Research Feature

Parsec: A Parallel Simulation Environment for Complex Systems

When parallel architectures, parallel databases, and wireless networks are scaled to larger configurations, sequential simulations become computationally intractable. UCLA researchers have addressed this problem with a simulation environment that supports a diverse set of algorithms and architectures and provides a visual programming interface.

Rajive Bagrodia
Richard Meyer
Mineo Takai
Yu-an Chen
Xiang Zeng
Jay Martin
Ha Yoon Song
UCLA

Systems are now being designed that can scale to very large configurations. Examples include parallel architectures with thousands of processors, wireless networks connecting tens of thousands of computers, and parallel database systems capable of processing millions of transactions a second. In some cases, systems have grown considerably larger than envisaged by their designers.

Design and development costs for such systems could be significantly reduced if only there were efficient techniques for evaluating design alternatives and predicting their impact on overall system performance metrics. Due to the systems' analytical intractability, simulation is the most common performance evaluation technique for such systems. However, the long execution times needed for sequential simulation models often hamper evaluation. For instance, it can easily take weeks to simulate wireless networks with thousands of mobile nodes that are subject to the interference and signal-fading effects of a realistic environment.

Also, the simulation of parallel programs typically suffers from slowdown factors of 30 to 50 *per processor*.[1] This means that simulation of a parallel program executing for five to 15 minutes on a 128-node multiprocessor can take anywhere from a few weeks to a few months—even on state-of-the-art sequential workstations. As the size of the physical system increases, the models' memory requirements can easily exceed the workstations' capacity.

The limitations of sequential model execution have led to growing interest in the use of parallel execution for simulating large-scale systems. Widespread use of parallel simulation, however, has been significantly hindered by a lack of tools for integrating parallel model execution into the overall framework of system simulation. Although a number of algorithmic alternatives exist for parallel execution of discrete-event simulation models, performance analysts not expert in parallel simulation have relatively few tools giving them flexibility to experiment with multiple algorithmic or architectural alternatives for model execution.

Another drawback to widespread use of simulations is the cost of model design and maintenance. The design and development costs for detailed simulation models for complex systems can easily rival the costs for the physical systems themselves.

The simulation environment we developed at UCLA attempts to address some of these issues by providing these features:

- An easy path for the migration of simulation models to operational software prototypes.
- Implementation on both distributed- and shared-memory platforms and support for a diverse set of parallel simulation protocols.
- Support for visual and hierarchical model design.

Our environment consists of three primary components: a parallel simulation language called Parsec (parallel simulation environment for complex systems); its GUI, called Pave; and the portable runtime system that implements the simulation algorithms.

0018-9162/98/$10.00 © 1998 IEEE

Three primary types of parallel synchronization protocols have been developed and are now being studied: conservative, optimistic, and mixed.

Parsec is based on the Maisie simulation language, with these significant modifications:

- It uses simpler syntax.
- It uses modified language, to facilitate porting code from the simulation model to the operational software.
- It has a robust and extensible runtime kernel that is considerably more efficient than its predecessor.
- It uses new protocols to predict parallel performance.

PARALLEL SIMULATION PROTOCOLS

A simulation consists of a series of *events*, which must be executed in the order of their time stamps. On a single processor, these events can be placed in a central queue so that the global event list algorithm can correctly order them. When run in parallel, however, not only is the event list distributed so that each processor has only a portion of it, but events may also arrive asynchronously from other processors. For parallel discrete event simulation (PDES), then, additional information is required to ensure that each processor executes events in their correct order. Three primary types of parallel synchronization protocols have been developed and are now being studied: conservative,[2] optimistic,[3] and mixed[4] (which may include submodels executing in either conservative or optimistic mode).[5]

Simulation structure

Parallel simulation models are commonly programmed as collections of logical processes, in which each LP models one or more physical processes in the system. Events in the physical system are modeled by message communication among the corresponding LPs, and each message carries a time stamp for the time when the corresponding event occurs in the physical system (which means that the terms *event* and *message* can be used interchangeably). Each LP has these variables:[4]

- *Earliest output time (EOT).* The EOT is a lower bound on the time stamp of any future messages that the LP may send. If EOT_s is infinity for some LP *s*, the remaining LPs can be executed independently of *s*. A *sink process* is an example of such an LP.
- *Earliest input time (EIT).* The EIT is a lower bound on the time stamp of any future message that the LP may receive. The EIT_s of an LP *s* is the earliest time that it may receive a message from another LP. The EIT of an LP is infinity if no other LP sends messages to it. A *source process* is an example of such an LP.
- *Lookahead.* The lookahead for an LP is the future time interval over which the LP can completely predict the events it will generate. The lookahead is used to calculate EOT. For instance, consider a first-in, first-out (FIFO) server that serves each incoming job for Δ time units. If the server is idle at some simulation time *t*, its lookahead is Δ and its EOT is $(t + \Delta)$.

In general, the lookahead (and hence the EIT and EOT) of an LP depends on the LP's state, which changes dynamically during the simulation. The relative value of these variables determines the degree or granularity of synchronization that must exist in a model. If the EOT of an LP is much larger than its current simulation time, other LPs in the system are likely to execute independently for longer time intervals. Generally, though, an LP can only *estimate* the time stamp of future messages, and the EOT represents a lower bound on this estimate. It is possible (and often the case) that even though an LP does not send any messages over a long interval, its EOT is only marginally ahead of its current simulation time at any given time in the simulation.

Synchronization

As it is commonly used, a PDES model is either *optimistic* (all LPs executed in the optimistic mode) or *conservative* (all LPs executed in the conservative mode).[6] A conservative LP cannot tolerate causality errors—events executing out of time-stamp order—so it will only process events with time stamps less than its EIT. There are a number of algorithms that compute the EIT of each LP in a distributed manner, and Parsec includes many of them. In general, a model's lookahead and communication topology significantly impact the performance of conservative algorithms.

An optimistic LP may process events with time stamps greater than its EIT, but the underlying synchronization protocol must detect and correct violations of the causality constraint. The simplest mechanism for this is an optimistic LP that periodically saves—or *checkpoints*—its state. If the LP is then found to process messages in an incorrect order, it can be rolled back to an appropriate checkpointed state, with the events then processed in their correct order. Also, an optimistic algorithm must periodically compute a lower bound, also called the *global virtual time* (GVT), on the time stamp of the earliest global event; checkpoints time-stamped earlier than GVT can be reclaimed. Using our model, it is sufficient for an optimistic LP to preserve at least one checkpointed state with a time stamp smaller than its EIT. (The minimum of the EIT of all optimistic LPs is a reasonable lower bound on the GVT of the model.)

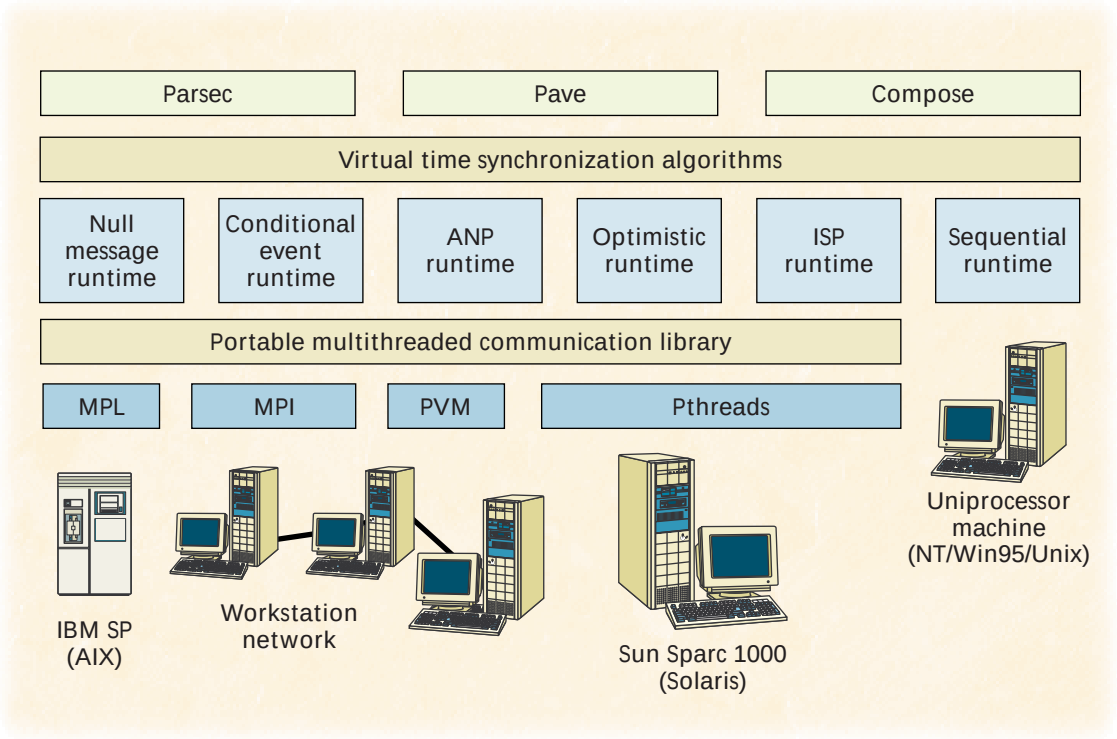

Figure 1. The UCLA simulation environment provides a number of user interfaces and implements several synchronization protocols using a portable threaded communication library, making it available across a variety of computer platforms.

Given appropriate mechanisms to advance the EIT and EOT of the conservative or optimistic LP, it is possible to implement a PDES model composed from optimistic and conservative submodels.[4] In general, any of the GVT computation algorithms, conservative algorithms, or even a combination of them can be used by a PDES to compute the EIT of each LP, regardless of the execution mode of the individual LP in the model. The choice of a specific algorithm for a given scenario is a matter of efficiency rather than correctness.

In our aggressive null-message-based scheme, whenever the EOT of an LP, say *s*, changes, EOT_s is sent using a null message to other LPs; the null message may of course be piggybacked on a regular message when feasible. On receipt of a null message, an LP recomputes its EIT and EOT and propagates changes to other LPs. Given a model with no zero-delay cycles (where all LPs have a zero lookahead), such an algorithm will eventually advance the EIT (and hence the time) of every LP, regardless of whether it executes in conservative or optimistic mode.[4] Hybrid protocols are also useful for the composition of autonomous simulators, where each simulation may internally use a conservative, optimistic, or sequential protocol.

SIMULATION DEVELOPMENT ENVIRONMENT

We developed our simulation environment so that an analyst could explore the utility of different simulation algorithms and parallel architectures for the execution of a given model. The environment supports a number of front ends for programming models: the C-based Parsec simulation language; a C++ library, called Compose, that can be interfaced with native C++ code to execute parallel simulations written in C++; and Pave. Figure 1 illustrates the environment, including the supported hardware, communication packages, OSs, synchronization algorithms, and program interfaces.

Parsec

Parsec adopts the process-interaction approach to discrete-event simulation. A Parsec program consists of a set of *entities* and C functions. Each entity is an LP that models a corresponding physical process; entities can be created and destroyed dynamically. Events are modeled by message communications among the corresponding entities. Each message carries a logical time stamp matching the time at which the corresponding event occurs in the physical system. An entity may also schedule for itself a special message, called a *timeout*, for a specific time in the future. This message is often used by an entity to simulate the passage of time in the physical system, and its handling has been optimized in the system.

Wireless networking. The following code fragment is a Parsec entity for simulating part of a wireless net-

working protocol. The entity implements the media access control (MAC) layer, which takes packets from a network-routing protocol and transmits them with a radio. The entity may also receive packets from the radio and forward them up the stack to the network router.

```
message ClearedToSend {};
message MACPacket {IP dest; byte
  buffer[PACKET_SIZE];};
message NetworkPacket {IP dest;
  byte buffer[PACKET_SIZE];};
message RadioPacket {IP dest; byte
  buffer[PACKET_SIZE];};
message RequestClearToSend {IP
  dest;};

entity MAC_Protocol (IP myIP) {
bool            transmitting;
ename           radio, network;
message MACPacket macPacket;

while (true) {
  receive (NetworkPacket np) when
    (!transmitting) {
      transmitting = true;
      push(np, buffer);
      send RequestClearToSend
        {np.dest} to radio;
  }
  or receive (ClearedToSend cts) {
    macPacket =
    buildPacket(pop(buffer));
    send macPacket to radio;
    hold(TRANSMISSION_DELAY);
    transmitting = false;
  }
  or receive {RadioPacket rp) {
    send rp to network;
  }
 }
}
```

First, several message types are defined just like a C `struct` code declaration. Then the entity itself is defined, much like a C function. The body of the entity consists primarily of a receive statement block, which has several *resume clauses*. The first resume clause accepts a packet from the network routing layer, but the when clause prevents this message from being accepted while the entity is transmitting an earlier packet.

When the entity does receive a packet, it instructs the radio to request a transmission window (Request ClearToSend), buffers the packet, and changes its state to "transmitting" until the packet is sent. The packet must be buffered because the variable *np* is local to the resume clause where it is declared. When the entity receives the clear signal from the radio, it delivers the packet and uses the hold statement to advance its time to after the transmission. The MAC layer may also receive the radio packets, which it forwards to the network routing layer. Notice that there are two forms of the send statement: one which uses the message type and a parameter list and a simpler one that just sends a message variable.

Facilitate migration. One of Parsec's major design goals is to facilitate the migration of simulation models into operational software. Parsec is built around a thread-based message-passing programming kernel called MPC (message-passing C), which can be used to develop general-purpose parallel programs. The only difference between Parsec and MPC is that a Parsec model executes in logical time, with all messages in the system being processed in the global order of their time stamps.

In contrast, each entity in an MPC program can autonomously process messages in the physical order of their arrival. Because of the common set of message-passing primitives used by both environments, it is relatively easy to transform a Parsec simulation model into operational parallel software in MPC. This has been done in the domain of network protocols, in which simulation models of wireless protocols were directly refined and incorporated into the protocol stack of a network operating system for PCs.[7] For parallel execution, the model is refined by first partitioning entities among available processors and then providing algorithm-specific information to improve parallel performance. For conservative protocols, this might imply adding code to compute the application-specific lookahead for an entity and providing dynamic connectivity information that restricts the transmission of updated EOT values of an entity to a subset of LPs in the model.

Similarly, for optimistic algorithms, the analyst may modify the default checkpointing or garbage-collection frequency. When sequential performance proves inadequate, parallel models are typically derived as refinements of equivalent sequential Parsec models. The models' performance can always be compared with the consistent sequential version to measure the benefit of parallel execution. (Of course, for sufficiently large configurations, resource constraints may preclude sequential execution.)

Visual model design

The Parsec Visual Environment (Pave) facilitates the visual design of related simulation component libraries, the construction of simulation models from these components in a simple visual framework, the generation of Parsec code for the models, and the optimization of the models for parallel execution. Component libraries

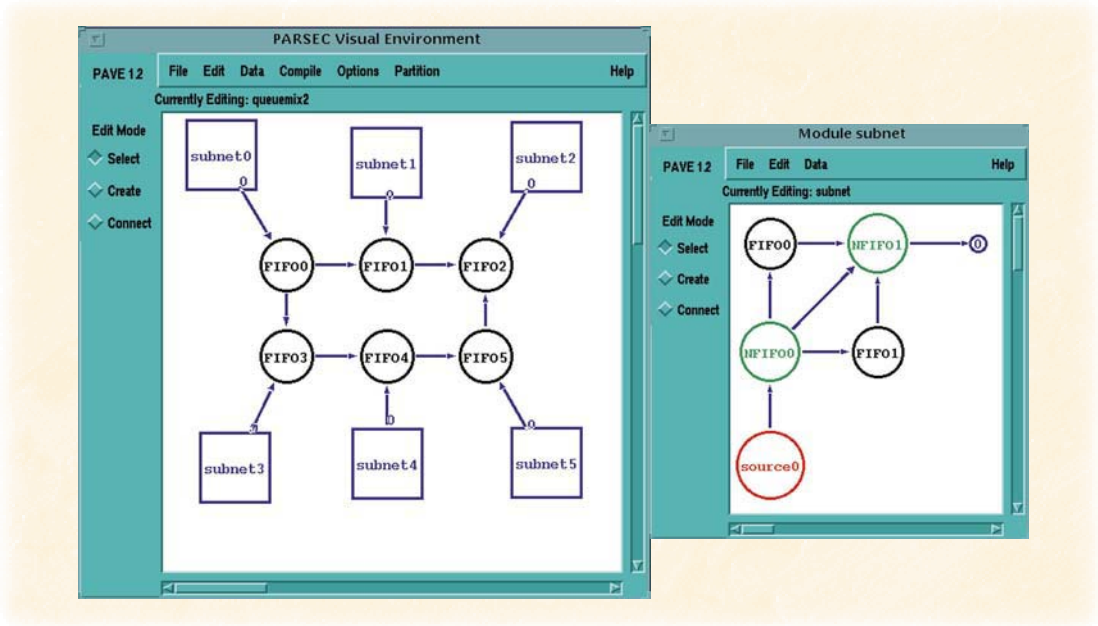

Figure 2. Modular simulation developed in Pave and queuing network topology with a subnet module.

can be created for domains such as queuing networks or mobile network infrastructures. Unlike most existing visual simulation tools, Pave was designed specifically to support parallel simulations.

Pave allows library design of related simulation components as well as the composition of these components into specific simulation models. Every level of model design has visualization, and features support both modular design and parallel execution, both of which permit scalability. Modular design allows structures to be constructed in ever larger increments, and parallelism allows their efficient execution.

Graphical programming. Using an enhanced flowchart description language, entities can be programmed graphically using common flowchart notation for sequential control flow and additional notations to specify message operations (sends and receives). Alternatively, existing hand-coded entity code can be included with only minor modifications.

A set of connected entities (for example, a subcircuit or a network node) may be grouped together as a module, which may be replicated or nested in higher level modules. This permits a hierarchy of models with an arbitrary level of nesting and arbitrary mixing of modules and entities at different levels. This hierarchical modeling capability supports parallelism and scalability by allowing models to be designed in large chunks.

Figure 2 shows a modular simulation developed in Pave that consists of various queuing servers and a traffic generator ("source0"). The figure shows the building of a simulation model from a collection of queuing entities and subnet modules. The smaller window shows the internal structure of the subnet module, which consists of a collection of queuing entities and a traffic generator. The entities are connected with simple mouse clicks.

Pave can be used to develop any Parsec model, and it can be tuned to particular domains through a set of appropriate parameterized objects. In the networking domain, for example, a set of library objects can be constructed to represent traffic generators, routers, routing protocols, and different types of hardware (such as radio transmitters).

Partitioning algorithms. For parallel execution, the model must be effectively partitioned onto the set of available processors to balance computation load and minimize message traffic between processors. We are developing for Pave a set of partitioning algorithms, including both user-assisted and totally automatic varieties. Pave's hierarchical design feature allows models to be designed in tightly coupled chunks, which in turn provide the system with useful information for dividing the problem among available processors.

For example, in Figure 2 the subnet module is a tightly coupled subset of the simulation having primarily local communication. Dividing the entities of the subnet module onto multiple processors would lead to a great deal of interprocessor communication. A partitioning algorithm being tested with Pave takes into account the logical relationship of the LPs within a module. Further, users will be allowed to specify a binding factor for each type of module—information which the designer may know *a priori*.

To enhance the overall simulator performance, each synchronization protocol may require additional information for each entity. For instance, the null-message-based synchronization scheme outlined earlier requires communication topology information to construct the source and destination sets for each entity. Minimizing these sets reduces synchronization overheads and leads to more efficient execution. This topology information is automatically generated by Pave. Protocol-specific parameters that are specified textually in the model—like lookahead and state-saving frequency—may also be specified through the visual interface.

Runtime system

A portable kernel executes Parsec programs on sequential and parallel architectures. Parallel Parsec

A number of simulation models in diverse domains have been developed using the Maisie and Parsec simulation environments.

programs may be executed in two modes—as parallel programs or as simulation models. When executed as parallel programs, the programmer must explicitly indicate any synchronization among the entities in the program; the runtime system delivers messages to each entity in the physical order of their arrival. When programs are executed as simulation models, every message has a time stamp derived from a simulation clock, and the program (or model) must be executed such that all messages are processed in the global order of their time stamps. (In some cases, messages may be processed out of time-stamp order, as long as the result of the execution is equivalent to that of processing messages in their time-stamp order.)

Parsec supports the following synchronization algorithms:

- A sequential or global event-list algorithm.
- Three parallel conservative algorithms: a null-message-based algorithm, a conditional event algorithm, and a combination of the two (the accelerated null message (ANM) algorithm).
- An optimistic algorithm based on space-time simulations.[8,9]
- The ideal simulation protocol (ISP), based on the critical path concept, which predicts a realistic lower bound on the execution time of a given parallel model.[10]

As shown in Figure 1, the kernel provides a unified simulation runtime system for implementing these synchronization algorithms on a variety of architectures. When running an experiment, the programmer specifies the synchronization algorithm as a command-line option, linking the appropriate library with the runtime system. A number of factors govern the performance of parallel simulation of a given application:

- Application characteristics that determine the inherent parallelism in the model.
- Partitioning methods used to decompose the model into multiple partitions.
- Architectural and operating system characteristics, including processor speed, communication latency, and context-switching costs.
- Overhead of the synchronization protocol, which is perhaps the most important factor.

Experimental measurements on the execution time of a given parallel model rarely offer sufficient insight into the underlying causes of the observed performance. The ISP algorithm provides a way for the analyst to separate those overheads due solely to the simulation protocol from other overheads of parallel execution, thus allowing the analyst to estimate a realistic lower bound on the execution time of a parallel simulation program.

The ISP algorithm computes the execution time for a given model on a given architecture by executing the model twice. The first run uses the null message protocol (or, in general, any synchronization protocol) to collect a complete legal message trace for that model. During the second run, each entity can use the trace to determine the order in which incoming messages are to be processed—without using any synchronization protocol.

Consider the following example: An entity receives a message m, with time stamp u, when the EIT of the entity is v and $v < u$. In general, the entity can process this message only when its EIT is u or larger. Assuming that the entity has no other messages that can be processed, it must remain idle as long as its EIT is less than u. However, using the message trace, the ISP "knows" the sequence in which incoming messages are accepted by the entity, thus allowing it to process the "next" message in the sequence as soon as it arrives.

In this example, if m is the next message in the sequence, an entity can process it as soon as it is received if there is no blocking (or rollback) overhead. In this manner, the ISP *excludes* overheads that result solely from the simulation protocol but *includes* overheads resulting from model partitioning, message transmission and buffering, and other factors related to the execution of a parallel program. With the speedup computed by the ISP algorithm, the analyst can determine if the observed inefficiencies in the execution of a parallel simulation model result primarily from implementation inefficiencies in the simulation protocol or from the inherent lack of concurrency in the parallelized model.[9]

CASE STUDIES

A number of simulation models in diverse domains have been developed using the Maisie and Parsec simulation environments: VLSI circuit models, telecommunication models, ATM models, wireless network models, parallel architecture and I/O system models, and parallel program models. In the two applications discussed here, results on the effectiveness of parallel model execution are provided as speedup graphs, where speedup is defined as

$$\text{Speedup}(N,A) = T_S / T_P(N,A)$$

where T_S is the time for sequential execution of the model with the sequential global event list algorithm implemented using a splay tree, and $T_P(N,A)$ is the execution time on N processors using simulation algorithm A. (Sometimes the one-node conservative implementations were faster than the global event list algorithm, in which case we used the faster time to compute speedup.)

The experiments were executed on a 32-node IBM

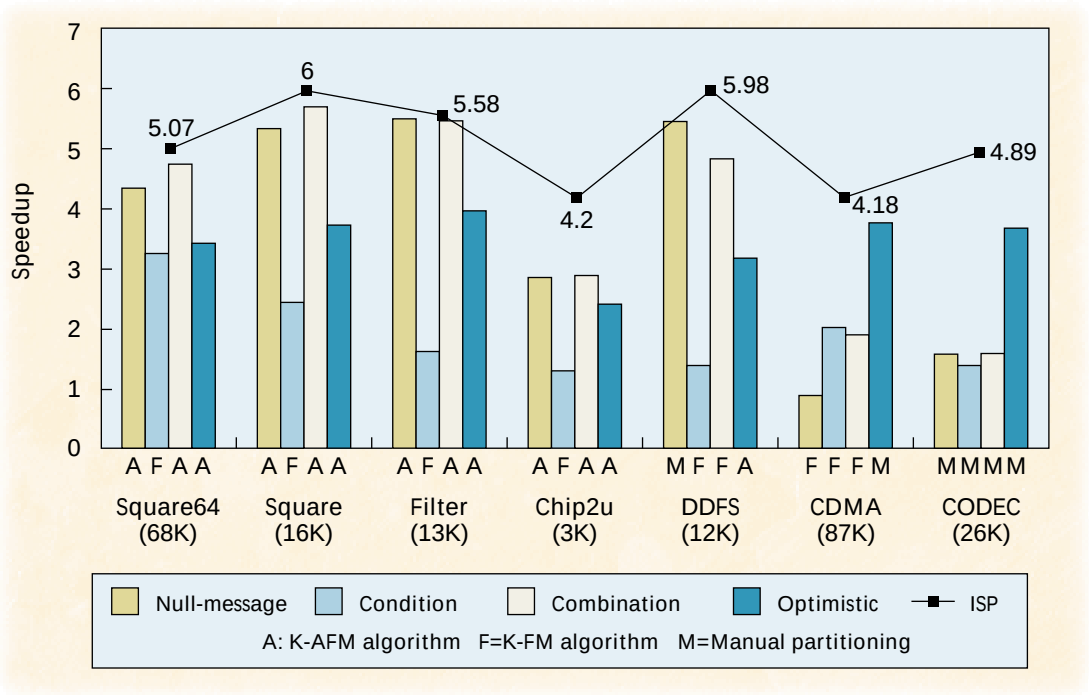

Figure 3. VLSI circuit simulation results, showing the best partitioning method for each synchronization algorithm over a set of sample circuits.

SP2 running AIX—each node being an RS/6000 workstation processor with 256 Mbytes of main memory and 66.7-MHz clock speed—and on a Sparc 1000 running Solaris 2.5.1 with eight SuperSparc CPUs running at 51 MHz and having 512 Mbytes total shared main memory.

Switch-level circuit simulation

Circuit simulation is a significant bottleneck in the design of VLSI circuits, and parallel execution can considerably reduce the simulation time for large circuits. We have developed Mirsim, a parallel switch-level circuit simulator that is a Parsec implementation of Irsim, an existing event-driven simulator incorporating a linear model of MOS transistors. Using a variety of circuit-partitioning techniques on the IBM SP2, we used the Parsec implementation to simulate a number of circuits with both conservative and optimistic protocols.[11]

Our benchmarks were a number of circuits ranging in size from 3,300 to 87,000 transistors. These circuits were first simulated using both Irsim and the sequential implementation of Mirsim. Compared to Irsim, the average slowdown for Mirsim was only 2.6 percent, with a maximum slowdown for any circuit less than 6 percent. The current set of partitioning algorithms supported by Mirsim includes an iterative partitioning algorithm (K-FM) and an *acyclic* algorithm (K-AFM), which imposes the constraint that generated partitions do not form cycles. In addition, a graphical facility allows a designer to manually partition the circuit.

Figure 3 summarizes the results of the simulation study for the VLSI circuits, showing for each circuit the best speedup that we obtained using both conservative and optimistic algorithms and the specific partitioning algorithm that yielded the corresponding result. No single synchronization or partitioning algorithm dominated, but acyclic partitions appeared to yield the best performance. Their performance improves as they eliminate circular dependencies in the parallel model, which for conservative algorithms significantly improves the look-ahead properties of the entities, resulting in a dramatic reduction in null messages. For optimistic algorithms, it reduces the number of rollbacks, although the improvements in execution time are not as dramatic.

Where acyclic partitions were infeasible, manual partitioning yielded better performance because the designer could use structural information from the circuits to minimize communication and synchronization overheads.

Figure 3 also shows the speedup predicted by the ISP. In many cases, the performance of the simulator is close to the lower bound predicted by the ISP, which clearly indicates that further improvements in performance are likely to come from better partitioning algorithms or more efficient IPC implementations than from any improvements in the synchronization mechanism itself.

Wireless network models

Wireless, multihop, mobile networks are useful for rapid deployment and dynamic reconfiguration, particularly in applications like battlefield communications and search-and-rescue operations.[7] Parsec has also been used to develop a parallel simulation library, called Glomosim, for simulation of such networks. The library has three primary design goals:

- Capability for simulating very large networks with tens of thousands of mobile nodes.
- Provision of a consistent framework for quantitative comparison of alternative protocol implementations at various layers in the wireless protocol stack and capability to port protocol

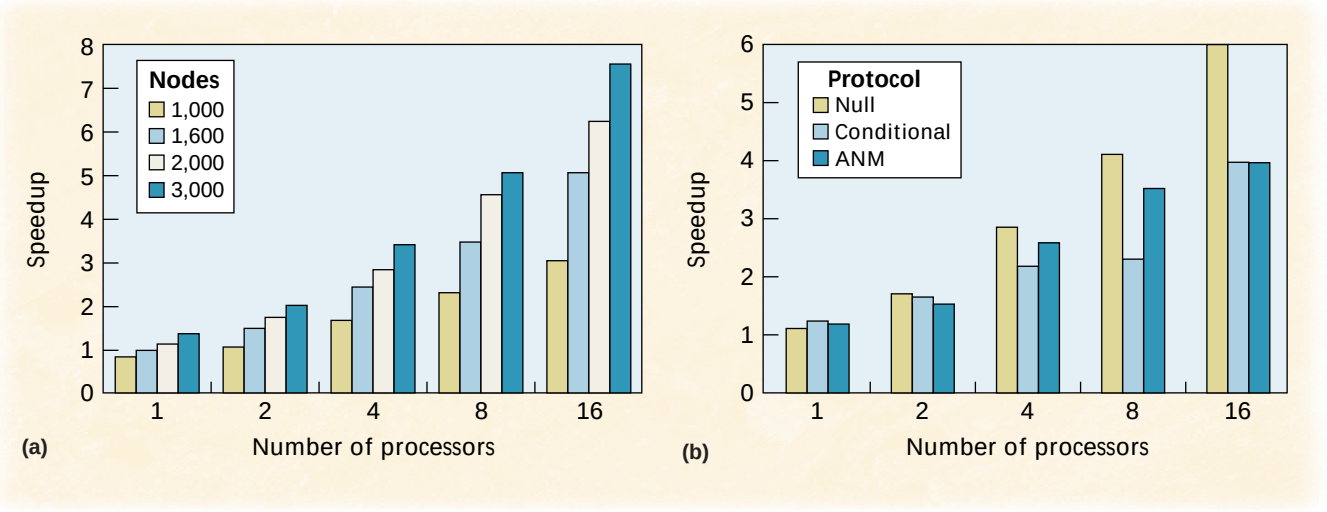

Figure 4. Simulation of Wireless network performance. (a) Speedup using null message protocol as the number of nodes increases. (b) Speedup obtained using three conservative protocols.

models into operational implementations.
- Integration of existing protocol implementations into the simulator.

Glomosim is extensible and composable, having been adapted from the standard ISO/OSI communication protocol stack for wireless networks. The stack has been divided into a set of layers, each with its own API. Models of protocols at one layer interact with models at a lower or higher layer only via these APIs. A number of protocols at several of the stack layers have been implemented. For example, the library provides different channel propagation models, from simple analytical function calls to a computationally intensive statistical impulse response model. Similarly, the MAC layer supports the simple CSMA protocol and the more efficient hidden-terminal collision-avoidance protocols like MACA. The library is continually adding new protocol models.

Our performance study of the Glomosim library simulated a network with up to 3,000 mobile nodes distributed randomly over an 800 meter × 800 meter region. Each node simulates a spread-spectrum radio with a transmission range of 50 meters using the MACA protocol at the MAC layer and a free-space model of the communications channel. The message traffic is generated using a Poisson process with a mean arrival rate of one packet per second for each node. Figure 4a shows the speedup obtained by the simulator using the null message protocol to simulate the network as the number of communicating nodes is increased. As expected, the speedup improves monotonically, with an increase in the network traffic and the number of processors. Figure 4b compares the speedups obtained by the three different conservative protocols supported by Parsec.

Performance prediction of large-scale complex systems using detailed simulation models is a computationally intensive task. Lack of appropriate tools has hindered the widespread use of this technology in domains where it otherwise could be

Parallel and Distributed Simulation

The term *distributed simulation* was originally used to describe the execution of a simulation model on multiple processors, presumably because it was conceived in the context of a network of computers.[1] Today, the term *parallel simulation*, or sometimes *parallel and distributed simulation* (PADS), is used to refer to the execution of a simulation model on multiple processors, regardless of the specific parallel or distributed hardware platform used.

To add further confusion, the term *distributed simulation* has also been used to refer to the interconnection of a number of autonomous simulations, particularly in a military context (for example, distributed interactive simulations, or DIS). The primary distinction between PADS and DIS is that PADS requires that all events in the system be executed in their causal order, where causality is typically determined on the basis of event time stamps, whereas DIS-like simulations do not require this strict level of synchronization among multiple components and can even tolerate message loss.[2]

References

1. K.M. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Trans. Software Eng.*, Sept. 1979, pp. 440-452.
2. R. Fujimoto and R. Weatherly, "Time Management in the DoD High Level Architecture," *Proc. 1996 Workshop Parallel and Distributed Simulation*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 60-67.

used profitably. Our parallel discrete-event simulation environment supports the design and execution of parallel models on shared-memory and message-passing platforms using a variety of synchronization algorithms. The speedup numbers for our case studies provide convincing proof of the potential of parallel model execution for reducing the execution time of detailed simulation models of complex systems.

However, a number of important research issues still need to be addressed. One is finding a design for partitioning algorithms that not only minimizes the communication costs among the partitions but also reduces the overheads of the parallel simulation algorithms. Another challenging task is finding a useful way to make parallel simulation technology accessible to the end user through the design of domain-specific libraries of parallelized models. Solving these problems will make parallel simulation as accessible to end users as sequential simulation is today. ❖

Acknowledgment

This work was partially supported by the US Defense Advanced Research Projects Agency under contracts J-FBI-93-112, DABT-63-94-C-0080, and DAAB07-97-C-D321.

References

1. E.A. Brewer et al., *Proteus: A High Performance Parallel Architecture Simulator*, Tech. Report MIT/LCS/ TR-516, Massachusetts Institute of Technology, Cambridge, Mass., 1991.
2. J. Misra, "Distributed Discrete-Event Simulation," *ACM Computing Surveys*, Mar. 1986, pp. 39-65.
3. D. Jefferson, "Virtual Time," *ACM TOPLAS*, July 1985, pp. 404-425.
4. V. Jha and R. Bagrodia, "A Unified Framework for Conservative and Optimistic Distributed Simulation," *Proc. 1994 Workshop on Parallel and Distributed Simulation*, Society for Computer Simulation, San Diego, Calif., 1994, pp. 12-19.
5. R. Bagrodia, "Parallel Languages for Discrete-Event Simulation Models," *IEEE Computational Science and Engineering*, Apr.-June 1998, pp. 27-38.
6. R. Fujimoto, "Parallel Discrete Event Simulation," *Comm. ACM*, Oct. 1990, pp. 30-53.
7. A. Alwan et al., "Adaptive Mobile Multimedia Networks," *IEEE Personal Comm.*, Apr. 1996, pp. 7-22.
8. K.M. Chandy and R. Sherman, "Space-Time and Simulation," *Proc. Distributed Simulation Conf.*, Society for Computer Simulation, San Diego, Calif., 1989, pp. 33-57.
9. R. Bagrodia, K.M. Chandy, and W.T. Liao, "A Unifying Framework for Distributed Simulation," *ACM Trans. Modeling and Computer Simulations*, Oct. 1991, pp. 348-385.
10. R. Bagrodia, V. Jha, and M. Takai, *Performance Evaluation of Conservative Algorithms in Parallel Simulation Languages*, Tech. Report CSD 980026, Computer Science Dept., UCLA, 1998.
11. Y. Chen and R. Bagrodia, "Shared Memory Implementation of a Parallel Switch-Level Circuit Simulator," *Proc. 12th Workshop on Parallel and Distributed Simulations (PADS) 98*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 134-141.

Rajive Bagrodia is an associate professor of computer science at UCLA. His research interests include parallel simulation, parallel languages and compilers, and nomadic computing. He received a BTech in electrical engineering from the Indian Institute of Technology, Bombay, and a PhD in computer science from the University of Texas, Austin.

Richard Meyer is a PhD candidate in computer science at UCLA. His research interest is parallel simulation. He received a BS in math and computer science and an MS in computer science, both from UCLA.

Mineo Takai is a post-doctoral scholar at UCLA. His research interests include synchronous algorithms in conservative parallel simulation. He received a BS, an MS, and a PhD, all in electrical engineering and all from Waseda University in Japan. He is a member of SCS.

Yu-an Chen recently received a PhD in computer science from UCLA. His research interests include parallel simulation and VLSI/CAD. He received a BS and an MS in computer science and information engineering from the National Taiwan University.

Xiang Zeng is an MS candidate in computer science at UCLA. Her research interest is scalable network simulation. She received a BE in computer science from the University of Science and Technology of China.

Jay Martin is a PhD candidate in computer science at UCLA. He is currently researching parallel simulation of parallel databases. He received a BS in computer science and a BS in math from the University of California, Irvine, and an MS in computer science from UCLA.

Ha Yoon Song is a PhD candidate in computer science at UCLA. His research interests are system performance evaluation, parallel and distributed simulation, distributed computing, and VLSI/CAD. He received a BS and an MS in computer science from Seoul National University, Korea.

Contact the authors at the Computer Science Department, UCLA, Los Angeles, CA 90095-1596.