Proceedings of the 1999 Winter Simulation Conference
P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, eds.

AN INVESTIGATION OF OUT-OF-CORE PARALLEL DISCRETE-EVENT SIMULATION

Anna L. Poplawski
David M. Nicol

Department of Computer Science
6211 Sudikoff Laboratory
Dartmouth College
Hanover, NH 03755-3510, U.S.A.

ABSTRACT

In large-scale discrete-event simulations the size of a computer's physical memory limits the size of system system to be simulated. Demand paging policies that support virtual memory are generally ineffective. Use of parallel processors to execute the simulation compounds the problems, as memory can be tied down due to synchronization needs. We show that by taking more direct control of disks it is possible to break through the memory bottleneck, without significantly increasing overall execution time. We model one approach to conducting out-of-core parallel simulation, identifying relationships between execution, memory, and I/O costs that admit good performance.

1 INTRODUCTION

Large scale simulations are limited not only by computational requirements (for which parallelism can be a solution), but also by memory requirements. Demand paged virtual memory systems are unable to *efficiently* provide the virtual memory space needed, because discrete-event simulations typically lack locality of reference. The problem is real, because interest in simulating very large models is growing. For instance, there is significant interest in simulating large portions of the internet (Cowie, Nicol and Ogielski 1999; Paxson and Floyd 1997). Application interest calls for an investigation of parallel simulations which do not fit in physical memory, referred to as *out-of-core* simulations. To our knowledge this paper is the first to consider solutions to this problem.

Unlike demand paging, out-of-core computations take direct charge of generating disk transfer requests. Such techniques have been developed in the context of *continuous* simulations such as the N-body problem (Salmon and Warren 1997). Data-parallel languages (Colvin and Cormen 1998; Thakur, Bordawekar and Choudhary 1994) have been developed to support scheduling disk accesses. However, data access patterns are significantly more random in discrete-event simulations, as is advancement of simulation time. These differences matter when considering a solution.

We describe one approach that is suitable in the context of large-scale network simulations. We identify some system issues that arise—temporal synchronization, pre-fetching strategies, allocation of memory and disk, scheduling—offer preliminary solutions, and *model* (using simulation, of course) the behavior of a parallel out-of-core simulation that uses these solutions. Exploring this model we discover that achieving high performance is entirely possible, depending on the nature of the network, the computational load, the memory demands, and the I/O system performance. Our work is preliminary, being best understood as a proof-of-concept.

The remainder of the paper is organized as follows. Section 2 describes our assumptions about the computation and the computer system upon which it is executed. Section 3 describes the various system design policies we model. Section 4 describes our experiments, and Section 5 presents the results. Section 6 contains our conclusions and future directions.

2 SYSTEM MODEL ASSUMPTIONS

We consider a model of a communication network in terms of a graph. Nodes represent network *entities* such as computers and routers. An edge—a communication *channel*—is weighted by the minimal latency of a message transferred across it. We assume that the memory demands of a network entity are small relative to the size of main memory. Entities are gathered together into *clusters*, using any one of a number of standard clustering algorithms. A cluster will be our minimal schedulable unit of work and I/O transfer. Latencies on channels between clusters play an important role in our synchronization algorithms.

We model the behavior of traffic by assuming that each node generates out-going events in accordance with a

Poisson process. A node's generation rate is proportional to the number edges at the node. A destination for the event is chosen uniformly at random from among the nodes that are adjacent to the source; the event is considered to have "arrived" at the destination after incurring the latency delay associated with the affiliated channel. A processing cost is associated with the event at the source, and another processing cost is associated with the target. The execution cost of a node is the sum of all such costs.

This model does not directly describe the flow of events across a network, but instead attempts to capture the effects of such flows. We adopted this higher level of model in order to benefit from significant computational efficiencies that it makes possible. Under these modeling assumptions the "work process" at a node is Poisson, with a rate that can be determined from topology. Rather than directly simulate event movement, we can *sample* workload and communication distributions from the appropriate probability distributions. The solution is not exact, because we assume independence between adjacent nodes' workloads, and they are clearly correlated.

We take the computer system hosting the simulation to be a shared-memory multiprocessor with uniform memory access times (although almost all memory references are local). We assume each processor has one disk, that may be accessed independently of all the other disks.

Finally, we assume that clusters are permanently assigned to processors, and that the shared memory is partitioned evenly among the processors for their private use (with the exception of some data-exchange areas).

3 SYSTEM ISSUES

When designing an out-of-core application, the primary goal is usually to minimize the total execution time while using only the available memory. A second goal is to not use more disk space than necessary. Our design focuses on the first goal. High performance is achieved if the pre-fetching is so successful that the CPUs rarely wait for data. Consequently, our focus will be on minimizing the time spent by processors blocked for data.

Next we consider some of the system issues involved in realizing high performance.

3.1 Synchronization

Processors in a parallel simulation must synchronize to ensure temporal consistency of the result. Of the general approaches (conservative or optimistic), we are pursuing a conservative one, largely because conservative methods are simpler, and have proven themselves in the networking problem domain. In an out-of-core context the synchronization requirements are intimately tied up with pre-fetching strategies, for a processor may have to block on the results of simulating some portion of the model that has yet to be brought into memory. We have developed an approach to synchronization that gives us some predictability for pre-fetching, and at the same time gives us the flexibility in scheduling that is needed when some workload is blocked, but other workload is not.

An entity cluster is our unit of schedulable work. In parallel simulation parlance it is a "logical process", meaning that it has its own simulation clock. Entities in the cluster are simulated in monotone non-decreasing time-stamp order, and the cluster engages in some synchronization protocol with other clusters to establish when it is safe to execute, and how far into simulation time it may execute. For each cluster i, let l_i be the smallest latency on channels connecting i with other clusters. Since channels are bidirectional, l_i is both the incoming and outgoing *lookahead* for cluster i, meaning that it takes at least l_i time for an event elsewhere to affect the cluster, and vice-versa. We will advance cluster i in time-steps of length l_i—once it is determined that the cluster may execute at time $(k-1)\cdot l_i$, the cluster is simulated across the window of simulation time $[(k-1)\cdot l_i, k\cdot l_i)$ without further synchronization. This represents a scheduling burst for i, at the end of which its simulation clock is taken to be $k\cdot l_i$. To determine whether it is safe to execute this burst, we check whether for every adjacent cluster j, its clock is at least as large as $(k\cdot l_i - l_j)$. For, if cluster j has simulated at least this far, then all future events from j to i will have time-stamps at least as large $k\cdot l_i$. When all such j satisfy this condition, cluster i must have received all external events that can affect its computation in its next burst. This is a variant of the synchronization method used with Ising spin models (Lubachevsky 1987).

3.2 Pre-fetching

To intelligently schedule memory resources we must be able to estimate memory requirements. In this study we assume that the memory requirements of the largest cluster can be determined prior to running the simulation, and used to estimate the number of clusters that can reside in memory concurrently. Let m denote the number of clusters *per processor* that can be co-resident. In our study we tended to be overly conservative; this is an area where improved policies are desirable.

Our pre-fetching strategy is understood in the context of the synchronization strategy. Cluster i executes in *bursts* that begin at timestamps of the form $k\cdot l_i$, for $k = 0, 1, 2, \cdots$. A processor builds a "schedule" by merging these sequences from all clusters assigned to it into a monotone non-decreasing sequence. For the sake of discussion, suppose the time-stamps of that schedule are $s_0, s_1, s_2, \cdots$, that for each j we denote the cluster whose burst starts at s_j by c_j; suppose also that s_k is the smallest element of the sequence for which the associated burst has not yet been

executed. Stated simply, the pre-fetching policy is to keep "the next" *m unique* clusters of the remaining sequence in memory. Stated more formally, at any time there is defined a "window" on the remaining schedule, beginning at s_k and extending to the m^{th} unique cluster in the sequence $c_k, c_{k+1}, \ldots$. Let us suppose the window contains c_k through c_{k+n}. Every cluster represented in this window is either resident in the memory or has been targeted to be brought into memory—once a cluster is executed, that burst is removed from the schedule. Completion of a cluster's execution may change the window. The lower edge always advances when the burst scheduled at s_k completes, moving forward to the next unexecuted burst (which need not be s_{k+1} because of the way we schedule). Whenever *any* cluster burst finishes, the upper edge of the window changes if and only if the cluster just executed (say c_j) has no un-executed burst in the current window. In this case it moves to the first element in $c_{k+n+1}, c_{k+n+2}, \ldots$ that is not in the current window. If the new upper edge points to a cluster $c_u \neq c_j$, c_j is scheduled to be written to disk and c_u is scheduled to be loaded in its place.

3.3 Scheduling

Cluster execution bursts are independently schedulable units of work. Since up to m clusters may be resident in a processor concurrently, we have scheduling options to consider. We do not insist that the clusters in a pre-fetch window be simulated in monotone non-decreasing order. For example, a cluster may be memory-resident but prohibited from being executed owing to synchronization constraints with clusters on other processors. When a scheduling decision must be made, among all clusters c_j that are memory-resident and can be safely executed, the one with least burst time s_j is chosen. If no such cluster exists, the processor blocks until one does. The set of clusters eligible for execution changes dynamically as scheduled I/O operations complete, and as clusters on other processors complete execution bursts to relax synchronization constraints.

3.4 Other Issues

The allocation of disks and disk locations must also be considered in an out-of-core simulation. Our model assumes one disk-per-processor (although our techniques are in no way limited by that assumption), and that a disk is controlled solely by its processor, containing only model state and event lists for clusters assigned to that processor.

We allocate a fixed block of disk space for each cluster, sized to accommodate estimated maximal cluster-state requirements *and* maximal event-list space requirements. A cluster's state and event list are moved together in all disk I/O operations. We note in passing that our out-of-core context impacts the event-list data structure implementation—we must avoid implementations that rely on absolute memory pointers.

Most of the events created by a cluster's execution are for the cluster's own entities, and go immediately into the cluster's own event list. Events targeting entities in different clusters are stored in main memory until the target cluster is next executed, at which point they are merged into the target cluster's event list.

Problems we've not addressed in this paper include general clustering methods, and dynamic load balancing. The networks we consider have "natural" clusters that we exploit; in general clustering is an important problem. We use a simple list-scheduling heuristic to assign clusters to processors; the networks studied are homogeneous enough for this technique to work well, and the workload is static enough (statistically) so that dynamic remapping is not needed. These all are areas of our future consideration.

4 EXPERIMENTAL MODEL ASSUMPTIONS

We consider a model of the internet, using topologies generated by the Georgia Tech Internetwork Topology Models package (Thomas and Zegura 1994), and parameters based on actual systems. Network graphs reflect a 3-level hierarchy used to define clusters. Network nodes are associated with spatial locations; communication latencies between nodes are based on Euclidean distances between them. The graphs are generated randomly. A cluster has, on average, 500 nodes, and an average of 7.5 channels connected to nodes in other clusters, referred to as inter-cluster channels. The minimal latency between two clusters is 10 msec, whereas the average latency between clusters is 100 msec. Intra-cluster latencies are much smaller, with an average 1 msec and minimum of 100 μsec.

Our baseline assumptions concerning the simulation representation of this model is that an entity requires 1KB of state, and an event requires 100 bytes. The generation rate for an entity is proportional to the number of channels connected to that entity, with an average of 0.01 events per msec. The processing cost of an event (either generated or received) is 50 μsec. The network studied has 5,000,000 entities. We explore the sensitivity of performance to changes in these baseline parameters.

We assume that execution overhead (when amortized) due to synchronization and scheduling is negligible compared to event processing costs. We do explicitly account for the cost of blocking due to synchronization constraints.

The baseline model is not especially large, relative to memory capacities of modern multiprocessors; less than 450MB of simulated physical memory is required for all scenerios simulated. Most require less than 125MB. The entire model state, excluding event-lists, requires 5GB of simulated storage space. However, our results show that

model size is not as important as the *ratio* of problem size to physical memory size. We can study smaller-than-realistic models and argue that the performance-degrading overhead to actual work ratio is no better than it would be for realistic sized models, thereby making our performance indices meaningful. The advantage of studying smaller models is that the computational effort needed to evaluate them is smaller, enabling us to explore more of the design space.

Finally, we presume the parallel system of interest has 10 processors, with one dedicated disk for each processor. We model disk behavior with the very accurate DiskSim tool (Ganger, Worthington and Patt 1998). The DiskSim models we use have been validated against actual behavior. However, the validated disks are older, and hence slower, than current disks. Our studies would show better performance were we to model faster disks.

5 EXPERIMENTAL RESULTS

The space of possible systems includes ones where good performance is impossible, and others where good performance is trivially achieved. In order to assess the potential for out-of-core parallel simulation, we need to assess how realistic are the regimes where good performance is possible. We do this by selecting a baseline system which has good performance but which is on a cusp of the performance surface, and then examine the sensitivity of performance to changes in those model assumptions.

We characterize performance using the standard notion of "utilization", or fraction of overall time spent in a specified activity. CPU utilization is the fraction of time a CPU spends executing useful work; I/O utilization is the fraction of time a disk spends servicing requests. Both of these are computed as the average CPU (respectively, disk) ratio of time spent executing (respectively, servicing I/O) to the longest overall finishing time among all processors. We also compute "Wait Time" utilization to gauge the fraction of time a CPU spends blocked on either synchronization constraints, or I/O. It turns out to be tricky to separate contributions to wait time due to synchronization from that due to I/O, and so we don't. However, we can measure the wait time of separate experiments where it is assumed that the entire model is always memory-resident. Our graphs will refer to this as the "Sync Time" utilization. In all the data shown, this utilization is negligible, meaning that synchronization is not a problem inherent to our model. It does *not* mean necessarily that the waiting time utilization in the out-of-core setting does not have a significant component due to synchronization. Introduction of I/O delay affects synchronization behavior.

The baseline system enjoys a CPU utilization of almost 90%, while at the same time I/O utilization is just under 75%; at this point we're very successful at overlapping CPU and I/O, but also vulnerable to increases in I/O demand (because there is very little "slack" capacity in the I/O system at this point), or to decreases in CPU demand.

Figure 1 supports our claim, made in the previous section, that the size of the model is not as important as the ratio of problem size to memory size. The default size of 5,000,000 entities is shown in the center of the graph. While the problem size is varied, the size ratio, the number of entities per cluster, and the average and minimum channel delays remain constant. Increasing the number of clusters in memory allows more flexibility in scheduling executions. Thus, even though the amount of I/O and CPU work increase at the same rate, the performance improves slightly as long as the disks do not become saturated with requests.

One experiment simply increases the size of an entity. This has no effect on the computational requirement, but increases the I/O bandwidth requirement. The effect of this (not shown) is to degrade CPU utilization, linearly in entity size. When using a model size of 2,500,000 entities, tripling entity size reduces CPU utilization from 80% to 30%, simply because the I/O bandwidth requirements are increasing and the CPUs increasingly wait for I/O. Another way of increasing I/O transfer overhead is to increase the *event* state size, the effects of which are plotted in Figure 2. Now the volume of events transferred to disk depends on link latency, for an event is "in the system" for as long as it is actively being communicated. In order to observe this effect without changing the synchronization, for each event size considered in Figure 2 we varied the intra-cluster latency between the baseline value of 1 msec, and 10 times that (we multiplied each intra-cluster latency by 10). Considering sensitivity, for the baseline case the CPU utilization is not much affected by increasing event size by a factor of 10. Likewise, leaving the event size fixed at 100 bytes and increasing latency by a factor of 10 does not significantly impact performance. However, when larger latencies *and*

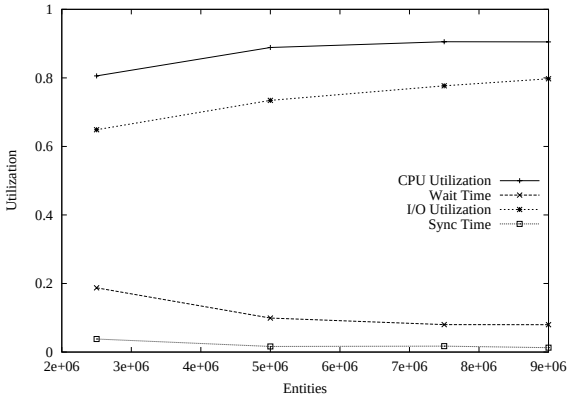

Figure 1: Effects of Varying Problem Size. The Ratio of Physical Memory to Problem Size Remains Constant as the Problem Size Increases.

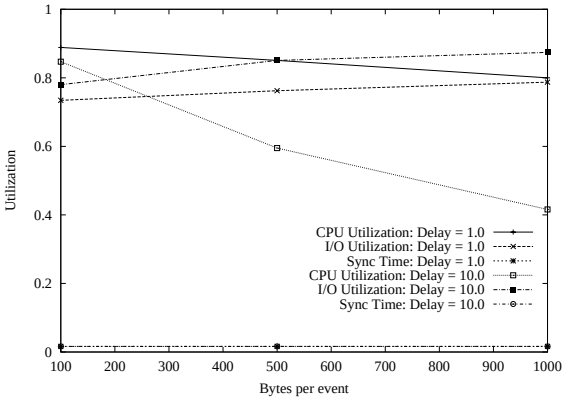

Figure 2: Effects of Varying Event Size and Intra-Cluster Channel Delays.

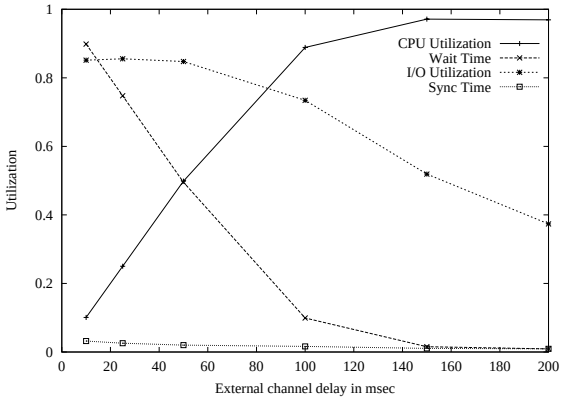

Figure 3: Effects of Varying Inter-Cluster Channel Delays.

larger event sizes are assumed, CPU utilization decreases significantly. Consider: intra-cluster events that go to disk are those generated in one cluster burst and received in another. For the baseline latency only 1% (1 msec intra-cluster latency relative to 100 msec inter-cluster latency) of the events of interest go to disk. Increasing the event size by a factor of 10 increases the bandwidth requirements by a factor of 10, just as does increasing latency by a factor of 10 for 100 byte events. However, increasing both increases the contribution of these events to I/O bandwidth demand by a factor of 100. The fact that performance drops by only a factor of 2 in the face of such increases means that the inter-cluster cluster events simply don't contribute much to the I/O demand.

This experiment leads us to another, that examines sensitivity to changes in *inter-cluster* latencies. These latencies directly affect the frequency (in simulation time) with which clusters are written to and read from disk. Halving the inter-cluster latencies doubles the I/O demand, but does not change the computational workload. Figure 3 illustrates the point (note the that baseline point of 100 msec is in the middle of the graph). CPU utilization is essentially a linear function of inter-channel latency between 10 msec (where utilization is 10%) and 100 msec (where it is 90%). However, changes in I/O utilization are significantly smaller over this range. This is because the overall finishing time is dominated by I/O; increasing inter-channel latency increases I/O cost and the overall execution time by essentially the same amount, hence the ratio of the two is relatively unaffected. However, increasing inter-cluster latency also proportionally increases the computation per execution burst, the net effect of which is to better mask the delays associated with pre-fetching clusters. The strong message delivered by this data is that clustering to maximize inter-cluster channel latencies is extremely important.

Next we consider sensitivity to changes in computational demand. The assumed computational time per event reflects the granularity of an event; simulations with significant detail require more work per event. On today's computers 100 μsec per event represents coarse granularity, whereas 10 μsec per event represents fine granularity. Figure 4 shows the effect of varying granularity from 10 μsec to 100 μsec with the baseline value of 50 μsec (20,000 events per second per processor) near the left edge of the graph, for two different entity sizes (1KB and 0.5KB). Clearly these changes strike at the heart of the baseline system's (1 KB entity) precarious balance of CPU and I/O costs. Doubling the event processing rate cuts the CPU demand in half, and the CPU utilization effectively drops by a factor of 2 in response in the portion of the graph where the I/O utilization remains constant. The CPU utilization curve for the 0.5KB entities mitigates this somewhat, by reducing I/O demand by a factor of two, but this really is just a horizontal shift in essentially the same curve.

Another way to change the computational demand is to change entity event generation rates, which changes the volume of events executed per unit simulation time. For a fixed set of channel latencies changing the generation rate

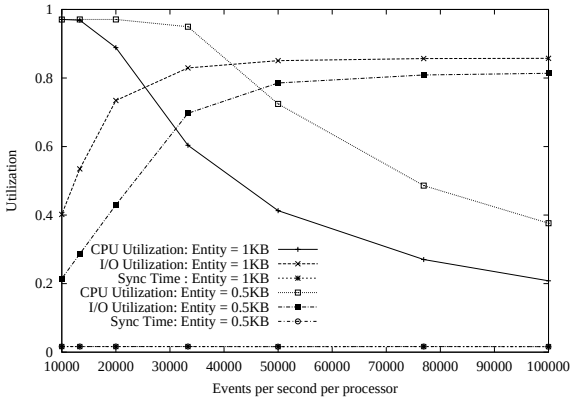

Figure 4: Effects of Varying Computation Requirements, for 1KB and 0.5KB Sized Entities.

will also change I/O demands (which are proportional to the product of inter-channel latency and event generation rate). Figure 5 plots the response of performance to variations in entity event rate; the baseline value of 0.01 events/msec is in the middle of the graph. The general shape of the graph is similar to that of Figure 3, because the same phenomena is occurring. In both cases the changing parameter proportionally affects the number of events executed per cluster burst (which affects the effectiveness of pre-fetching) and affects the I/O demand.

Figure 6 shows the effects on the CPU usage when both the inter-cluster channel delay and the event generation rate are changing. These are the two most significant parameters to the simulation system which are expressed in terms of simulation time. This graph demonstrates the need for simulations which can be broken into clusters such that the inter-cluster channel delays are high relative, primarily, to the rate are which the entities are generating events. More work needs to be done to improve the results for simulations which cannot meet these characteristics.

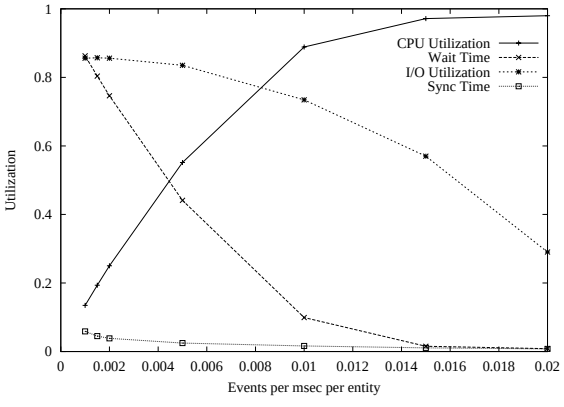

Figure 5: Effects of Varying Event Generation Rates at Entities.

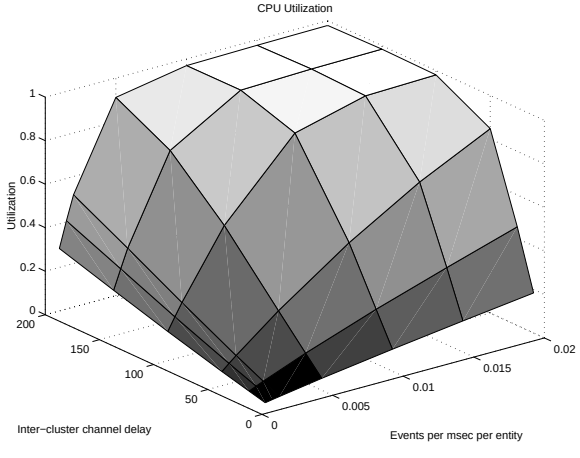

Figure 6: Effects of Varying Event Generation Rates at Entities and Inter-Cluster Channel Delays.

6 CONCLUSION AND FUTURE WORK

Although the size of physical memory has long been considered an upper limit on the size of a system which can be simulated, we have shown that this is not necessarily the case. We present a model of a simulator which computes the I/O time, CPU time, and time during which the CPU is blocked. Using this, we can determine under what conditions running a simulation outside the limits of physical memory is feasible. We have also proposed the use of a relatively simple conservative synchronization method which minimizes the time spent by the disk. We have shown that in a configuration where less than 10% of the simulation data fits in physical memory, the CPU can be utilized over 90% of the time given simulation characteristics which fall in the proper ranges.

We have also identified the features of the model to be simulated which are significant for obtaining good performance out-of-core. First, the graph of entities in the model must be able to be divided into clusters such that the average of the minimum inter-cluster delays over all edges connected to a cluster is as large as possible. Second, a combination of a reasonably small entity size, a moderate event rate caused by a somewhat detailed simulation, and a high workload must exist in the simulation so that the CPU has enough work to do to offset the cost of I/O. Other details of the model to be simulated, such as the size of the model, the number of nodes in a cluster, the delay on edges within a single cluster, and the size of an event are less significant in most cases.

There is a great deal which remains to be studied in the area of out-of-core parallel discrete-event simulation. Many of the issues, some of which are new to the PDES community, have been identified in this paper. By exploring these issues farther we hope to be able to reduce the I/O overhead farther for the sets of parameters which give poorer performance using our current design. We also must explore in more detail the effects of heterogeneity in the system to be simulated. We plan to eventually implement our design as an actual simulator which we hope will be used in practical applications where such a tool is needed.

ACKNOWLEDGMENTS

This work was supported in part by DARPA contract N66001-96-C-8530, and by NSF Grants ANI-98-08964 and CDA-98-02068.

REFERENCES

Colvin, M. A. and T. H. Cormen 1998. ViC*: A compiler for virtual-memory C*. In *Third International Workshop on High-Level Parallel Programming Models and Supportive Environments*, 23–33. Full paper available

as Dartmouth College Computer Science Technical Report PCS-TR97-323.

Cowie, J. H., D. M. Nicol, and A. T. Ogielski 1999. Modeling the global internet. *Computing in Science & Engineering*.

Ganger, G. R., B. L. Worthington, and Y. N. Patt 1998. The DiskSim simulation environment. Technical Report CSE-TR-358-98, Department of Electrical Engineering and Computer Science, The University of Michigan.

Lubachevsky, B. D. 1987. Efficient parallel simulations of asynchronous cellular arrays. *Complex Systems* 1:1099–1123.

Paxson, V. and S. Floyd 1997. Why we don't know how to simulate the internet. In *Proceedings of the 1997 Winter Simulation Conference*.

Salmon, J. and M. Warren 1997. Parallel out-of-core methods for n-body simulation. In *Proceedings of the Eighth SIAM Conference of Parallel Processing for Scientific Computing*.

Thakur, R., R. Bordawekar, and A. Choudhary 1994. Compiler and runtime support for out-of-core HPF programs. In *Proceedings of 8th ACM International Conference on Supercomputing*.

Thomas, M. and E. W. Zegura 1994. Generation and analysis of random graphs to model internetworks. Technical Report GIT-CC-94-46, College of Computing, Georgia Institute of Technology.

AUTHOR BIOGRAPHIES

ANNA L. POPLAWSKI received a B.S.E. degree in Computer Science from Princeton University in 1995 and a M.S. degree in Computer Science in 1998 from Dartmouth College where she is currently a Ph.D. student. Her primary research interest is in parallel discrete-event simulation.

DAVID M. NICOL is a Professor of Computer Science, at Dartmouth College. His research interests include parallel processing, and computer modeling/simulation; he is Editor-in-Chief of the ACM Transactions on Modeling and Computer Simulation, and is a Senior Member of IEEE. He holds a B.A. degree in mathematics from Carleton College, M.S. and Ph.D degrees in computer science from the University of Virginia.