Proceedings of the 1994 Winter Simulation Conference
ed. J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila

TIMING SIMULATION OF PARAGON CODES USING WORKSTATION CLUSTERS

Phillip M. Dickens
ICASE
NASA Langley Research Center
Hampton, VA 23681, U.S.A.

Philip Heidelberger
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598, U.S.A.

David M. Nicol
Department of Computer Science
The College of William and Mary
Williamsburg, VA 23185, U.S.A.

ABSTRACT

This paper describes a parallel simulation software tool, **LAPSE-nx-lib**, that executes on a network of workstations. Using this tool, message-passing codes written for an N processor Intel Paragon can be executed on M workstations ($N \geq M$). The timing estimate produced by the simulator is a prediction of how long the code would have run had it been executed on the N processor Paragon. **LAPSE-nx-lib** combines **nx-lib**, a publicly available library that provides the functionality of Intel Paragon **nx** message-passing calls on networks of workstations, with LAPSE, a parallelized timing simulator that we originally developed to run on the Paragon. We report on our early experiences with this tool on a network of Sun Sparc-10 workstations.

1 INTRODUCTION

nx-lib (Stellner *et al.* 1994) is a software library developed by researchers at the University of Munich allowing the development and execution of codes developed for the Intel Paragon multicomputer, using an ordinary network of workstations. Codes run under **nx-lib** have the functionality of Paragon codes. However, owing to temporal sensitivities, it is possible for the execution path of a code to be different on the workstations than it would be on the actual Paragon. Furthermore, calls to system clocks reflect the workstation's own sense of time, not the time on the Paragon being simulated. Thus **nx-lib** cannot be used to provide accurate timing estimates of how long the code would have taken had it been run on the Paragon.

We have ported the distributed memory LAPSE (Large Application Parallel Simulation Environment, Dickens *et. al* 1994) to the **nx-lib** environment, creating a tool we call here **LAPSE-nx-lib**. The combined system augments **nx-lib**'s functionality with more accurate temporal behavior and information. This paper briefly describes LAPSE (which has been reported upon in more detail elsewhere (Dickens *et al.* 1994)), and our early experience with **LAPSE-nx-lib** on a small network of Sun Sparc-10's. We primarily examine performance issues related to distribution of application and simulation processes, paying special attention to slowdowns and speedups. The performance obtained so far is very promising.

2 OVERVIEW

A parallel program for a distributed memory machine consists of M application processes, distributed among $N \leq M$ processors. Most parallel programs are constructed so that $N = M$, an equivalence we presently assume. Application processes communicate through message-passing using explicit calls to system library routines. From the application's viewpoint, the program executes application code between calls to these library routines. The length of time spent "inside" one of these subroutines depends on the communication network and its behavior—things that are unobserved by the application. On the other hand, the network views the program as "bursts" of execution by application processes, punctuated by requests for network services. The details of the execution bursts are irrelevant to the network, only the specifics of the request matter, e.g., time of request, length of message, message destination. In LAPSE the application code is instrumented to measure the length of the execution bursts, and redirect message-passing library calls to LAPSE routines. The LAPSE routines remap certain application coordinates (like identity of target processor on a send) to internal LAPSE coordinates, and then cause the requested application request to occur. Next the LAPSE routine notifies a simulator process of the application activity. The set of simulator processes coordinate with each other to simulate the network traffic induced by the application requests.

The application process must provide the LAPSE simulator with estimates of the time it requires to execute between calls to message library routines.

LAPSE estimates execution time by modifying the application assembly code to count the number of instructions executed. (The counter is updated only at basic block boundaries, a technique also used in Proteus (Brewer *et al.* 1991), Tango (Davis *et al.* 1991) and WWT (Reinhardt *et al.* 1993.)) The overhead to implement the instruction counting varies from 5 to 12 instructions per basic block, depending on the source code language. From the instruction counts we estimate time passage by taking into consideration the measured effects of cache miss ratios, different numbers of cycles for different instruction types, and so on.

To obtain accurate timing of the application, the operating system overhead for sending and receiving messages needs to be properly accounted for. LAPSE currently estimates such operating system overheads. For example, the overhead (in instructions) to execute a message send can be modeled as $A + B \times L$ where A is a startup cost, L is the message length and B is the cost per byte. Estimates of A and B can be obtained by measurements of the operating system. Experiments on the Intel Paragon on a variety of application codes have shown LAPSE predict performance accurately, usually within about 5%.

LAPSE traps all references to message library routines by recompiling the application code with a file of macros that redirect message library calls to LAPSE routines. These LAPSE interface routines are linked into the same memory address space as the application process. The LAPSE routine identifies the call as an *event*. Parameters of the event are collected, e.g., event type (read, write, probe), number of instructions executed since the last event by this process, characterization of the the message event (e.g., message id, starting memory location, length, recipient processor(s), recipient process id). The requested activity is actually initiated (e.g, a message is sent to be received by a different LAPSE interface routine), and then a message is sent to an application simulator process (the one assigned to this application process) describing the event.

A key contribution of the LAPSE system is an efficient parallelization of the Paragon communication network simulation. Details of the synchronization scheme are provided elsewhere (Dickens *et al.* 1994). Here it suffices to comment that the synchronization protocol is conservative (e.g., processes never roll back or save state), and that the protocol takes advantage of long periods where the application execution path is insensitive to timing—LAPSE's success on the Paragon is tied directly to this excellent lookahead it can exploit.

Several other projects use direct execution simulation of multiprocessor systems. Among these we find two pertinent characteristics, (i) the type of network being simulated, and (ii) whether the simulation is itself parallelized. Table 1 uses these attributes to categorize relevant existing work, and LAPSE.

LAPSE and HASE (Howell *et al.* 1994) simulate a message passing network with a parallelized simulator. WWT (Reinhardt *et al.*) simulates a shared memory environment with a parallelized simulator. MaxPar (Chen *et al.* 1990), Maya (Agrawal *et al.* 1994), Proteus (Brewer *et al.* 1991) and Tango (Davis *et al.* 1991) simulate a shared memory network with a serial simulator. RPPT (Covington *et al.* 1991) and Simon (Fujimoto 1983) simulate a message passing network with a serial simulator.

Table 1: Direct Execution Simulation Tools

Tool	communication (simulator)
LAPSE	message-passing (parallel)
HASE	message-passing (parallel)
MaxPar	shared memory (serial)
Maya	shared memory (serial network)
Proteus	shared memory (serial)
RPPT	message-passing (serial)
Simon	message-passing (serial)
Tango	shared memory (serial)
WWT	shared memory (parallel)

Among most current simulators other than our own, simulation of cache-coherency protocols are an important concern. However, the Intel Paragon does not support shared virtual memory. Coherency protocols complicate the simulation problem considerably, but are a facet LAPSE need not deal with. However, existing work has identified context-switching overhead as a key performance consideration, and it is one that directly affects us. As much as an order of magnitude improvement has been observed when a direct-execution simulator uses its own light-weight threads constructs to accelerate context-switching (for small grain sizes). One of the self-imposed constraints of LAPSE is that it simulate general codes written by others. Some tricks for placing multiple virtual processes in the same address space are not available to us, at least not without substantial compiler-oriented work. For instance, HASE uses a graphical user interface from which code is generated. All variables "global" to a virtual process can be declared as local variables to a super-process, and placed on the run-time stack. For LAPSE to do the same would require a complete reparsing and analysis of multiple file applications, a task from which we shrink.

The Wisconsin Wind Tunnel (WWT) is to our knowledge the only working multiprocessor simulator that uses a multiprocessor (the CM-5) to execute the simulation (HASE was not operational in parallel at the time Howell *et al.* (1994) was published). It is worthwhile to note the differences between LAPSE and the WWT. First, the WWT simulates cache-coherency protocols for applications running on shared memory machines whereas LAPSE simulates message-passing applications. Second, because of the tight coupling of shared memory applications, the WWT needs to keep its processes in close synchrony, and does so with fairly frequent barrier operations. (This method of synchronization is a special case of the YAWNS protocol discussed in Nicol *et al.* (1989) and Nicol (1993).) Because of the looser coupling of message-passing codes, better lookahead possibilities are available to LAPSE. This lookahead comes from the observation that, in many applications, long portions of the execution path are independent of timing behavior. In such a case, the application code can be executed well in advance of actually simulating the timing. Where the execution path is not independent of timing, lookahead can still be obtained provided there is a lower bound on the operating system overhead required to send or receive a message.

3 LAPSE-NX-LIB

nx-lib is designed to accept most Paragon codes, and run them on a variety of workstations. Consequently the effort required to port LAPSE to the networked workstation environment should, in principle, be minimal. The most consuming requirement would seem to be the instrumentation of a different instruction set to provide instruction counts. However, in the specific case of the Sparc architecture, both the Wisconson Wind Tunnel (Reinhardt *et al.* 1993) and Proteus (Brewer *et al.* 1991) already do this, and we are able to adopt their solutions (but have not yet done so. The performance reported here is of uninstrumented code, execution times are estimated using clocks).

The Intel Paragon supports multitasking on each processor node, indeed this feature makes LAPSE possible. Like LAPSE, **nx-lib** creates a unique Unix process for every Paragon process that is loaded, and multitasks them on workstations. In addition to the application processes, LAPSE creates simulation processes that communicate with each other, and with application processes that are actually running the application code. To **nx-lib**, a code running on LAPSE is no different from any other multitasked Paragon code. Messages on the Paragon are uniquely distinguished by message type, processor id, and process id. Message type parameters are defined by the programmer to distinguish between messages with different meanings, the processor and process id information specifies the destination. **nx-lib** supports this model by creating a TCP socket for every uniquely referenced Paragon process. All messaging activity related to process (processor id, process id) is mapped to the corresponding socket.

We did have to modify LAPSE in two respects, one minor, and one less so. The first release of **nx-lib** was prone to errors occurring in codes that use dynamic memory allocation. Subsequent releases include file headers to be added by hand to source code, in order to prohibit interrupts in the middle of application calls to dynamic memory routines. The more significant issue arises because of a difference in philosophy between **nx** and **nx-lib**. Despite the Paragon's ability to multitask, the designers of **nx** have a single-process-per-processor view of the computation. Blocking calls to **nx**, e.g. a synchronous receive, cause busy-waiting in **nx** rather than a context switch. This was unacceptable to LAPSE; it transforms all blocking calls into non-blocking calls with explicit context switches. Effectively this means that LAPSE decides when contexts must be switched, with the exception of quanta expiration. By contrast **nx-lib** has a multitasked view of the computation, and switches context on blocking calls. Because of this **nx-lib** does not support the explicit **nx** call to release the thread. We therefore modified LAPSE to undo its control of context switching. This was accomplished simply, using macros.

The most significant consideration one has using **LAPSE-nx-lib** is process distribution. Given M actual processors and an N processor code, LAPSE allows one to either map one simulator process and M/N application processes to each actual processor, or to partition M into sets of size A and S ($M = A + S$), placing one simulator process on each of S processors, and multitasking M/A application processes on each of the other processors. Initial experimentation has suggested that the separated option is desirable when LAPSE uses a detailed network simulator (which has more simulation and synchronization activity) and to otherwise use the former partitioning. A key point is that the Paragon has a high-speed, high bandwidth network, and that network contention is rarely an issue running LAPSE. This is not true for **LAPSE-nx-lib**, since most workstation networks share a single communication line which effectively serializes communication. One of the points we need to address with **LAPSE-nx-lib** is that of distribution. Intuition suggests that communication overhead will play a larger role in distributing **LAPSE-nx-lib** processes than it did in distributing LAPSE processes.

4 EXPERIMENTS

The experiments reported here are of a code, SOR (Successive Over-Relaxation, Press *et al.* 1988) used to solve linear systems of equations arising from tHe discretization of partial differential equations. The code assigns a $G \times G$ subgrid of points to every processor. Each iteration, each processor communicates the values of points on its boundary to the processors sharing that boundary. Each processor then receives boundary values from its neighbors, and engages in a computation phase where it updates each of its assigned points with a weighted average of its previous value and the values of points adjacent in the mesh. Each processor monitors the changes in point values, retaining the magnitude of the largest such. The processors then check for convergence, by cooperatively identifying the largest point difference in the entire mesh. If that change is less than the pre-specified tolerance value the computation terminates. If not, the boundary values are again exchanged and the process continues.

The key parameters governing this computation are G^2 (which determines the computation assigned to each processor) and N, the total number of processors. As G^2 grows, the number of messages passed each iteration does not change, but the length of the messages grows proportionally with G. The computation/communication grows as $G^2/G = G$. As G grows the workload of the LAPSE network simulators does not change (since it handles only descriptions of messages, not the messages themselves), but the workload of the application processes does. In fact, increasing G serves to improve relative performance, since the computation phase whose work increases in G is perfectly parallelizable.

At the time of this writing, the workstation network available to us is not yet executing large **LAPSE-nx-lib** codes reliably (this appears to be a problem between the network configuration and **nx-lib**, not LAPSE). The performance we report is limited to five Sparc-10 workstations, and four SOR processes. Nevertheless some interesting and important observations are revealed despite the small size of the problem.

Figure 1 plots the performance of the native SOR code running on **nx-lib** alone on four workstations, as a function of the problem size. We present a normalized performance metric, the number of seconds expended on each grid point per iteration. Means and standard deviations for 10 runs on 50×50, 200×200, and 500×500 meshs are plotted. The value for the smallest mesh is nearly 10 times that of the larger grids. The underlying reason is a relatively fixed

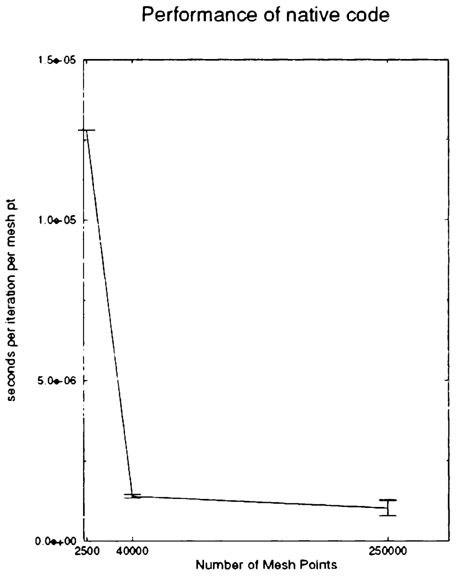

Figure 1: Performance of SOR code on native nx-lib configuration

amount of interprocessor communication overhead is incurred independent of the problem size. The high cost per point is suffered because there are relatively few points over which to amortize this cost. The 200×200 mesh has 16 times more workload and almost completely amortizes that cost, as shown with comparison with the 200×200 mesh performance shows.

Figures 2,3, and 4 plot the performance of a four process SOR code under **LAPSE-nx-lib**, on meshes of size 50×50, 200×200, and 500×500, respectively. Each data point reflects the sample mean and standard deviation from ten runs. Two curves are shown, one associated with placing one simulator process on each workstation jointly with multitasked application processes, and another associated with allocating one extra workstation that holds a single simulator. The horizontal axis plots the number of workstations dedicated to the application processes (the separated placement curve uses one more processor than given on this axis).

One interesting feature of the joint placement data is that the improvement in performance from one processor to two is more than a factor of two. This feature, observed on all graphs, can be attributed to the higher non-scaling overhead costs (e.g., increased paging, more context switching) suffered on one processor. The large difference between the separated

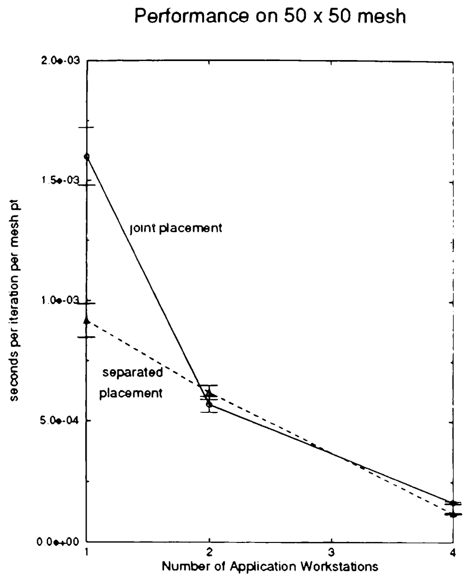

Figure 2: Performance of SOR code on 50 × 50 mesh

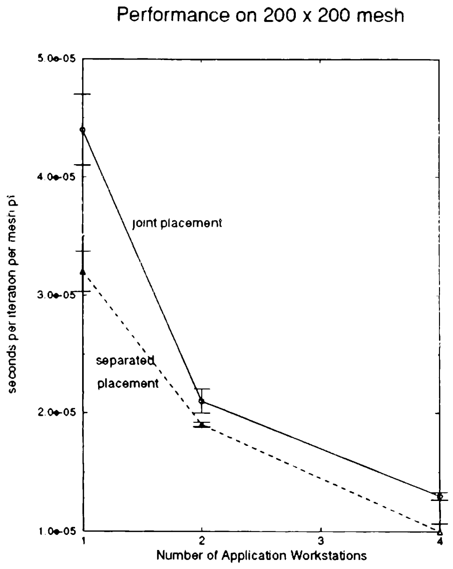

Figure 3: Performance of SOR code on 200 × 200 mesh

and joint placement methods in the case of one processor is due in part to the increased parallelism (two processors as opposed to one), and in part to reduced context switching (due to moving the simulator off on its own workstation). The next point of interest is the scale of each graph. The normalized costs of the 200 × 200 mesh are an order of magnitude less than those of the 50 × 50 grid; the normalized costs of the 500 × 500 mesh are an order of magnitude less than those of the 200 × 200 mesh. As we have already seen that the per-point **nx – lib** costs are virtually the same on the large two grids, we can attribute the decreasing normalized cost to the increasing amortization of LAPSE overheads.

It is also interesting to note the relative lack of difference between the joint placement and separated placement strategies in the two and four application processor cases. We cannot from this data infer anything about placement strategies for larger numbers of workstations, but at least on this data it appears that the simulator workload is not the bottleneck, else its serialization would manifest itself as a relative degradation of performance.

LAPSE overheads can be assessed by considering *slowdown*, or the degree to which running a code un-

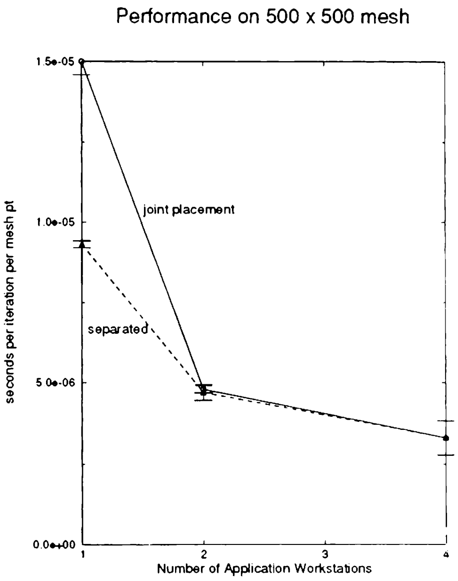

Figure 4: Performance of SOR code on 500 × 500 mesh

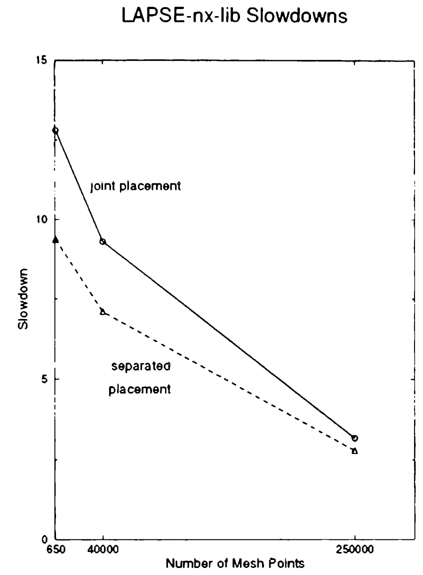

Figure 5: LAPSE-nx-lib slowdowns

der **LAPSE-nx-lib** runs more slowly than running that same code under **nx-lib** alone. Figure 5 plots these as a function of grid size. As the problem size increases, increasingly more of the computation is spent in the application processes, and not **LAPSE-nx-lib**. Furthermore the slowdowns are quite good in the context of direct-execution simulators. LAPSE slowdowns on the Paragon are better; the higher slowdowns here are due to the higher cost of communication.

5 CONCLUSIONS

LAPSE is a parallelized direct-execution simulator of Intel Paragon codes. We have ported LAPSE to run under **nx-lib**, a software package that provides **nx** library functionality on networks of workstations. The combination, **LAPSE-nx-lib**, provides more temporally accurate execution behavior and timing information. This paper describes the combined system, and provides preliminary reports on its performance.

ACKNOWLEDGMENTS

Small portions of this paper are reprinted with permission from "A Distributed Memory LAPSE: Parallel Simulation of Message Passinf Codes", 8^th Workshop on Parallel and Distributed Simulation, IEEE Press. This work was performed while Philip Heidelberger was on sabbatical at ICASE This work was performed while David Nicol was on sabbatical at ICASE. This work is also supported in part by NSF grant CCR-9201195.

BIBLIOGRAPHY

Agrawal, D., M. Choy, H.V. Leong, and A. Singh 1994. Maya: A simulation platform for distributed shared memories. In *Proceedings of the 8^{th} Workshop on Parallel and Distributed Simulation*, pages 151–155, July 1994.

Brewer, E., C. N. Dellarocas, A. Colbrook, and W. E. Weihl 1991. PROTEUS: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, September 1991.

Chen, D., H. Su, and P. Yew, 1990. The impact of synchronization and granularity on parallel systems. In *Int'l. Symp. on Computer Architecture*, pages 249–248, May 1990.

Covington, R., S. Dwarkadas, J. Jump, S. Madala, and J. Sinclair 1991. Efficient simulation of parallel computer systems. *International Journal on Computer Simulation*, 1(1):31–58, June 1991.

Davis, H., S. Goldschmidt, and J. Hennessy 1991. Multiprocessor simulation and tracing using Tango. *Proceedings of the 1991 International Conference on Parallel Processing*, pages II99–II107, August 1991.

Dickens, P., P. Heidelberger, and D. Nicol 1994. Parallelized direct execution simulation of message-passing programs. Technical Report 94-50, ICASE, July 1994.

Fujimoto, R. 1983. A simulator of multicomputer networks. Technical Report UCB/CSD 83.137, ERL, University of California, Berkeley, 1983.

Howell, F., R. Williams, and R. Ibbett 1994. Hierarchical architecture design and simulation environment. In *MASCOTS '94, Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 363–370, Durham, North Carolina, 1994. IEEE Computer Society Press.

Nicol, D., C. Micheal, and P. Inouye 1989. Efficient aggregaton of multiple LP's in distributed memory parallel simulations. In *Proceedings of the 1989 Winter Simulation Conference*, pages 680–685, Washington, D.C., December 1989.

Nicol, D. 1993. The cost of conservative synchronization in parallel discrete-event simulations. *Journal of the ACM*, 40(2):304–333, April 1993.

Press, W., B. Flannery, S. Teukolsky, and W. Vettering 1988. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, 1988.

Reinhardt, S., M. Hill, J. Larus, A. Lebeck, J. Lewis, and D. Wood 1993. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference*, pages 48–60, Santa Clara, CA., May 1993.

AUTHOR BIOGRAPHIES

PHILLIP M. DICKENS, Ph.D., University of Virginia, 1992, is a staff scientist at ICASE (Institute for Computer Applications in Science and Engineering) at the NASA Langely Research Center. His research interests are in performance modeling and tools for performance predictions.

PHILIP HEIDELBERGER received a Ph.D. in Operations Research from Stanford University in 1978. He has been a Research Staff Member at the IBM T.J. Watson Research Center since 1978. In 1993 - 1994, he spent sabbaticals at ICASE and Cambridge University. He is an area editor of the ACM's *Transactions on Modeling and Computer Simulation*, and has served as program chairman of the 1989 Winter Simulation Conference, and program co-chairman of the ACM Sigmetrics/Performance '92 Conference. He is a Fellow of the IEEE.

DAVID M. NICOL received a B.A. in Mathematics from Carleton College in 1979, and received a Ph.D. in Computer Science from the University of Virginia in 1985. He is an Associate Professor at the College of William and Mary, and an associate editor for the ACM's *Transactions on Modeling and Computer Simulation* and for the ORSA *Journal on Computing*. His research interests are in parallel simulation, performance analysis, and algorithms for mapping parallel workloads.