1). Each process in the network processes the messages it receives in a FCFS order with a shifted-exponential service time, and sends the outputs to any of its C consecutive neighbors to the right with equal probability. Each process starts with J messages. Upon processing each message, the process executes COMP iterations of a *for* loop. COMP can be varied to vary the computation granularity. The values of the parameters N, C, J, and COMP can be changed to study the effect of various factors on performance of the conservative algorithms.

The benchmarks used are homogeneous, *i.e.* all the processes have the same characteristics as far as connectivity, service time, processing time etc. are concerned. Two types of graphs are plotted for each benchmark: Speedup with respect to a GEL protocol using splay trees and NMR (Null Message Ratio – ratio of the number of null messages to the number of regular messages processed in the system). All the experiments reported in this section were performed on the IBM SP.

6.2.1 Lookahead

We study only the effect of improving the lookahead of a system in a homogeneous way, *i.e.* lookahead characteristics of all the processes in a simulation are the same and are varied across different simulations. [17] reports some of the results on lookahead variations 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 processes. Each process is an FCFS server. An FCFS server can be programmed as an eager server, which sends the output as soon as it processes the corresponding input, or as a lazy server, which sends an output with timestamp t only when it is sure that it will not receive any input with a timestamp smaller than t [15]. Also, it may precompute service time to be used as lookahead [22]. To further vary the degrees of lookahead in the synthetic workload, we express, using Maisie lookahead constructs, only a fraction of the precomputed service time, called LAF, as lookahead. Thus, even though the application *knows* the amount of timestamp increment on the next message that it would process, it expresses only a fraction of it as the lookahead. Note that LAF, as defined above, corresponds to the inverse of LAR as defined by Fujimoto[15].

Figure 11 shows how the speedup (on 16 processors) varies with the value of LAF for null message protocol, for lazy and eager servers, respectively. For the case of lazy server, the speedup improves dramatically as we increase LAF [2] from 0.1 to 1.0. The speedup of the eager server is not affected much by an increase in LAF since it has very good transparent lookahead, and precomputing the service time does not help it appreciably. Figure 14 confirms the improvement in efficiency of the lazy server because of an increase in LAF. Note that for LAF=1.0, the NMR for the lazy server and eager server is about the same; however, the speedups are different because of a higher blocking time for the lazy servers.

6.2.2 Communication Topology

We study the effect of changing communication topology by changing the connectivity of 16 processes arranged in a circular fashion. A connectivity of 1 corresponds to a ring topology, with each process sending all its outputs to the following process. A connectivity of i means that each process sends its outputs to the next i processes, distributing them with equal probability. A connectivity of 16, therefore, corresponds to a completely connected network.

All the experiments performed start with 1 job per process. Increased connectivity results in a drop in the speedup as shown in Figure 15. There are two reasons for this:

- An increase in connectivity causes some amount of load imbalance in the system. Consider a connectivity of 1 with 1 job per entity initially. Since all jobs take the same amount of time to be processed, there is always one unprocessed job at every entity. Therefore, assuming a fixed message processing time p and a fixed communication time c, the parallel execution will take $(p + c) * E$ where E is the number of events processed by each LP. If N is the number of processes (= number of processors) in the system, this will result in a speedup of $N/(1 + r)$ in the ideal case, where $r = c/p$. However, when the connectivity is increased to say, 2, there is a "bunching" effect. For instance, at the end of the first iteration, there is a 0.25 probability that an entity will have 2 unprocessed jobs, and a 0.25 probability that it will have no unprocessed jobs. This causes load imbalance.

- For null message protocol, an increase in connectivity means that each process has to synchronize with a higher number of process, thus increasing the blocking time for each message. Also,

[2] 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.

a higher number of null messages have to be sent (because there are more output channels). This results in a decreasing efficiency, and an increase in NMR (as shown in Figure 18).

6.2.3 Message Population

The experiments in this section were carried out with a connectivity of 2. As shown in Figure 16, an increase in message population improves the speedup. This is because it improves the load distribution in the application; if the message density is high, chances are high that each process has some jobs to process at any given time. The improvement levels off eventually when each server has a job in its queue at all times, and other overheads begin to dominate.

6.2.4 Computation Granularity

The experiments in this section were carried out with J=1 (and C=1). This implies that there would be many situations where a processor is idle, waiting for a message to arrive, *i.e.* computation and communication are not perfectly overlapped. The communication latency would be a significant contributor to the total overhead of parallel execution in such a situation. However, this overhead is independent of the computation granularity for each message. Same is true of the overheads due to the processing of null messages. Increasing the computation granularity in such a situation can improve the computation to overhead ratio for the parallel executions, resulting in better speedups. This behavior is confirmed by Figure 17. The speedup reaches a saturation when other factors like blocking due to lack of parallelism become the dominant factors in determining the processor utilization and speedup. Interestingly, Figure 20 shows that the NMR remains more or less unchanged with increased computation granularity This confirms an analytical result in [20], namely, *the number of null messages in a deadlock avoidance simulation is independent of the architecture parameters* (Strictly speaking, Lin's result is only applicable to an eager null message protocol. However, for one entity per processor and low job densities, the number of null messages in the eager and lazy null schemes can be expected to be the similar).

6.3 Dynamic Topology

As mentioned in section 4.2, we allow dynamic creation of processes, and a restricted form of dynamic topology. For simulating dynamically changing networks, this offers a lot of modeling convenience, and possibly better simulation efficiency. In this section, we describe the simulation of a simple queueing network that demonstrates this.

The system, shown in Figure 21, has S serving stations (experiments reported here correspond to $S = 1$) each of which has a certain number of servers. A server may be *up* or *down*. Each

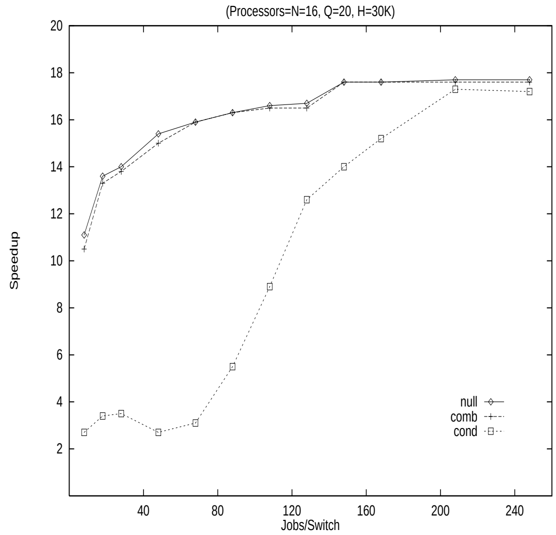

Figure 9: CQNF(Speedup vs. Jobs)

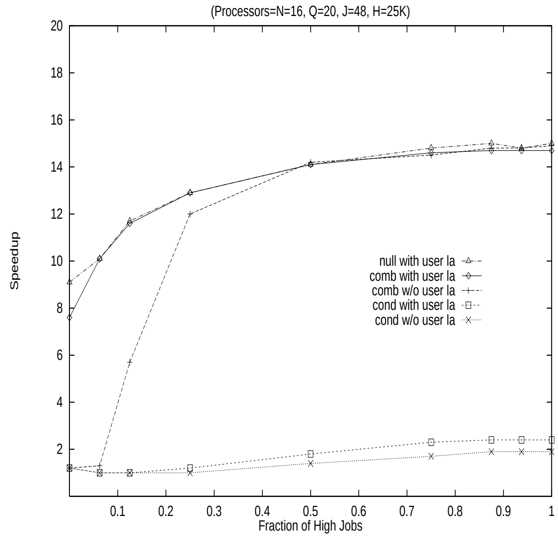

Figure 12: CQNP(Effect of Lookahead)

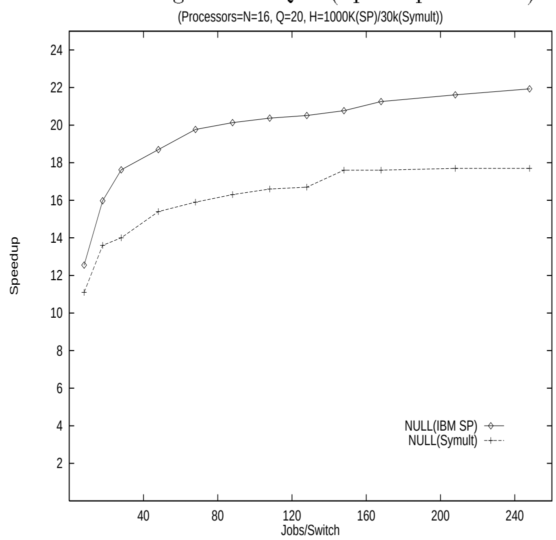

Figure 10: CQNF (Symult and IBM SP)

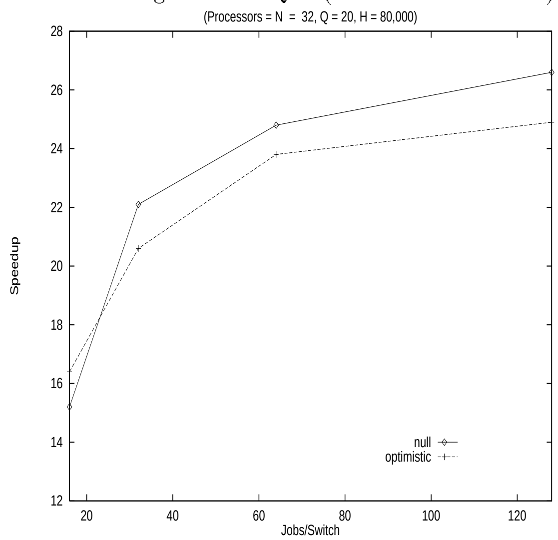

Figure 13: CQNF (Cons. vs. Opt.)

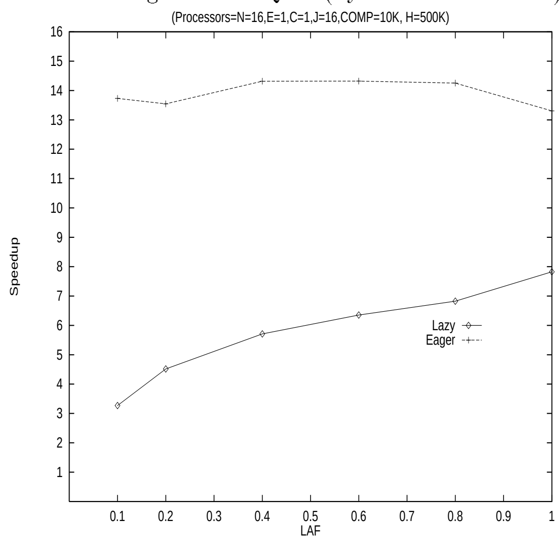

Figure 11: Synthetic(Lookahead):Speedup

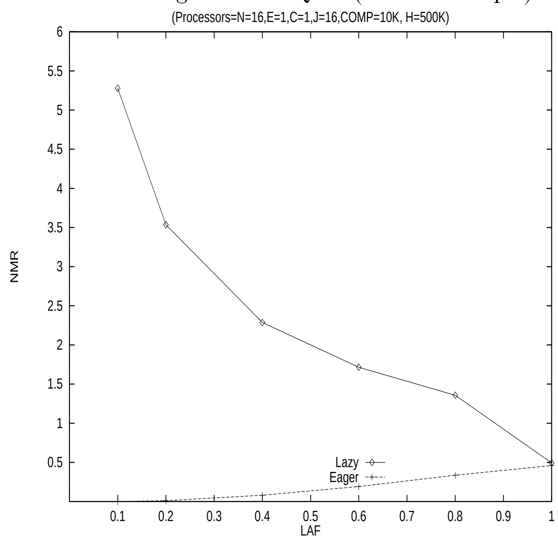

Figure 14: Synthetic(Lookahead):NMR

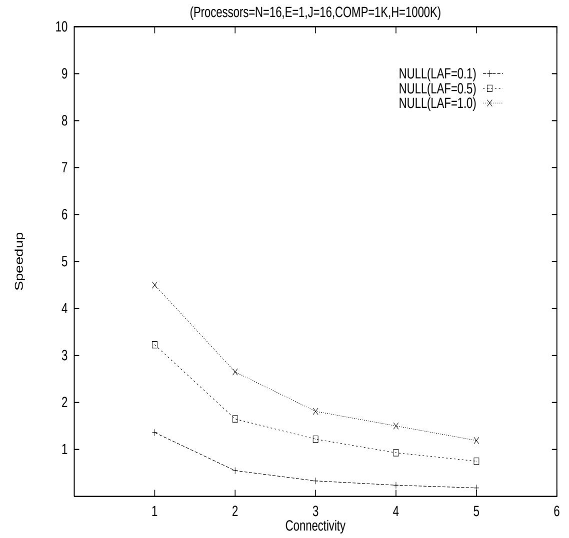

Figure 15: Synthetic(Topology):Speedup

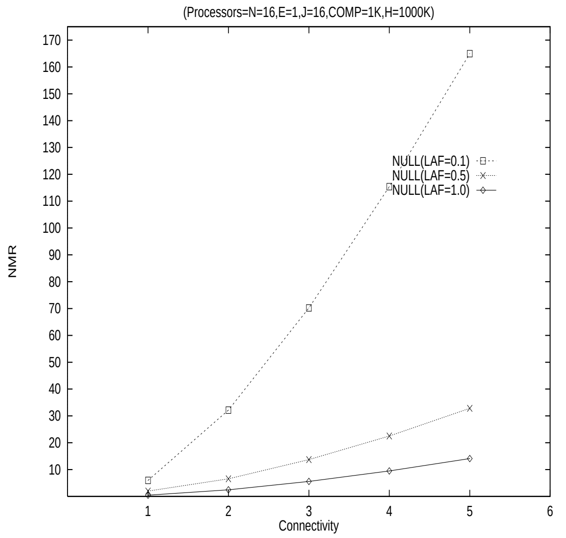

Figure 18: Synthetic(Topology):NMR

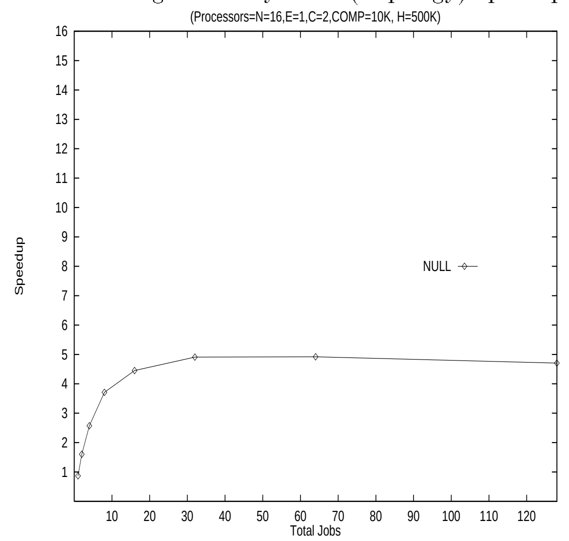

Figure 16: Synthetic(Msg. Pop):Speedup

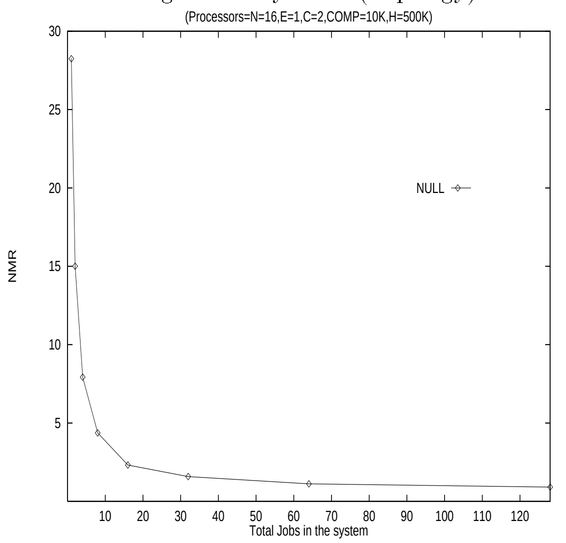

Figure 19: Synthetic(Msg. Pop):NMR

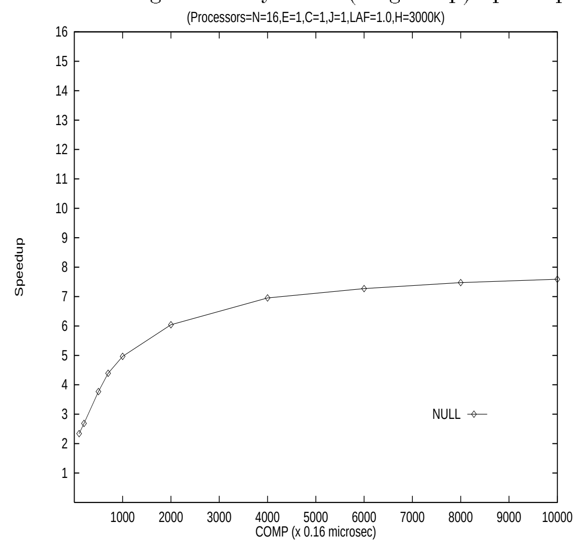

Figure 17: Synthetic(Granularity):Speedup

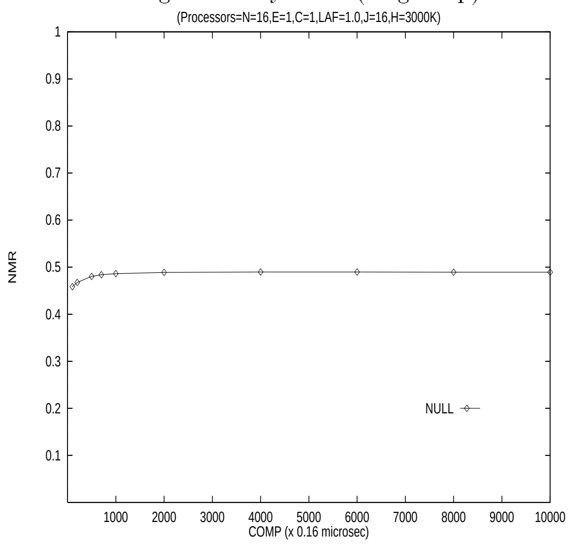

Figure 20: Synthetic(Granularity):NMR

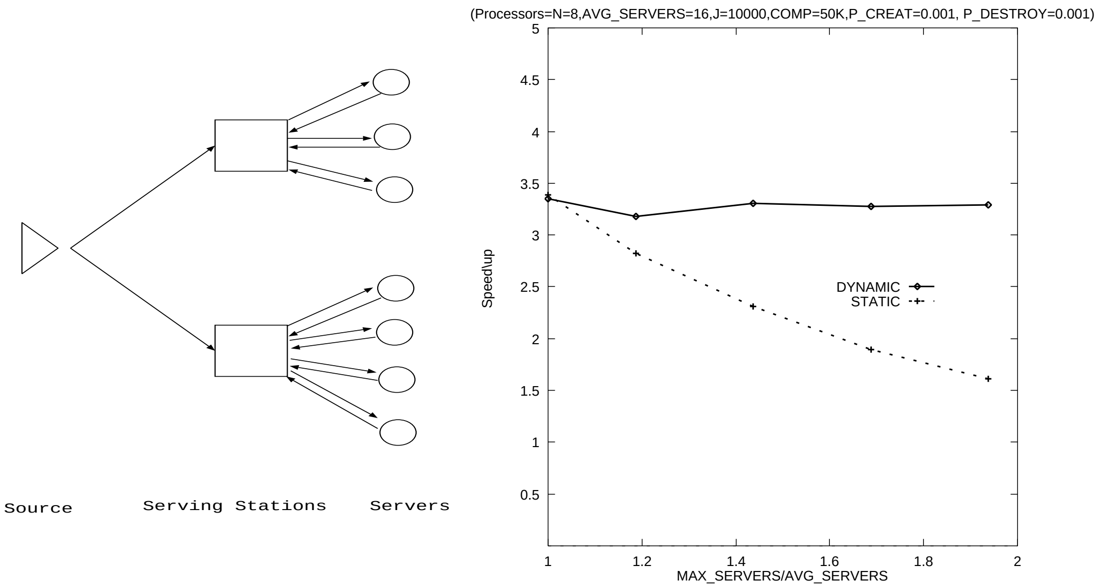

Figure 21: Synthetic Benchmark with Dynamic Topology

server that is *up* may be either *idle* or *busy* serving a job. The source process sends J jobs to each serving station for servicing. Upon receiving a job, the station sends it to one of the idle servers, if one is available, for servicing. Every time the servicing of a job finishes, with some probability, P_DESTROY, one of the idle servers goes *down* (modeling the failure of a server). At the same time, with some probability, P_CREATE, a server that was *down* may come *up* (modeling the repair of a previously broken server). The total number of servers that are *up* at each station is not allowed to go below MIN_SRVR, or above MAX_SRVR. At the start of the simulation, each station starts with AVG_SRVR, equal to (MAX_SRVR+MIN_SRVR)/2, number of servers that are *up*.

We simulate the model described above in the following two ways:

- **Static**: Each station creates MAX_SRVR number of servers at the start of the simulation. The total number of entities in the simulation (and their topology) remains fixed during the entire simulation.

- **Dynamic**: At any time during the simulation, there are only entities corresponding to active servers. Destruction of a server is modeled by actually terminating the corresponding entity. Similarly, creation of a new server is modeled by actually creating a new entity corresponding to it.

Clearly, in the null message based simulation of the static model, each station has to synchronize with even the inactive servers (since the corresponding entities are still alive, and may be re-used later to simulate a newly created server). This results in higher overhead. However in the dynamic model, each station only synchronizes with the currently active servers. Therefore, the performance of the dynamic model would be expected to be better than that of the static model unless the probabilities for creation and destruction are so high that the entity creation and destruction overheads dominate. Figure 21 confirms this. In this figure, we plot the speedup against the ratio MAX_SRVR/AVG_SRVR, while keeping the value of AVG_SRVR fixed. Since in the dynamic model, each station synchronizes with only the active servers, its performance remains relatively unchanged during the experiment. Whereas in the static case, since each station synchronizes with MAX_SRVR number of processes at any given point in simulation, its performance degrades as MAX_SRVR increases.

7 Related Work

Prior work on evaluating performance of discrete-event models with conservative protocols have used analytical and simulation models, as well as experimental studies. Languages/systems that support conservative simulation protocols include Yaddes [23], SIMA [24], and OLPS [1]. 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 do not provide language level constructs to express lookahead and dynamic topology.

Performance of the null message deadlock avoidance algorithm [10] using queueing networks and synthetic benchmarks has been studied by Fujimoto [15]. Chandy and Sherman [11] describe the conditional event algorithm and study its performance using queueing 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.

The effect of lookahead on the performance of conservative protocols was studied by Fujimoto [15]. Nicol [22] introduced the idea of precomputing the service time to improve the lookahead. Cota and Sargent [13] have described the use of graphical representation of a process in automatically computing its lookahead.

8 Conclusion

An important goal of parallel simulation research is to facilitate its use by the discrete-event simulation community. Maisie is a simulation language that separates the simlation model from the specific algorithm (sequential or parallel) that is used to execute the model. Transparent sequential and optimistic implementations of Maisie have been developed and described previously [2]. This paper addresses 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 describes problems with supporting arbitrarily dynamic topologies with conservative methods, and provides language constructs to simulate dynamically changing networks whenever it is safe to do so. Experiments are presented to show the usefulness of allowing this restricted form of dynamism. The paper also describes how certain types of lookahead behavior can be extracted transparently by the simulation system. It 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.

Parameterized synthetic benchmarks were used to evaluate the effect of varying four application characteristics: lookahead, computational granularity, message population, and communication topology. User defined lookahead was found to substantially improve the performance when the inherent (transparent) lookahead in the system was low (lazy FCFS server) but, it had only marginal effect when the inherent lookahead was high (eager FCFS). Increasing the computational granularity improved the performance until it reached saturation. We identified the minimal computation granularity needed in an application under a given topology to obtain performance improvements with parallel execution. Increasing the message population in the system also improved the performance significantly until it reached a saturation point. A higher average connectivity was found to degrade the performance.

Future work includes performance evaluation using more applications from different discplines.

References

[1] M. Abrams. The object library for parallel simulation (OLPS). In *Proceedings of the 1988*

Winter Simulation Conference, pages 210–219, San Diego, California, December 1988.

[2] R. Bagrodia, K.M. Chandy, and W. Liao. A unifying framework for distributed simulations. *ACM Trans on Modeling and Computer Simulation*, October 1991.

[3] R. Bagrodia, Y. Chen, V. Jha, and N. Sonpar. Parallel gate-level circuit simulation on shared memory architectures. In *Parallel and Distributed Simulation*, 1995.

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

[5] R. Bagrodia, Gerla M., Chen Y., Palnati P., Martin J., Jha V., and Walton S. Scalable simulation of a high speed wormhole network in a parallel language. In preparation.

[6] R.L. Bagrodia. Language support for parallel discrete-event simulations. In *Winter Simulation Conference*, December 1994.

[7] R.L. Bagrodia, Zheng Li, Vikas Jha, Yu-an Chen, and Jason Cong. Parallel logic-level simulation of vlsi circuits. In *1994 Winter Simulation Conference*, Orlando, FL, December 1994.

[8] S. Bellenot. Global virtual time algorithms. In *Multiconference on Distributed Simulation*, January 1990.

[9] R. Brown. Calendar queues: A fast 0(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10), October 1988.

[10] 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.

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

[12] Y. Chen, Bagrodia R., and Jha V. Parallel switch-level vlsi simulation. Technical report, Computer Science Dept., UCLA, June 1995.

[13] B.A. Cota and R.G. Sargent. A framework for automatic lookahead computation in conservative distributed simulation. In *Multiconference on Distributed Simulation*, January 1990.

[14] R. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[15] R. M. Fujimoto. Performance measurements of distributed simulation strategies. Technical Report Tech. Rep. UUCS-87-026a, University of Utah, Salt Lake City, 1987.

[16] R.M. Fujimoto and D.M. Nicol. State of the art in parallel simulation. In *Winter Simulation Conference*, December 1992.

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

[18] W. Liao. *Maisie, A Language Supporting Efficient Distributed Discrete-Event Simulations.* PhD thesis, UCLA, 1993.

[19] J.M. Lin. Exploiting dynamic topological information to speed up concurrent multicomputer simulation. In *Parallel and Distributed Simulation*, January 1992.

[20] Yi-Bing Lin. *Understanding the Limits of Optimistic and Conservative Parallel Simulation.* PhD thesis, University of Washington, Seattle, August 1990.

[21] J. Misra. Distributed discrete-event simulation. *Computing Surveys*, 18(1), March 1986.

[22] D.M. Nicol. Parallel discrete event simulation of fcfs stochastic queueing networks. In *Parallel Programming: Experience with Applications, Languages and Systems*, pages 124–137. ACM SIGPLAN, July 1988.

[23] Bruno R. Preiss. The Yaddes distributed discrete event simulation specification language and execution environments. In *Proceedings of 1989 SCS Multiconference on Distributed Simulation*, pages 139–144, Tampa, Florida, March 1989.

[24] H. Rajaei and R. Ayani. Language support for parallel simulation. In *Parallel and Distributed Simulation*, January 1992.

[25] C.L. Seitz, J. Seizovic, and W. Su. The c programmer's abbreviated guide to multicomputer programming. Technical Report Caltech-CS-TR-88-1, Caltech, 1988.

[26] Wen-king Su and C.L. Seitz. Variants of the chandy-misra-bryant distributed simulation algorithm. In *1989 Simulation Multiconference: Distributed Simulation*, Miami, Florida, March 1989.

List of Footnotes

- Affiliation of Authors: Rajive Bagrodia and Vikas Jha are with Computer Science Department, University of California, Los Angeles, CA 90024. Email: <rajive/jha@cs.ucla.edu>.

- This research was partially supported by the U.S. Department of Justice/Federal Bureau of Investigation/Advanced Research Projects Agency, ARPA/CSTO, under contract J-FBI-93-112 and the U.S. Department of Defense/Advanced Research Projects Agency ARPA/CSTO, under contract DABT-63-94-C-0080. We also acknowledge the support of Caltech for providing access to Symult, and that of UCLA Office of Academic Computing and Argonne National Lab for providing access to IBM SP.

1. Transitive closure of source-set in many applications is almost the same as the set of *all* LPs in the system.

2. 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.

List of Figures

List of Tables