PARALLEL LANGUAGES FOR DISCRETE-EVENT SIMULATION MODELS

Rajive L. Bagrodia
Computer Science Department
University of California at Los Angeles
Los Angeles, CA 90095
rajive@cs.ucla.edu

Version #2, Submitted March 26th, 1998

ABSTRACT

Interest in exploiting parallelism in the execution of discrete-event simulation models has increased in a number of application areas ranging from VLSI designs and communication network models, to theater level battle models. A number of software systems have been designed to support the execution of these models on diverse parallel architectures. This paper discusses some of the primary design choices and tradeoffs that have been made in the design of parallel simulation languages and briefly describes one of the systems, PARSEC or PARallel Simulation Environment for Complex systems, developed at UCLA.

1. Introduction

Simulations are among the most significant, if not the key consumers of computer cycles. For instance, estimates have placed the aggregate compute cycles dedicated to digital simulation in the EDA industry at over 90% of the total EDA compute cycles. The complexity of circuit simulations is such that it routinely requires a few days to a few weeks to simulate specific instances of a processor level design on contemporary desktop workstations. As another example, the simulation of wireless networks with thousands of mobile nodes communicating in a realistic environment subject to interference and signal fading effects can easily take days of simulation time using state of the art uniprocessor workstations. Simulations have traditionally 'pushed the envelope' in terms of their ability to always require at least an order of magnitude more computation power than is made available by any generation of machines! Given the importance of simulations to computing, it is hardly surprising that research in simulation languages has had a significant impact on the evolution of programming languages [Nance 1993]. The limitations of sequential model execution has led to growing interest in the use parallel execution for simulating large scale systems, which is the primary focus of his paper.

Continuous vs. Discrete Models: Simulation models are primarily classified to be continuous or discrete. Computer system models are typically *discrete*, where the state of the system being modeled is assumed to change only at discrete instants of time. In contrast, models of physical systems like the global climate or the model of a chemical process are typically referred to as a continuous model, as the state of the system is defined as a continuous function of time. The focus of this paper is restricted to discrete event models.

Selecting a Language: In developing a (sequential or parallel) simulation model, among the first choice that must be made by the analyst is the selection of a language in which to program the model [Law & Kelton 1991]. This can range from a general purpose language like C or Java; a simulation language like SIMULA [Franta 1977], CSIM [Schwetman 1986], SIMSCRIPT II.5 [Law & Larney 1984] or Maisie [Bagrodia & Liao 1994], or a simulation library like GASP [Pritsker 1974] or SPEEDES [Steinman 1992] written in a general purpose programming language. The primary advantage of simulation languages or libraries is that they provide the programmer with built-in facilities for the common simulation tasks like event scheduling, time advance, random number generation, statistical analyses and report generation, and a library of commonly used objects, such as server objects, job objects, and queue objects. Some of the most common facilities are discussed in Section 3. Simulation languages also provide a framework or world view that typically makes it easier for the analyst to design a model. Of course, it is also the case that a given simulation language may force the analyst to use a specific simulation world view, whereas a general purpose language may offer more flexibility. Commonly used simulation world views are discussed in Section 3.2.

The primary drawback in using a simulation language is the effort needed to learn a new programming environment, although this is somewhat mitigated by simulation languages that are enhancements of familiar general purpose languages. Another pragmatic benefit of general purpose languages is that they are widely available, have been ported to most computers, and come with better program development environments than is the case with the typical simulation language.

In the case of parallel simulations, the complexity of implementing the time advance and related simulation mechanism is typically enormous and there is relatively little benefit to an analyst in developing parallel simulations in a general purpose parallel language. Issues that pertain to parallel programming and parallel simulations are addressed in Sections 3.3-5.

Parallel and Distributed Simulation: There are two primary motivations for using parallel simulation: first, is the desire to reduce the turnaround time for large models by exploiting multiple computers which is similar to the traditional use of parallel computers to reduce the execution time for any CPU intensive computation. This area of work is commonly referred to as parallel discrete-event simulation or PDES. The second motivation is *interoperability*: the goal is to connect multiple simulators each of which may be simulating a different component of a complex system and may be possibly running on geographically distributed machines. The recent High Level Architecture (or HLA) [Defense 1997] initiative being pursued by the US Department of Defense Modeling and Simulation Office reflects the latter motivation. The term 'parallel discrete-event simulation' (or PDES) is typically used to refer to the work in the first area whereas the term 'distributed simulation' is more commonly used to refer to work in the latter area. The primary distinction between parallel and distributed simulations (as these terms are used today) is that all events in a PDES model must be synchronized using logical time whereas a distributed simulation as proposed in HLA supports multiple synchronization mechanisms, including the use of physical time to synchronize events across the multiple simulators. Note that one of the simulators in a distributed simulation may be a PDES model. The focus of this paper is on PDES models.

PDES Tools: Available tools for parallel simulation include operating systems, simulation languages, and domain-specific simulators [Bagrodia 1994]. The primary advantage of the OS-based approach is that it provides complete flexibility in the choice of a programming language, and that much of the scheduling, memory management, and IPC functionality provided by the OS need not be duplicated by the parallel simulator. The major drawback of this approach is the sheer complexity of implementing an operating system while tracking the multiple parameters that eventually determine its performance.

Parallel simulation languages and libraries (PSLs) integrate a set of simulation primitives (similar to those provided by sequential simulation languages) with a set of parallel programming primitives. The primary advantage of PSLs over the OS approach is that the former can provide a more abstract interface to the programmer. At the end of the spectrum are domain-specific simulators that are designed to serve a specific application area like network models. The parallel simulators may be developed using general purpose languages or by writing domain-specific libraries using some simulation language. The obvious advantage of this approach is that the simulator can exploit specific application characteristics to reduce overheads of parallel model execution and present a domain-specific interface to the programmer to facilitate model description. Three domains which have attracted significant research efforts include VLSI circuit simulators (e.g., PLDVISIM [Briner et al 1991] and MIRSIM [Chen et al 1997]), parallel program simulators (e.g., LAPSE [Dickens et al 1996], Wisconsin Wind Tunnel [Reinhardt et al 1993] and MPI-SIM [Prakash 1996]), and communication network models (e.g., GloMoSim [Zeng et al 1998] and SimKit [Gomes et al 1995]). In each of the preceding areas, performance improvements of at least one order of magnitude have been obtained for realistic applications.

The primary focus of this paper is on PDES languages and libraries. The next section gives an overview of the protocols for parallel synchronization. Section 3 describes the primary features found in PSLs and considers alternative approaches and design choices. Section 4 describes one parallel simulation language, PARSEC, and presents results from a case study on the use of PARSEC in the design of a parallel model library to simulate wireless protocols. Section 5 concludes with a brief discussion of trends and important research directions in PSLs.

2. Parallel Simulation Protocols

In a parallel discrete-event simulation, the system being simulated is modeled by a number of logical processes (LP) that interact by exchanging timestamped event messages. Each LP must eventually process incoming messages in their global timestamp order. Enforcing this requirement, referred to as the causality constraint, is the central problem in efficient execution of parallel simulations. We briefly describe the three primary approaches that have been suggested to solve the synchronization problem: conservative, optimistic, and adaptive or hybrid approaches.

Conservative algorithms [Misra 1986] do not permit any causality error: each object in the simulation processes an incoming message only when the underlying synchronization algorithm can guarantee that it will not subsequently receive a message with a smaller timestamp. This constraint may introduce deadlocks, which are typically avoided by using *null* messages. A null message is a timestamped signal sent by an LP to indicate to other LPs a lower bound on the timestamp of its future messages. If an LP sends a null message with timestamp T at simulation time t, T>=t, we say that the LP has a lookahead of (T-t). In general, models that exhibit good lookahead have demonstrated the best performance with conservative protocols. In addition to lookahead, the communication topology of a model also impacts its performance; models that use a sparse communication topology typically exhibit better performance. Conservative implementations are generally simple to develop and basically need support to implement mechanisms for *null message transmission* and *lookahead specification*. A variety of optimizations have been suggested to reduce null message traffic. A common variation is the use of demand driven null messages, where null messages are sent only when explicitly requested by an LP. Another alternative is to use conditional events, where a global computation is used to identify the earliest global event and no null messages are necessary. This scheme can be effective for models with poor lookahead. Yet a third alternative, called the accelerated null message protocol uses both null messages and conditional events to allow an LP to deduce when a local event can be executed without violating the causality constraint. A good overview of the performance of these three algorithms is in [Jha & Bagrodia 1993].

In optimistic protocols, an LP is allowed to process events in any order; however, the underlying synchronization protocol must detect and correct violations of the causality constraint. The simplest mechanism for this is to have each LP periodically save (or checkpoint) its state. Subsequently, if it is discovered that the LP processed messages in an incorrect order, it can be rolled back to an appropriate checkpointed state, following which the events are processed in their correct order. The rollback may also require that the LP unsend or cancel the messages that it had itself sent to other LPs in the system. An optimistic algorithm is also required to periodically compute a lower bound on the timestamp of the earliest global event, also called the Global Virtual Time or GVT. As the model is guaranteed to not contain any events with a timestamp smaller than GVT, all but one checkpoints timestamped earlier than GVT can be reclaimed. Thus the primary facilities needed to implement optimistic methods include *checkpointing*, *message cancellation*, *rollback and recomputation*, and *GVT computation*. Considerable research has gone into architecture- and application-specific optimizations for each of the preceding activities [Fujimoto 1991].

Recently, adaptive or hybrid protocols have been suggested that allow each LP in a model to individually select either the conservative or the optimistic execution mode and possibly change its mode dynamically [Jha & Bagrodia 1994].

The next section describes languages that have been used to develop simulation models for parallel execution. We identify some common themes in their design, describe a few primary characteristics, and present sample performance numbers where available.

3. Design of Parallel Simulation Languages

3.1 Languages and Libraries

A number of simulation software systems have been developed to support parallel execution of DES models. The primary approaches used to design general purpose parallel simulation software include the following:

- Library based approaches represented by systems like GTW [Das et al 1994], UPS [Nicol & Heidelberger 1995], COMPOSE [Martin & Bagrodia 1995], and SPEEDES [Steinman 1992]. These systems typically provide simulation and parallelism capabilities via calls to libraries implemented in standard sequential languages like C or C++. Their primary advantage is that the user does not have to learn a new language. A major drawback is that because no translator is used, the library routines must provide a less abstract interface and do considerably less error checking than is typically possible with a high-level simulation language.

- Enhancement of sequential (simulation) languages with primitives for parallel simulation; examples include Sim++ [Baezner et al 1990], APOSTLE [Wonnacott & Bruce 1996], Maisie [Bagrodia & Liao 1994], MOOSE [Waldorf & Bagrodia 1994]—an object-oriented extension of Maisie, and PARSEC [Bagrodia et al 1998]—a next generation implementation of Maisie. The ability to use a translator allows languages to provide a more succinct and 'natural' interface for the programmer as compared with simulation libraries. It is also possible to implement a number of optimizations like automatic granularity control as in APOSTLE and automatic reduction of rollback distances as in Maisie.

A parallel simulation language (PSL) must provide a set of simulation primitives (similar to those provided by simulation languages), parallel programming primitives (similar to those found in many parallel programming languages and process-oriented simulation languages), and perhaps constructs to optimize the execution of a simulation model with a specific parallel simulation algorithm. We consider each of these issues in turn.

3.2 Simulation World Views

Traditionally, simulation languages have used one of the following three world views for the design of discrete-event simulation models: event scheduling, activity scanning, or process interaction. Each of these paradigms has its proponents, but all three paradigms are equally powerful in a representational sense: *i.e.* any simulation model that can be designed using one world view can also be designed using another. Most simulation languages adopt one of the three world views although it may be possible, with additional effort, to program models using multiple world views in a given language. As relatively few PSLs use the activity scanning world view, it will not be discussed further in this paper.

The primary difference between the event-scheduling and the process-interaction paradigms is that the former views the execution of a model explicitly as a sequence of events, where an event may be represented by a message (as in GTW or TWOS) or as an object as in SPEEDES. Each event specifies a destination LP and a timestamp. The LP defines an event handler routine that processes the event at the indicated (simulation) time by executing code that may change the program state and/or schedule future events. In a PSL, execution of an event may involve additional activities that are related to the specific simulation algorithm being used. For conservative algorithms, the LP may modify the value of its lookahead variable and perhaps change the communication topology, whereas for optimistic algorithms, it may checkpoint its state, perhaps initiate a rollback, or reclaim memory used for previously checkpointed states that are no longer needed.

In a process-interaction world view, the events are not identified explicitly in the model. Rather the model is described via the actions executed by a set of interacting objects or co-routine style processes. Each LP in the simulation is an instance of a specific process (or class) type that describes the attributes and the time-ordered sequence of actions to be executed by all objects of that type. Events are extracted by the simulation kernel from the interactions among the LPs, where the interactions may be expressed by the programmer using method invocations (as in Sim++ and ModSim II), message exchanges (as in Maisie), or in the (very) rare case by accessing shared variables. As two or more LPs may simultaneously try and acquire exclusive access to a given resource (e.g., a server), appropriate constructs must be defined to synchronize the LPs.

The reader familiar with parallel programming will note the similarity of vocabulary and ideas between the process-interaction approach to simulation and parallel programming. Thus it should not be surprising to discover that most parallel simulation languages adopt the process-interaction world view. However unlike sequential simulation languages that have adopted this world view (e.g., SIMULA), PSLs do not give an LP carte blanche access to the data space of all other simulation objects in the program. In the remainder of this paper, we restrict ourselves to languages that adopt the process interaction world view for simulation. Good discussions of the simulation world views may be found in [Franta 1977, Evans 1988]

Languages that adopt the process-interaction world view provide primitives for the creation and termination of processes, for communication and synchronization among processes (referred to as IPC or interprocess communication primitives), and for scheduling events and processes. The primary overlap between PSLs and general purpose parallel languages is in the process creation, termination, and IPC constructs, which are discussed in the next section. Section 3.4 discusses scheduling constructs.

3.3 Parallel Programming Primitives

In the domain of task parallel programming languages, the common programming models include the shared nothing or message-passing model where processes do not share any data; the shared everything or the shared-memory model where all data can be referenced by every process; and the shared-something models where some data can be shared among processes and the rest is private. Although sequential simulation languages based on the process-interaction approach routinely use the shared everything model (e.g. SIMULA), PSLs have primarily used the shared nothing model with some restricted form of shared state.

In the shared nothing model, a PSL provides constructs for the (static or dynamic) creation and termination of processes (also called *entities* or *LPs*), and primitives for interprocess communication. The IPC mechanisms typically use asynchronous non-blocking send primitives together with a blocking receive. Some systems support the notion of message types and provide a sophisticated capability to selectively accept messages. For instance Maisie uses *guards* and APOSTLE uses *path expressions* to impose some concurrency control by selectively accepting messages in an order that is perhaps different from the order in which the messages are received by the LP. Libraries like GTW provide a minimalist communication facility with no notion of message types or selective receives. In most PSLs, LPs directly address each other; APOSTLE supports direct addressing and the use of *ports* for communication among LPs. The primary advantage of ports is their support for dynamic reconfiguration of models which comes with the minor inconvenience of having to connect the destination LPs to appropriate ports for communication to be initiated. It is unclear that ports have a significant impact on reusability; variables can be declared as place holders to store the identifier for communicating partners of an LP. The actual identifiers for the communicating partners can be subsequently passed as arguments to the library object at run time.

Specification of the smallest unit of concurrency is another important characteristic of a parallel language. Most parallel languages are multi-threaded in that they allow multiple entities to be executed concurrently on a processor. Multi-threading is clearly desirable from a programmer's perspective as it allows a program to be decomposed into an arbitrary number of components based on the dictates of modularity rather than the hardware configuration on which the program will be executed. Dynamic creation and destruction of threads is also desirable from the same perspective. Multi-threading can also use latency hiding to increase *potential* parallelism in the program: when a given thread on a processor is blocked for any reason, another thread may be executed. However, as is well known, multi-threading may also worsen performance due to the increased scheduling and context-switching overheads.

Almost all PSLs support multi-threading. Given that a PSL supports multi-programmed LPs, the scheduling algorithm can have a significant impact on performance. Typically, LPs are scheduled in strict order of their event or message timestamps. In other words, assume that given three multi-programmed LPs, *l1*, *l2*, and *l3* and events *e1, e2 … e6* with respective timestamps 1,2,..6 such that *e1* and *e4* are scheduled for *l1*, *e2* and *e5* on *l2* and *e3* and *e6* on *l3*. In this case, the execution of the model using the traditional sequential algorithm requires a context switch and a separate dequeue operation for the execution of each event. In contrast, if a conservative algorithm is used to schedule multi-programmed LPs, then it might be able to deduce that both events *e1* and *e4* are safe (meaning that there is no causality violation in executing e4 before events e2 and e3 which have earlier timestamps) and thus reduce the context switching overheads. As has been observed by many PDES researchers [e.g., Wagner 1991], execution of the model on a uniprocessor using conservative algorithms is sometimes *faster* than executing it with the traditional sequential algorithm; in some cases we have noticed improvements of as much as 30% over the sequential time with Maisie models.

Given that a PSL supports multi-threading, a related issue is whether each thread is considered to be a simulation object or LP or whether threads may be coalesced (explicitly or implicitly) into fewer LPs. The primary advantage of coalescing threads is that it tends to increase the event granularity and reduce context-switching overheads. Event granularity measures the average amount of *local* computation performed by an LP for an event. Consider a switch-level circuit simulation model that consists of a very large number of very simple switching devices. Each device has a given set of input and output connections and devices are connected together by linking the outputs of one device to the inputs of one or more other devices. In the extreme case, each device could be an LP; clearly the small event granularities and large number of context-switches will overwhelm its sequential and parallel execution times. In such cases, the programmer typically decides that it is not efficient for each switch to be a separate object. Instead, the model is first decomposed into a number of partitions, each of which is an LP. The 'events' that occur at a switch within an LP are handled locally and do not have to incur the overheads of insertion, deletion, and synchronization associated with the external event-list. The number of partitions (and hence LPs) may be greater than the number of available processors. The optimal number of partitions appears to depend significantly on the architecture and synchronization algorithm that is being used (and in some cases on the specific circuit that is simulated); thus the ability of a PSL to support multi-programming appears to be crucial. For homogeneous systems like circuit simulations, automated partitioning algorithms exist to decompose a circuit into a specified number of partitions, using appropriate heuristics to minimize the overheads due to communication and synchronization among the partitions that model the corresponding sub-circuits. Notice that in this approach, each LP in the model is also an object in the program.

An alternative approach adopted by PSLs like APOSTLE is to allow a simulation object (or LP) to be a coalition of multiple objects in the model. Events on the multiple objects that are thus coalesced together need not incur the synchronization and scheduling overheads of events on objects that belong to different LPs. Although this approach can be used for homogeneous systems like VLSI models, it is particularly useful for the simulation of heterogeneous systems that contain many types of objects with very different behaviors. Significant improvements have been reported in the execution time of a set of models with the use of granularity control to reduce the number of LPs [Wonnacott & Bruce 1996]. With both manual and automated granularity control mechanisms, the optimal number of simulation objects was found to depend on the parallel architecture and the application itself. However, the improvements in execution time that were reported in both these cases, are consistent with the reduction in execution time that was noticed with the use of conservative scheduling algorithms for sequential model execution. In the latter case, note that the improvements were obtained not from the reduction in the number of LPs, but from more efficient *scheduling* of multi-programmed LPs.

The object-oriented paradigm that is commonly used by parallel programming languages has also been adopted by many PSLs. Examples of PSLs that have used the object-oriented model include

ModSim II [Rich & Michelsen 1991], MOOSE, COMPOSE [Martin & Bagrodia 1995], Sim++, and APOSTLE. In this model, an LP is programmed as an object with an associated set of methods. An object can asynchronously invoke a method defined in another object or use messages for this purpose. In the latter case, bindings are typically defined between a message type and a specific method as in COMPOSE. In this case, the invoked methods cannot return any values to the calling LP. (Remote) method invocation may be synchronous or asynchronous. In the synchronous case, the semantics are similar to those of a remote procedure call: the calling LP is blocked until the method has been executed in the remote object, and return values, if any, have been collected by the caller. In the asynchronous case, the remote methods typically cannot have output parameters that return values and communication is one-way as with traditional message-based languages. Languages like MOSS and ModSim II support both types of method invocations. APOSTLE uses the concept of *promises* with asynchronous method invocations to synchronize with the completion of the remotely invoked method. This allows an APOSTLE object to support output parameters with asynchronous method invocations.

A variety of parallel architectures ranging from SIMD-like machines (e.g., Connection Machine CM-2), shared memory architectures, distributed memory architectures, and networks of workstations have been used to execute parallel programs. Although each architecture typically provides a proprietary parallel programming kernel, some systems (e.g. the MPI library) have been implemented across a variety of architectures. Typically, a PSL has been implemented on only *one* architecture usually using only one class of synchronization algorithms. PSL implementations on shared memory architectures are perhaps more common, primarily due to the wider availability of 4 or 8-way symmetric multiprocessors.

3.4 Scheduling Primitives

In addition to the primitives for LP (or process) creation, termination, and interactions, simulation languages also provide a set of scheduling constructs to change the status of an LP. As discussed in the previous section, most simulation languages are multi-threaded which implies that the execution of multiple LPs in a simulation must be interleaved. Using SIMULA-like terms, an LP is *active* if it is currently being executed by the processor; *suspended* if it is scheduled for activation at a future (simulation) time; and *passive* if the LP has not been terminated and is neither active or suspended. When no LP is active, the scheduler selects the LP with the earliest activation time for execution and advances the simulation clock to this value. The selected LP transitions to the active state. (If all LPs are passive, the simulation is terminated.) Once an LP is active, most simulation languages allow it to execute until it autonomously terminates, suspends, or becomes passive. We next discuss constructs to suspend or passivate an LP.

The two most common scheduling constructs supported by discrete event simulation languages include the *hold* and *wait until* primitives [Franta 1977]. The former allows an LP to suspend itself for a pre-determined duration in simulation time. This statement is used to simulate the passage of time in a physical system. For instance, on the arrival of a job, an idle FIFO server may execute a *hold(delay)* statement. Execution of such a statement suspends the LP and reschedules it for activation after *delay* time units. Control is once again passed to the scheduler which proceeds as explained before. Almost all PSLs support a hold primitive that is similar in most respects to the preceding construct to suspend and reschedule the activation of an active LP.

The *wait until* primitive is a generalization of the hold construct and allows relatively complex synchronization conditions to be implemented among LPs in a model. In general, this construct allows an LP to be suspended for a given duration of simulation time or until a specified Boolean condition (also referred to as a guard or resume condition) is satisfied. As proposed in SIMULA, an LP becomes passive when it executes a wait statement and must be explicitly activated by another LP. The semantics of the wait until and activate constructs in SIMULA are similar in this respect to the wait() and resume() operations proposed for monitors.

Most PSLs support some form of the wait until primitive although many (e.g., Sim++, MOOSE and PARSEC) integrate it into their message receive or call acceptance mechanisms. PSLs differ in the specific part of the system state that may be used to specify the resume condition. For instance, Modsim II provides a *WAIT* statement to allow an object to suspend itself until a specified method in another object has been executed or a specified interval of simulation time has elapsed (whichever occurs first). Similarly PARSEC (like its predecessor Maisie) provides a receive statement that can suspend an LP until it receives a message that satisfies a specific guard or a specified interval of simulation time has elapsed (whichever occurs first); the guard can reference the local state of the receiving LP or the message parameters. Note that most PSLs do not require an LP to explicitly activate another LP that is blocked by a wait statement; rather the 'resume' condition is tested locally whenever a new message (event) is received or when the LP executes another event.

Simulation languages also typically provide routines to sample the simulation clock and a standard set of functions for random number generation and statistics collection. PSLs may need to modify the library routines provided by DES languages. For instance, optimistic PSLs must ensure that library functions do not use permanent variables or heap storage, as that will make it difficult to checkpoint or rollback the computation.

3.5 Support for Parallel Simulation

A PSL must implement one or more of the parallel simulation protocols to ensure that all events in the parallel model are eventually executed in the global order of their time stamps. Some systems (e.g. TWOS) attempt to keep the underlying algorithm completely transparent from the programmer, whereas others (e.g. SPEEDES) require that functions needed by the underlying synchronization algorithm (e.g. state saving or rollback) be programmed explicitly in the model. Whereas the latter approach tends to produce the most efficient implementations, it also makes the design of a correct model significantly harder. In the case of optimistic systems though, the sheer complexity of tasks such as checkpointing, detecting and initiating rollbacks, canceling remote messages, etc., is such that without significant support from the run-time system, the code will be complex and error prone. As there are relatively few restrictions on the types of models than can be simulated using optimistic algorithms, these tasks can also be more easily made transparent with some performance degradation. With the exception of SPEEDES, almost all other PSLs that support the optimistic algorithms hide the algorithmic details from the programmer. Efficient implementation of optimistic simulators is complex and involves numerous tradeoffs. It is not unusual to find that for a given application, multiple optimistic simulators display dramatic performance differences.

Most PSLs implement a number of optimizations to improve parallel performance. In some case, the optimizations are transparent to the programmers, whereas in others they can be invoked via a set of well-defined interfaces. As these optimizations do not affect the correctness of the model, but only its performance, they can be used after the model has been defined. As an example of transparent optimizations, consider the use of adaptive checkpointing. As an alternative to checkpointing the state of an LP before processing every event, this technique can be used to track the frequency and duration of roll backs in an LP to vary the frequency with which its state is checkpointed. As another example, consider the handling of rollbacks in Maisie: in a typical optimistic model, whenever an LP receives a straggler message, it must be rolled back to a state preceding the timestamp of the straggler message. However as explained in the previous section, a Maisie LP may use guards to buffer messages that are not immediately accepted by the LP. Thus, an LP need not be rolled back whenever it *receives* a straggler message; rather the rollback is initiated only if the timestamp of the straggler message is less than that of a message that has been *accepted* by the LP. Languages like APOSTLE allow the programmer to vary the granularity of a model by the simple mechanism of mapping multiple program objects onto a single LP. As an example of conservative optimizations, PARSEC provides primitives that can be used by an LP to dynamically update its communication topology and hence improve its lookahead properties. Both PARSEC and UPS provide constructs that may be used by an LP to communicate its (dynamic) lookahead values to the simulation kernel (which may be better than the minimum lookahead necessary to prevent deadlocks with conservative models).

4. PARSEC

PARSEC, a Parallel Simulation Environment for Complex systems is a parallel simulation language that uses the process-interaction approach to discrete event simulation. PARSEC has been implemented on both distributed and shared memory platforms, and supports both conservative and optimistic parallel simulation protocols. Supported conservative protocols include the null message based schemes as well as protocols based on the conditional event algorithm, whose performance has not been studied widely. The optimistic protocol is based on the space-time paradigm, an early implementation of which is described in [Bagrodia et al 1991]. PARSEC also supports a new protocol called the Ideal Simulation Protocol or ISP which can be used by an analyst to compute a tight lower bound on the execution time of a parallel simulation model on a given architecture [Jha & Bagrodia 1996]. Although there are other tools that have been implemented on multiple architectures or support multiple simulation protocols, to the best of our knowledge, no existing simulation environment provides such a broad coverage of parallel architectures and protocols. PARSEC has been used to simulate a diverse set of applications including VLSI circuits, parallel programs, parallel IO systems, interconnection networks, electronic LANS, ATM networks, aeronautical communication networks, and wireless communication protocols.

A PARSEC program consists of a set of modules, where each module is an entity or C function. Each entity is an LP that models a corresponding physical process. Entities can be created and destroyed dynamically and multiple entities can be mapped to a single processor. For parallel execution, each entity is explicitly mapped to a processor when it is created and cannot subsequently be migrated to a different processor. Events are modeled by message communications among the corresponding entities. Each entity is associated with a unique message buffer and asynchronous send and receive primitives are provided to deposit and remove messages from the buffer. An entity in a PARSEC model simulates the local actions of a physical object using C code and describes the interactions of the corresponding physical object via timestamped messages. It supports both the *hold* and *wait until* primitives introduced earlier. The PARSEC implementation of a wait until statement allows many complex enabling conditions to be expressed directly, without requiring the programmer to describe the buffering explicitly. The resume condition can be a complex expression and may reference (though not modify) state variables of the entity and the message buffer. We present two examples to illustrate PARSEC constructs for parallel simulation.

The first example illustrates the asynchronous message passing constructs used by PARSEC using an entity called *manager* to simulate a resource manager. The first three lines declare the types of messages that may be sent or received by the entity: type *request* that is used by another entity (with id *his_id*) to request a certain number of units (*num*), type *release* to return resources to the pool, and type *done* to inform the requesting unit that the requested number of units are available. The body of the entity is basically an infinite loop with a receive statement to process the two message types (*request* and *release*) that may be received by the entity. We note that the request message includes a Boolean expression or *guard*. The guard has two expressions: the first ensures that a given request is accepted by the entity only if the required number of units are currently available with the manager. The second expression ensures that if the buffer contains both *request* and *release* messages, *release* messages are accepted by the manager first.

```
Message  Request {int num; ename his_id;};
Message  Release {int num;};
Message  Done {};
Entity Manager (int maxa)
{     int units=maxa;
      for (;;)
      receive (Request req) when ((req.num<=units) && qempty(Release))
      {     units -= req.num;
            send Done to req.his_ id;
      }
      or receive (Release rel)
            units += rel.num;
 }
```

Resource Manager

The next code fragment illustrates the simulation constructs supported by PARSEC. The fragment simulates the transmission of a data packet by an entity to another entity with id *next-hop*. The first statement sends a message of type *data-packet*. The next statement waits to receive either a message of type *ack* or a message of type *nack* from the recipient; the incoming message is copied in the corresponding variable. If either of these messages is received, the entity will execute function *process_ack* or *process_nack* respectively. The hold statements in the respective code blocks suspend the entity (in simulation time) for a duration needed to process the corresponding message in the physical system. The entity also schedules a timeout message to itself at a time *round-trip-delay-time* units of simulation time in the future. If an *ack* or *nack* message is received by the entity within the specified duration, the timeout event is automatically canceled; otherwise a timeout message is received by the entity which then executes procedure *resend_data_packet*. The definition of the various message types have been omitted for brevity.

```
send data-packet{data} to next-hop;
receive {     (ack a)
    {
      hold(ack-time);      /* suspend for time to process an ack   */
      process_ack_received(a);         /*  in the physical system  */
     }
    or (nack n) {
    hold(ack-time);        /* suspend for time to process a nack   */
    process_nack_received(n);          /*  in the physical system  */
}
    or timeout after (round-trip-delay-time) {
      resend_data_packet(data);
     }
   }
```

Data Transmitter

Every PARSEC program must include an entity called *driver*. Execution of a PARSEC program is initiated by executing the first statement in the body of entity driver. As mentioned earlier, PARSEC supports a number of parallel simulation protocols and architectures. The specific protocol and architecture to be used in a given execution of a model is specified as a command line option. A complete description of the language is available in the Parsec User Manual [UCLA 1998], and a report on the parallel performance of the simulator with a variety of applications has been summarized in [Bagrodia et al 1998].

5. Conclusion

Although research in the area of parallel simulation has demonstrated significant potential for improvement in the execution time of models in many application domains, it is widely accepted that PDES tools have penetrated the commercial simulation market in limited ways [Fujimoto 1993]. A major impediment to this has been the lack of integration of the PDES tools into the commercial simulation work flow. Given that there are *many* applications, for which parallel execution does not *a priori* guarantee an improvement in performance, and given that parallel programs in general—and parallel simulation models in particular, are significantly harder to develop than functionally equivalent sequential programs, for widespread acceptance of this technology it is necessary that the simulation tools be designed with the aim of making an iterative transition to parallel execution. It is also important that parallel simulation languages not be restricted to a single architecture or synchronization protocol but that they cleanly separate the issues in the design of a discrete-event simulation program from its efficient implementation on a given (sequential or parallel) architecture.

This paper presented an overview of parallel simulation languages (PSLs). A PSL must provide a set of simulation constructs similar to those provided by simulation languages, parallel programming primitives similar to those found in many parallel programming languages and process-oriented simulation languages, and constructs to optimize the execution of a simulation model with a specific parallel simulation algorithm. In each of the preceding areas, this paper identified some common themes in their design, described a few primary characteristics, and presented a number of design trade-offs that have been made by specific PSLs.

The area of parallel simulation languages and environments is still in its infancy with much of the research in this area beginning in the late eighties. Much remains to be done in the design of general purpose parallel simulation environments, in their application to specific domains, in the identification of application characteristics that may be exploited to reduce the overheads of specific synchronization protocols, and in their integration in the sequential discrete-event simulation workflow. However, as described, multiple efforts are underway both within universities and companies to develop languages to address many of these challenging issues.

6. References

D. Baezner, G. Lomow, and B. Unger, “Sim++: The transition to distributed simulation,” *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, San Diego, California, January 1990, pp. 211-218.

R.. Bagrodia, “Language support for parallel discrete event simulation,” *Proceedings of the Winter Simulation Conference*, December 1994, pp. 1324-1331.

R. Bagrodia, K. Chandy, and W-T. Liao, “A unifying framework for distributed simulations.” *ACM Transactions on Modeling and Computer Simulation*, Vol. 1(4), October 1991, pp. 348-385.

R. Bagrodia and W. Liao, “Maisie: A language for the design of efficient discrete-event simulations,” *IEEE Transactions on Software Engineering*, Vol. 20(4), April 1994, pp. 225-238.

R. Bagrodia, R. Meyer, B. Park, H. Song, Y. Chen, X. Zeng, J. Martin, M. Takai; “PARSEC: A parallel simulation environment for complex systems,” *IEEE Computer Magazine*, 1998 (to appear).

J. Briner, J. Elis, and G. Kedem, “Breaking the barrier of parallel simulation of digital systems,” *Proceedings of ACM/IEEE Design Automation Conference*, 1991.

K. Chandy and R. Sherman, "The conditional event approach to parallel discrete-event simulation," *Distributed Simulation Conference*, Tampa, FL, March 1989.

Y. Chen, V. Jha, and R. Bagrodia, "A multidimensional study on the feasibility of parallel switch-level circuit simulation," *Proceedings of the 11th Workshop on Parallel and Distributed Simulations*, Lockenhaus, Austria, June 1997, pp.46-54.

S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette, "GTW: A time warp system for shared memory multiprocessors," *1994 Winter Simulation Conference*, Lake Buena Vista, FL, December 1994.

Defense Modeling and Simulation Office, *HLA Interface Specification 1.1*, February 1997.

P. Dickens, P. Heidelberger, and D. Nicol, "Parallelized direct execution simulation of message-passing parallel programs," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7(10), October 1996.

J. Evans, *Structures of Discrete Event Simulation: An Introduction to the Engagement Strategy*, Elis Horwood LTD, 1988.

F. Gomes, S. Franks, B. Unger, Z. Xiao, J. Cleary, and A. Covington, "SimKit: A high performance logical process simulation class library in C++," *Proceedings of the 1995 Winter Simulation Conference*, Arlington, VA, December 1995, pp. 706-713.

W. Franta, *The Process View of Simulation*, North-Holland, New York, 1977.

R. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, Vol. 33(10), October 1990, pp. 30-53.

R. Fujimoto, "Parallel discrete event simulation: Will the field survive?" *ORSA Journal on Computing*, Vol. 5(3), pp. 213-230.

D. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems*, Vol. 7(3), July 1985, pp. 404-425.

V. Jha and R. Bagrodia, "Parallel implementations of Maisie using conservative algorithms," *1993 Winter Simulation Conference*, Los Angeles, CA, December 1993.

V. Jha and R. Bagrodia, "A unified framework for conservative and optimistic distributed simulation," *Proceedings of the 8th Workshop on Parallel and Distributed Simulations*, Edinburgh, Scotland, U.K., July 1994, pp. 12-19.

V. Jha and R. Bagrodia, "A performance evaluation methodology for parallel simulation protocols," *Proceedings of the 10th Workshop on Parallel and Distributed Simulations*, Philadelphia, PA, May 1996, pp.180-183.

A. Law and W. Kelton, *Simulation Modeling & Analysis*, McGraw-Hill, San Francisco, 1991.

A. Law and C. Larney, *Introduction to Simulation Using SIMSCRIPT II.5*, CACI Products Company, La Jolla, CA, 1984.

J. Martin and R. B. Bagrodia "COMPOSE: An object-oriented environment for parallel discrete-event simulations," *Proceedings of the 1995 Winter Simulation Conference*, December 1995, pp. 763-767.

J. Misra, "Distributed discrete-event simulation," *ACM Computing Surveys*, Vol. 18(1), March 1986, pp. 39-65.

R. Nance, "A history of discrete event simulation programming languages," *Proc History of Programming Languages Conf,* ACM SIGPLAN Notices, Vol. 28(3), March 1993, pp. 149-175.

D. Nicol and P. Heidelberger, "On extending parallelism to serial simulators," *Proceedings of the 9th Workshop on Parallel and Distributed Simulations*, 1995, pp. 60-67.

S. Prakash, *Performance Prediction of Parallel Programs*, Ph.D. Dissertation, UCLA Computer Science Department, December 1996.

A. Pritsker, *The GASP IV Simulation Language*, John Wiley, 1974.

S. Reinhardt, M. Hill, J. Larus, A. Lebeck, J. Lewis, and D. Wood, "The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers," *Proceedings of the 1993 ACM Sigmetrics Conference*, May 1993, pp. 509-518.

D. Rich and R. Michelsen, "An assessment of the MODSIM/TWOS parallel simulation environment," *Proceedings of the 1991 Winter Simulation Conference*, 1991, pp. 509-518.

H. Schwetman, "CSIM: A C-based process oriented simulation language," *Proceedings of the 1986 Winter Simulation Conference*, 1986, pp. 387-396.

J. Steinman, "SPEEDES: A multiple-synchronization environment for parallel discrete-event simulation," *Int'l Journal of Computer Simulation*, Vol. 2(3), 1992, pp. 251-286.

UCLA Parallel Computing Laboratory, *PARSEC User Manual*, Release 1.0, Computer Science Department, University of California, Los Angeles, CA, http://pcl.cs.ucla.edu, February 1998.

D. Wagner, "Algorithmic optimizations of conservative parallel simulations," *Proceedings of the SCS Multiconference on Advances in Parallel Simulation*, Anaheim, CA, January 1991, pp. 25-32.

J. Waldorf and R. Bagrodia , "MOOSE: A concurrent object-oriented language for simulation," *Int'l Journal in Computer Simulation*, George W. Zobrist (Ed.), Vol. 4(2), 1994, pp. 235-257.

P. Wonnacott and D. Bruce, "The APOSTLE simulation language: granularity control and performance data." *1996 Workshop on Parallel and Distributed Simulation*, Philadelphia, PA, May 1996, pp. 114-123.

X. Zeng, R. Bagrodia and M. Gerla, "GloMoSim: A library for the parallel simulation of large wireless networks," *Proceedings of the 12th Workshop on Parallel and Distributed Simulations*, Banff, Alberta, Canada, May 1998 (to appear).

ACKNOWLEDGEMENTS: This research was supported by National Science Foundation PYI Award ASC-9157610 and by the U.S. Department of Defense/ARPA Contract DAAB07-97-C-D321, "Design of Mobile Adaptive Networks Using Simulation and Agent Technology," (DOMAINS) .