IDES: A Java-based Distributed Simulation Engine

David M. Nicol
Department of Computer Science
Dartmouth College

Michael M. Johnson
Ann S. Yoshimura
Michael E. Goldsby
Sandia National Laboratories

Abstract

This paper describes the design and performance of IDES, a Java-based distributed simulation engine being developed at Sandia National Laboratories. The feasability of using Java is demonstrated by achieving order of magnitude speedup gains, on a model with three quarters of a million simulated entities, on a "off-the-shelf" system of 56 PentiumPro processors.

1. Introduction

The *Infrastructure for Distributed Enterprise Simulation* (IDES), is a distributed simulation engine under development at the Sandia National Laboratories. As one of IDES's principle goals was portability and use in heterogeneous computing environments, we have implemented IDES in Java. This paper describes IDES's organization, and reports on overhead costs of executing simulations under IDES, on a large-scale computing system built of clusters of PentiumPro processors. While overall processing rates are fully in accordance with Java's reputation for slowness, we find that by increasing the number of processors (and so also increasing the available memory size) we effectively increase the size of model that can be simulated, and increase the aggregate rate at which the simulation's work is performed. IDES is a general simulation framework, suitable for simulating computer systems and communication networks. While these and the motivating IDES application area are of interest, the principle contribution of this paper is in demonstration of a distributed simulator built in Java, run on a reasonably large configuration, and assessment of the strengths and limitations of distributed simulation using Java. Such an assessment is quite timely, as the general systems area of distributed simulation is of growing practical importance, and Java is the most visible language today for doing distributed programming.

IDES targets a variety of potential simulation problems; its general structure admits to discrete-event simulations that can be described in terms of entities that communicate with each other through message-passing. An entity is "woken up" for processing as a result of either having scheduled itself to wake up at that time, or having received a message. When an entity "wakes up" it performs some computation, possibly generating one or more messages to itself or other entities as a result—and schedules (as a function of its present state) the time at which it will next wake up.

While sparse, this paradigm is general enough to support construction of richer computational models, such as process orientation. Even without that addition it is rich enough to model many computer and communication systems. However, one class of problems of particular interest to Sandia has strongly influenced our approach to synchronization. In that class, the calculation an entity performs to determine its next wakeup time is the numerical integration of a differential equation. An example of such in communications modeling is when one uses a Fluid Stochastic Petri Net (FSPN) to model the network [6]. Differential models describe fluid flow; in an IDES implementation of FSPN one might integrate to determine when, in the absence of further changes in the inflow behavior, the fluid level at a fluid place reaches a critical threshold; that time becomes its next wakeup time. It may happen that after scheduling its next wakeup time at w, an entity receives a message to be processed at time $s < w$, and as a result reschedules its next wakeup time at s. The fact that such a rescheduling depends on the solution of a differential equation implies that it is very difficult in general to predict a lower bound on when next an entity will wake up. This means that such models have little "lookahead", a facet that affects synchronization. Section 2 discusses this in more detail.

The emergence of Java as the lingua franca of network computing, plus the existing and growing importance of simulation in all facets of complex system design has led a number of groups to look at issues of Java-based simulation. **simJava**, developed at the University of Edinburgh was among the first publically released simulators written in Java. It provides a process-oriented modeling environment, complete with animation classes. However, **simJava** is unsuitable for our purposes as it neither supports distributed simulation, and its construction makes it performance-challenged. However, Page, Moose and Griffin have added functional distribution to **simJava** using Java's Remote Method Invocation (RMI) methods [10], although no performance or capability assessments are offered. Java-based distributed simulation using conservative synchronization has been reported by Ferscha and Richter in [2]. That project also explores the use of Java's RMI features in a simulation context. Capabilities in either problem size or processing speed are not assessed. Java is sparking interest in application to simulation in purely serial contexts as well [7, 5].

2. Distributed Simulation

A distributed simulator like IDES is comprised of a collection of processors, each of which is assigned a group of entities to simulate. The problem of maintaining temporal fidelity between processors is a well studied problem in the Parallel and Distributed Simulation (PADS) literature, e.g. ([3, 8]). So-called "conservative" techniques work by prohibiting a processor from doing any simulation work at time w before the processor is certain that none of its entities will be affected by a message from an entity in another processor, at a simulation time smaller than w. "Optimistic" techniques use speculative computing, check-pointing state to support rolling back to an earlier simulation time when a message arrives in an entity's past.

It has been well established that to get good performance using conservative methods it is necessary for the simulation model to provide good *lookahead*. In the context of IDES-type models, lookahead means an ability to predict a lower bound on the simulation time when next an entity may wake up and send a message to another entity. But, as we have already seen, when wakeup times are computed by numerical solution of differential equations, lookahead in general is very hard to find. For this reason we look towards an optimistic method.

A bevy of optimistic synchronization protocols exist, all complex and all challenging to program correctly and efficiently. The models motivating IDES are characterized by extremely large data states associated with some entities. We are very concerned about memory consumption, since the state-saving associated with optimistic processing, can, if left uncontrolled, run amok. We have therefore adapted a protocol that (with modifications we describe) provides a great deal of control while eliminating anti-messages.

Breathing Time Buckets (BTB) is in a class of synchronous protocols that engage in global synchronization periodically. It revolves around the notion of an "event horizon", which we define as follows[1]. Supposing that entity states are globally synchronized at time t, a processor executes events (in the IDES context, wakes up entities) in monotone non-decreasing time-stamp order. After processing the n^{th} event after the synchronization point, let t_n be the time of the smallest known future event on the processor. Now as the processor executed events, it generated messages, some of which are destined for other processors. We should note here that a message generated by an event processed at time s has a *send time* of s. However, the time at which the message affects the recipient, called the *receive time*, may be larger. The gap between send-time and receive-time reflects things like transmission time over a communication channel, or the time lapse between when the job enters service and when it leaves service. This gap is pervasive and essential to achieving high performance in parallel distributed simulation. Returning to the notion of event horizon, we let r_n be the least receive time among all off-processor messages generated by the n events executed since the last synchronization point. A processor's *local event horizon* (LEH) occurs at the earliest time r_n such that $r_n \leq t_n$. The system's *global event horizon* (GEH) is the minimum processor LEH. Now, if processors last synchronized at time t, and $g(t)$ is the GEH following that, the interval of simulation time $[t, g(t))$ has the useful property that no inter-processor message whose send-time is in $[t, g(t))$ can also have a receive-time in $[t, g(t))$. Therefore, provided that at time t a processor already has all messages targeted to its entities with receive times in $[t, g(t))$, all processors can correctly simulate forward from t to $g(t)$ without receiving messages from entities on other processors. The problem is that at time t the value of $g(t)$ is not known—no processor knows when to stop! By its very definition, $g(t)$ must be computed. This is where optimism comes in. Conceptually a processor may execute forward from time t until it discovers its local event horizon. As it executes, it saves state. Reaching its local event horizon, it then coordinates with other processors to compute $g(t)$, and then rolls back to its state at time $g(t)$. All off-processor messages generated by executing events in $[t, g(t))$ are known then to be correct, and can be incorporated into the simulation's state for evaluation in future synchronization windows.

There are a number of papers concerning BTB and modifications to its implementation [15, 13, 14]. IDES has made additional ones. First, the original notion of event horizon (and all the literature describing it) is in terms of *entity* local event horizons; from our perspective it is as though every entity executes on its own processor. As a consequence, IDES global event horizons are larger, so more simulation work is accomplished per synchronization window. Another contribution made by the IDES project was in describ-

[1] All the literature on BTB defines the event horizon more restrictively. We comment on this later.

ing the "pre-emptive min-reduction" [9], yet another in noting and studying when it makes sense to alter the BTB message passing scheme. Rather than withhold all messages at the sender until $g(t)$ is known, and then distribute ones with send-times less than $g(t)$, one can send messages as generated, and then have the receiver filter out ones with send-times as large as $g(t)$.

The problem of reducing state-saving costs is another widely studied area in PADS (see the bibliography of [4]). At present IDES uses so-called "copy saving", which means that prior to executing a wakeup at time t the variable portion of an entity's state is checkpointed in its entirety. In the future IDES will use more sophisticated "incremental" state-saving techniques. However, the modification we describe here for memory usage control should be adaptable to any checkpointing scheme.

Optimism uses memory by saving information needed to accomplish recovery. If after time t processors synchronize at a time $h(t) \leq g(t)$, then the same properties hold as before—no event executed in $[t, h(t))$ depends on the execution of some event on some other processor which was not known at time t. Under BTB, memory used to save state before time $h(t)$ can be made available for reuse once $h(t)$ is known. The reclaimed memory is used then to save state in the next synchronization window.

Memory control is an important issue in optimistic simulation. The best known parallel simulators, GTW [1] and SPEEDES [15] both control memory by initiating garbage collection periodically; essentially a processor garbage collects (also known as computing the *global virtual time*, or GVT) after every n events processed, where n is a configuration parameter. In our context it is likewise straightforward to use available memory to control the size of the synchronization window, bringing the window to a close when there is a threat of exhausting available space. We accomplish this by having each processor track the number of bytes of state memory it has saved. Prior to executing an event, the amount of used memory is compared against the threshold. When the threshold is exceeded, the processor offers its time-of-next-event to the window reduction, just as though it were its own LEH. The time-stamp offered is clearly no larger than the processor's LEH, so that the ultimate window edge computed, $h(t)$, is no larger than $g(t)$. This scheme is similar in concept to the GTW and SPEEDES memory control methods, as the BTB window calculation is the moral equivalent of GVT. In the remainder of the paper we will speak of the end-of-window time, $h(t)$, with the understanding that it is the time computed at which the processors re-synchronize.

One attraction of Java over C++ is its relatively simple and direct syntax for expressing exception handling. We could in principle recover from exhausted memory by letting Java throw an "out-of-memory" exception, and use this to trigger the window calculation. While appealing in concept, the approach has implementation and performance problems. First, by the time Java exhausts memory, it is very likely thrashing the swap disk (we confess to have discovered this the usual way, empirically). We want to reclaim the memory before we start shaking the disk. Second, at the point Java realizes the memory is gone, we will need to immediately reclaim some memory that is needed for the end-of-window calculation, such as message buffers. So while the proposed scheme requires identification of a suitable threshold, in practice it is much easier to implement than an exception-based scheme.

3. Implementation Details

IDES differs from other BTB-based simulators both by being implemented in Java, and in the particulars of how it manages to reconstruct the simulator state when $h(t)$ is known. As these details are pertinent to anyone who would use Java to build a simulator or to anyone who would use the BTB protocol to synchronize a distributed simulation, we will sketch some of those details.

One of the attractions of Java is that threads are built directly into the language. It is tempting to use threads liberally in the simulation model itself, e.g., in a process-oriented simulator map a process directly to a thread, but in our experience threads should be used cautiously. There is a much greater potential for synchronization errors and race conditions using threads. However, threading is precisely what is needed to deal with asynchronous communication. IDES establishes a two-way socket connection between every pair of processors (and we note in passing that Java makes socket creation and management blessedly simple). A processor throws a thread for each such socket to listen for and deal with incoming messages. A processor also throws a thread to deal with scheduling and executing events. Thus, if there are P processors used in the system, each processor throws $P + 1$ threads. Potential synchronization problems (and hence mechanisms to protect from those problems) are then limited to entities that are shared between message handling code and simulation workload code.

IDES does not buffer and "bundle" messages destined for a common destination. This is a common enough performance optimization used throughout high-performance computing; we are interested in determining whether there are any benefits in our context, but have not yet done so.

Another performance-sensitive area of our implementation proved to be loops over all entities on a processor, and associative access to those entities. Java provides *Vectors* and *Hashtables*, each with a rich assortment of methods for using them. In our performance tuning we discovered that these methods are quite expensive. Since our pool of entities is static, we could (and did) change loops over all entities to loop over a simple array of entities, rather than use Java enumerators on Vectors.

Our techniques for reconstructing the simulator state at the end of a window depend on IDES organizational details. An entity is woken up when among all on-processor entities, its time-of-next-wakeup is least. At that point the entity removes from its own message queue every message whose receive-time is no larger than the wakeup time. Ordinarily this is equivalent to saying that the receive-time is equal to the wakeup time. However, we allow for the possibility that a scheduled wakeup is un-interruptible—for instance, when a non-preemptive server finishes servicing a job in service, in which case there may be messages whose receive-times are strictly less than the wakeup time. Messages are removed from the entity's message queue in increasing time-stamp order, each one being processed individually. Any new message that is generated as a result of this processing is handed off to the processor's *Router*. If the destination entity of the new message is on the same processor as the source, that message is immediately inserted into the destination's message queue, possibly changing the destination's time-of-next-wakeup. If the destination is off-processor the message is held by the Router, and will be transmitted later if its send-time is smaller than the computed edge of the synchronization window.

The number of entities managed on a processor can be very large, indeed, distributed simulation only makes sense when there is a great deal of simulation workload. We must be careful then to use a data-structure for the wakeup-time event list that "scales" as the number of entities grows. In addition, the data structure we use must support efficient identification and deletion of events, this to accommodate canceling an entity's wakeup time at w and rescheduling for some time $s < w$. The last requirement is that the data-structure efficiently support rollback and reconstruction.

Jeff Steinman, the inventor of BTB, has written about data structures used in his tool [13, 14]. Without going into details, we observe that our definition of the event horizon is different than his—his BTB window is defined as the least receive-time among *all* messages generated, ours is defined as the least receive-time among all off-processor messages. Our windows are necessarily larger than his, but in return our rollback requirements are more complex. His data structures are built around his notion of event horizon, and do not suit ours. Furthermore, there is no evidence of support for event cancellation.

As we will see, even with our expanded notion of window, the end-of-window synchronization costs are onerous in a distributed system, and so we are willing to accept the mildly more complex rollback requirements. To appreciate these, consider the desired state of the simulator at window edge $h(t)$: for every entity there should be a wakeup event that was scheduled at a time less than $h(t)$, which will occur at a time at least as large as $h(t)$. Similarly, each entity's message queue must contain all those and only those messages for which it is the destination, with send-times less than $h(t)$, and receive times at least as large as $h(t)$. Finally, each entity's state must reflect the effects of the last event to affect it prior to time $h(t)$.

The essential problem is that before the end-of-window time is known, the processor may process wakeups with time-stamps larger than $h(t)$, consuming messages with receive-times larger than $h(t)$, pushing entity state past time $h(t)$. We consider message queues, entity state, and event-list management separately.

For the message queues we borrow the trick used in good Time Warp simulators to not actually delete a message once consumed. Rather than have a queue from which messages are consumed once processed and into which messages (from co-resident entities) are inserted as generated, we use a queue into which messages may be inserted, and whose "next message" is pointed at in the interior of the queue. The message queue is maintained in monotone non-decreasing receive-time order. Once $h(t)$ is known we need to release "committed" messages—those we now know contribute to the state of the simulator at $h(t)$, and get rid of messages generated after time $h(t)$. It is easy to identify and eliminate the committed messages by scanning forward through increasing receive times. We have to examine each of the remaining messages individually to determine whether to keep or discard it, as *that* decision is based on the message's send time, and the messages are ordered by receive-time.

All that is needed to save entity state is a queue of state vectors (one queue per entity), organized in monotone non-decreasing time-stamp order. An entity is check-pointed at time w, just prior to being woken up at time w. Its variable state is copied into a state-vector, which is appended to the end of the entity's state-vector queue. Once $h(t)$ is known we need to restore the entity to the state it had at time $h(t)$ (excluding any modification that might have optimistically been made at time $h(t)$). This is accomplished by searching the queue for the state-vector whose checkpoint time w' is least among all those at least as large as $h(t)$, and restoring the entity's variable state to the vector stored at that point. It may be that no such state-vector is found, which means that the entity is already in the correct state. Following state restoration, all memory associated with saving entity state may be reclaimed.

Finally we consider the most complex data structure, the event-list. We could support rollback by leaving events behind in the event-list as they are processed, just as we did with the entity message queues. However, the event-list is the central data structure in the simulator, and the cost of accessing it tends to increase with the number of events in the list. In our experience most events that are executed are committed and so do not need restoration. Optimizing then to the common case, as events are processed they are removed from the event list, but added to the end of an auxiliary deque (queue where additions are made at the end, removals at either end). We therefore have a means of restoring to the event-list any events that were prematurely executed. Once $h(t)$ is known, we scan the deque from the front to identify committed events—those that occurred at times less than $h(t)$. Space for committed events can be reclaimed. The send time of each remaining event is compared with $h(t)$, if smaller the event is re-inserted into the event queue, if as great as $h(t)$ the event is discarded.

Restoration of speculatively processed events is only one correction that must be made. A second correction is needed due to the fact that as new events are generated, we insert them into the event-list. Consequently, any event inserted by the processing of an event at time $h(t)$ or larger needs to be removed. To support this correction, every time an event is inserted into the event-list (aside from the end-of-window correction described above), an addition is made to a deque of scheduled events. At the end of a window, the deque gives a description of every event scheduled during that window, in monotone increasing time-stamp order. Once $h(t)$ is known we simply scan the deque in *decreasing* time-stamp order looking for events scheduled at times as large as $h(t)$. For each such one we use the event description from the deque to find and remove the event in the event-list. Once a deque element scheduled before $h(t)$ is found, the remainder of the deque is discarded.

There is a final correction that must be made, one relating to event cancellations. Suppose that an entity's next wakeup time is scheduled to occur at time w. Now suppose at some time s the entity receives a message that causes its wakeup time to be reduced to $w' < w$. The first wakeup event must be canceled, and the second one inserted. However, whether or not that cancellation commits depends on whether $s < h(t)$. If we cancel an event, we must be prepared to restore it if it turns out that the cancellation was speculative (e.g., $s \geq h(t)$). Towards this end, when an event is canceled, we add a description of the cancellation action to yet another deque, this one ordered in increasing time-of-cancellation-action. When $h(t)$ is known it is easy to identify canceled events that must be restored—those whose time-of-cancellation-action are at least as large as $h(t)$, a set determined easily by scanning the cancellation deque from the back. Once one encounters deque elements with times less than $h(t)$, the remainder of the deque can be discarded.

The requirements of a central event queue can be determined from these operations we've described. Insertions, deletions, and searches must all be handled efficiently. There are a number of suitable data-structures, almost all of them are complex. The simplest of the group is the Skiplist [11], a randomized data structure that supports each operation in logarithmic time, with high probability. The IDES central event list is a Skiplist, chosen largely because of its simplicity. There remains a great deal of interest in identifying "the best" event-list algorithm (e.g., see [12]), however, the cost of event-list management pales against the computational overhead we expect for IDES applications, so our only concern is that the algorithm used be efficient and scalable. The Skiplist satisfies that requirement nicely.

4. Performance Evaluation

Next we turn to an empirical study of IDES, on a network of 14 PC clusters. Each cluster has 256Mb of memory shared among four PentiumPro processors running at 200MHz, each processor has 256K secondary cache. Seven clusters are interconnected through an Ethernet fast switch, the other seven through another, with a 200Mps connection between those switches. Each cluster is running Linux and JDK-1.1.3. Just-in-time compilation is not a part of this distribution.

The performance of a distributed simulation is always a complex function of many factors including load imbalance, communication volume, state-saving costs, and event granularity. Our goal is to assess the intrinsic overheads, while avoiding as many of these complicating factors as we can. In our study the workload driving the simulator is not representative of the applications we anticipate for IDES, nor is it representative of simulations of computer and communication systems. However, the facets this workload minimizes are those whose effects we can reasonably gauge. State-saving is minimal—yet we know that as the amount of state saved grows, the per-event execution time will increase proportionally. This increase would not be reflected in an optimal serial simulation, so we know that the relative performance of the simulation will decrease proportionally. By the same token, the computational granularity in our workload is minimal. As the amount of computational work per event grows the execution time per event grows, but this growth *is* reflected in an optimal simulation. As the execution granularity of an event grows, so too does the ratio of work-to-overhead, and the relative performance of the distributed simulation grows. Since the necessary state-saving and the computational grain are problem dependent, in this study we minimize both so as to evaluate the fixed costs of running the distributed simulation.

The workload we use is the PHOLD model (parallel hold) which figures prominently in many analytic models of parallel simulations. In the PHOLD model a fixed number of messages circulate through a network of entities. An entity processes a message with time stamp w by generating a random increment δ and a random destination d, then schedules the message to be received at d, at simulation time $w + \delta$. In our adaptation of PHOLD, δ is drawn from an Erlang-5 distribution (selected to provide substantial variance while avoiding a high frequency of very small samples). The usual form of PHOLD chooses destination entities uniformly at random. This would induce a large volume of communication, more so than one typically expects, and so much that its cost would surely obscure other factors. In our version we consider the entities to form a logical ring. When an entity selects a destination, it chooses a geometrically distributed offset, and with equi-likely probability adds, or subtracts that offset from its own ring position to compute a destination entity. In the runs reported here the mean of the geometric is 10, so that significant locality is enjoyed in the communication pattern. The workload associated with processing a message is minimal in the sense that there is no computational payload associated with it; all of the work involves choosing the next destination, and the time the message appears there. Message processing typically consists of removing an event from the event list, generating three random variables, generate a message and enqueue it, scheduling a new event.

The workload is balanced perfectly, by assigning a contiguous subchain of entities to each processor, each subchain having the same length. The state per entity is minimal, consisting of a few double precision numbers, and a few integers, just 72 bytes total.

By stripping down the model driver to the barest essentials, we hope to be able to determine where the fixed overheads are in the IDES engine. We go about that task by first looking at IDES run serially. If we disable state-saving, use an ordinary Skiplist as the priority queue, eliminate thread-switching, and eschew all end-of-window processing (because there are no windows!), then IDES on one processor is a perfectly good optimized serial simulator. We can study its performance to get a sense of how fast a Java-based simulator can run, to assess how large a model it can simulate, and to use as a baseline against IDES with extra costs, run on one processor.

The model size that can be simulated is a function of available heap memory, which is a Java command-line argument. All of our experiments are conducted on machines that have 64Mb memory per processor; in these experiments we allocate 50Mb memory for the heap. Experimentation established that largest model that could consistently be run without memory faults by the serial simulator has approximately 30,000 entities. The largest model the IDES-on-one-processor version can handle (that is, IDES with all state-saving and window control overheads enabled) is slightly smaller. This works out to be 1.67K bytes per entity. That strikes us to be on the high side, but not completely insane, as Java encodes a great deal of information in entities. Used memory is the cost of that encoding. Then, as we will see, once we introduce communication, the available memory drops considerably, presumably large chunks are taken over by communication entities.

The bottom line on serial performance is that when the number of entities simulated ranges from 3K to 30K, the rate at which messages are processed ranges from 1488 messages/second for 3K, to 1320 messages/second for 30K. The decrease may be attributed to increasing event-list costs as the size of the model increases; however, the event-list costs clearly scale, degrading by only 10% while the problem size increases by an order of magnitude.

On the architecture used, had we encoded this example in C, it would be reasonable to expect message processing rates that are an order of magnitude faster, on models that are an order of magnitude larger. For now, this is the price paid for compile-once-run-anywhere portability. Java aficionados promise great things for just-in-time compilation, and we will certainly view a significant increase in speed with favor.

When running IDES on one processor, we wished to measure some of the overheads encountered at the end of a window. We can force a window calculation through the memory control mechanism described earlier. The question then becomes, "how often?". In order to keep the state-saving cost per entity per unit simulation time constant across runs, we set the memory control parameter to trigger a window calculation after using memory equivalent to state-saving one quarter of the entities. Over the same range of entity counts from 3K to 30K as considered before, the message processing rates range from 1020 messages/second, to 920 messages/second, smaller than before by roughly a third. Instrumentation establishes that of that extra overhead, 45% is due to reconstruction of the event queue, 20% to entity rollback and memory reclamation (although the Java garbage collector is not explicitly called), and 35% to the execution-time costs of state-saving and manipulating the auxiliary deques we've associated with the event queue. The overhead due to end-of-window processing is something we manipulate by our memory control.

We turn now to data from runs performed on a relatively large-scale computation, on the architecture described at the beginning of this Section. We are principally interested in how capabilities grow, and how overheads grow, as the size of the architecture grows. In our experiments we fix the workload per processor by fixing the number of entities assigned to a processor, and then grow the problem and architecture together. The largest power-of-two number of entities/processor we can reliably simulate without exhausting memory is $2^{14} = 16,384$. Since some overheads are amortized over larger workloads, we also consider 2^{10} entities/processor and 2^{13} entities/processor. The first metric we examine is the aggregate committed messages per second. Figure 1 plots this metric as a function of architecture

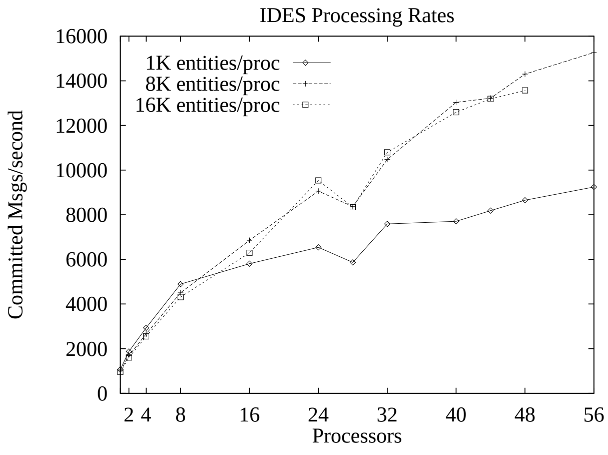

Figure 1. Committed messages processed per second for various problem sizes.

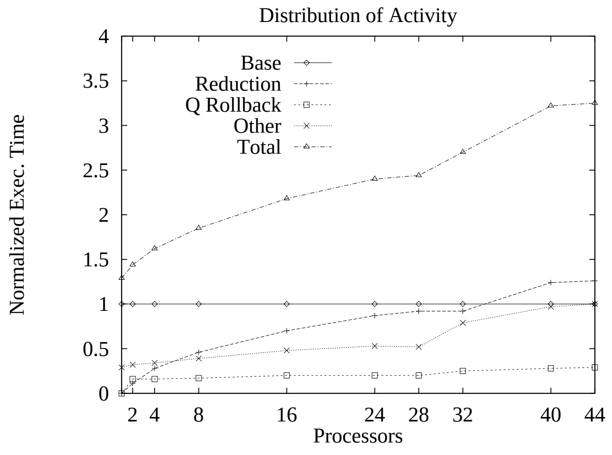

Figure 2. Breakdown of processing time.

size, on a set of pilot runs. Each data point is taken from one moderately long run. Statistical variation that would be smoothed out by formal statistical estimation of means is evident.

The benefits of amortizing overhead over more entities per processor is evident from the degree to which the 1K entities/processor performance falls off at larger numbers of processors. There seems to be little significant difference between 8K and 16K entities per node. The performance hit at 28 processors may be due to the fact that at that point, for the first point in the data, all clusters are in use and the full communication network is being used. After that point, adding more processors induces communication only between processors co-resident in a cluster. However, the most important point to glean is this—we have scaled the problem up to over three quarter of a million entities total and are making effective use of the aggregate memory of many many computers. The aggregate message processing rate is an order of magnitude larger than that of an optimized serial simulator. We have demonstrated that distributed simulation using Java is feasible.

However, the aggregate processing rate is clearly not increasing linearly as the number of processors increases. We are interested in understanding what overheads are growing. Examination of the data clearly indicates (not surprisingly) that the end-of-window barrier and minimum reduction are responsible.

There are two ways in which we can reduce the dominant overhead cost. First, the standard BTB technique is to withhold messages until $h(t)$ is known, and forward only those with send times less than $h(t)$. Every window there are *two* global synchronizations—one to establish $h(t)$, the other to ensure that all messages are flushed. One attraction of the pre-send method we've proposed is that the minimum reduction that establishes the window and the synchronization to flush messages are one and the same. By pre-sending messages we reduce the number of global synchronizations by a factor of 2. Another technique is to loosen the memory constraint that drives a processor to synchronize before it reaches its local event horizon. Experiments not presented here clearly demonstrate the performance advantages of pre-sending messages, and loosening the memory control.

Figure 2 illustrates the distribution of processing time when such optimizations are employed, on the PHOLD variation we've described, where each processor has $8K$ entities. For each activity we plot the average time a node spends in that activity, normalized by the *total* time an optimized serial simulator spends on the same problem. We plot times for three types of activity. The "Base" activity is the work that the serial simulator does (and hence is constant with respect to the normalization), the "Reduction" curve plots the time spent in the barrier synchronization and minimum reduction, "Q Rollback" is the time spent reconstructing the event queue at the end of a window, "Other" is all the rest (e.g., state-saving). The curve label "Total" reflects the addition of all four components.

We see that on problems using 8 or few processors, the majority of the time is spent in the same activities as does the optimized serial simulator. As the system size grows, so too do the overheads, so that by 44 processors, "Other", "Base", and "Reduction" are comparable. Relative to these costs, event queue reconstruction costs are quite small.

Observe that the overall increase in processing power over an optimized serial simulator can be deduced from these graphs—if the normalized per processor execution time is x using p processors, then the performance increase over the optimized serial case is p/x. For example, using 44 processors we have an increase of $44/3.25 = 13.5$.

Finally, we consider the impact of reducing the memory constraint. While this has the potential of reducing the number of synchronizations by as much as a factor of four, in this case it reduces the number of windows by something less because the normal BTB mechanism for terminating a window comes into play before the memory threshold is crossed. On the runs considered here, when memory is tightly constrained the number of messages committed per synchronization is essentially unaltered as the number of processors is increased. If memory is not tightly constrained, the number of messages committed per window starts at about four times that of the unconstrained case, but as the processor count approaches 56, this factor is reduced to 2.5. This occurs because as the number of processors increases, the number of values contributing to the GEH increases, and so the window size tends to shrink provided that the memory control mechanism does not kick in first. This is precisely what we observe in our data.

5. Conclusions

Discrete-event simulation of large complex systems requires a great deal of memory. One way to acquire more memory is to used distributed simulation. A natural alliance to consider is distributed simulation using Java, as Java offers the potential for using many different capabilities to network-wide simulation models.

This paper has described a Java-based distributed simulator, IDES. We examine its performance on a sparse workload model in order to assess its fundamental processing capabilities. While the raw processing power of IDES is smaller than one would like, this is due to Java's slowness and is a problem that may be corrected by just-in-time compilers. We examine IDES performance on a network of up to 56 PentiumPro based processors, find that it can increase overall processing capability by an order of magnitude and increase the memory available for modeling in proportion to the number of processors used.

Our work on IDES continues. We aim to investigate the impact of using just-in-time compilation, and to incorporate more sophisticated state-saving mechanisms. And, importantly, we are developing prototypical IDES applications to assess IDES's performance in its intended context.

References

[1] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: A Time Warp system for shared memory multiprocessors. In *1994 Winter Simulation Conference Proceedings*, pages 1332–1339, December 1994.

[2] Alois Ferscha and Michael Richter. Java based conservative distributed simulation. In *Proceedings of the 1997 Winter Simulation Conference*, pages 381–388, Atlanta, December 1997.

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

[4] Fabian Gomes, Brian Unger, John Cleary, and Steve Franks. Multiplexed state saving for bounded rollback. In *Proceedings of the 1997 Winter Simulation Conference*, pages 460–467, Atlanta, December 1997.

[5] Kevin Healy and Richard Kilgore. Silk : A Java-based process simulation language. In *Proceedings of the 1997 Winter Simulation Conference*, pages 475–482, Atlanta, December 1997.

[6] G. Horton, R. Kulkarni, D. Nicol, and K. Trivedi. Fluid stochastic petri nets: Theory, application, and solution. *European Journal on Operations Research*. To appear.

[7] Wolfgang Kreutzer, Jane Hopkins, and Marcel van Mierlo. SimJava–a framework for modeling queueing networks in Java. In *Proceedings of the 1997 Winter Simulation Conference*, pages 483–488, Atlanta, December 1997.

[8] D. M. Nicol and R. M. Fujimoto. Parallel simulation today. *Annals of Operations Research*, 53:249–286, December 1994.

[9] David Nicol, Michael Johnson, Ann Yoshimura, and Michael Goldsby. Performance modeling of the ides framework. In *Proceedings of the 1997 Workshop on Parallel and Distributed Simulation*, pages 38–45, Lockenhaus, Austria, June 1997.

[10] Ernest Page, Robert Moose Jr., and Sean Griffin. Web-based simulation in simJava using remote method invocation. In *Proceedings of the 1997 Winter Simulation Conference*, pages 468–474, Atlanta, December 1997.

[11] William Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Proceedings of the Workshop on Algorithms and Data Structures*, Ottawa, Canada, August 1989.

[12] Robert Rönngren and Rasul Ayani. A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation*, 7(2):157–209, April 1997.

[13] J. Steinman. Discrete-event simulation and the event horizon. In *Proceedings of the 1994 Workshop on Parallel and Distributed Simulation*, pages 39–49, Edinburgh, Scotland, 1994. The Society of Computer Simulation.

[14] J. Steinman. Discrete-event simulation and the event horizon part 2: Event list management. In *Proceedings of the 1996 Workshop on Parallel and Distributed Simulation*, pages 170–178, Philadelphia, 1996. IEEE Press.

[15] J.S. Steinman. SPEEDES: Synchronous parallel environment for emulation and discrete event simulation. In *Advances in Parallel and Distributed Simulation*, volume 23, pages 95–103. SCS Simulation Series, Jan. 1991.