Proceedings of the 1996 Winter Simulation Conference
ed. J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain

PERILS AND PITFALLS OF PARALLEL DISCRETE-EVENT SIMULATION

Rajive L. Bagrodia

Computer Science Department
University of California, Los Angeles
Los Angeles, California 90095, U.S.A.

ABSTRACT

The design of efficient parallel discrete-event simulation (PDES) models often appears to be a mysterious art practiced primarily by academic researchers who have been rigorously ordained in this task. This tutorial attempts to unravel some of the mysteries. It describes the process of generating an efficient parallel implementation of a discrete-event simulation (DES) model. Common pitfalls in the parallel execution of the models are described together with suggestions on their avoidance.

1 INTRODUCTION

Parallel (or distributed) discrete-event simulation refers to the execution of a discrete-event simulation program on a parallel (or distributed) architecture (Fujimoto 1990). In recent years, interest in exploiting parallelism in the execution of discrete-event simulations in a number of domains including network design and configuration, personal communication systems, parallel programs, digital battlefields, and digital circuits has been growing. This demand has been fueled both by the increasing availability of parallel computers (PCs with multiple processing units have become widely available!) and by the increasing complexity and scalability of systems that is making sequential model execution computationally intractable.

The focus of many PDES studies remains on the design of a *parallel* simulation model rather than on the design of a discrete-event simulation (DES) model for which parallelization can be explored as *one* execution option. Part of the reason for this is that it is harder to use parallel model execution as an 'afterthought.' Unless a modeler pays careful attention to some of the complex issues that must be addressed to support parallel execution from the initial stages of model *design*, subsequent parallelization efforts may prove to be overwhelming. In this regard, there is a close analogy between the design of general purpose parallel programs and PDES: it is much harder to port a 'dusty deck' FORTRAN or C program to a parallel machine than it is to port a program that was designed for eventual migration to a parallel architecture.

This tutorial describes the process of generating an efficient parallel implementation of a DES model. It outlines some of the common pitfalls in the design of the initial DES model that can make subsequent parallelization considerably harder, if not impossible. A companion paper in this volume (Liu et al. 1996) describes a case study in porting a model for wireless network simulation to a distributed memory architecture. The next section is an overview of parallel synchronization protocols. Section 3 discusses the role of simulation languages in the design of a PDES. Section 4 addresses the range of issues that must be addressed in preparing a DES model for parallel execution. Section 5 is the conclusion.

2 PARALLEL SIMULATION PROTOCOLS

Three primary types of synchronization protocols have been described in the literature: conservative (Misra 1986), optimistic (Jefferson 1985), and mixed (Jha and Bagrodia 1994), where the latter may include sub-models that execute in either conservative or optimistic modes. This section gives an overview of the synchronization problem in PDES and presents an algorithm for its solution.

2.1 Model

A typical simulation is assumed to consist of a collection of logical processes (or LPs), where each LP models some physical process in the system. Many papers in the PDES literature use an LP to represent the sequential unit of computation in a model. However the decomposition of a model into LPs may be driven by issues of modularity and software design rather than by concerns of parallel performance. In this paper, we view a PDES as a collection of Sequential Discrete Event Simulation (SDES) models. Each SDES models a subsystem of the physical system and executes on a unique processor; the processor may represent a machine in a network of workstations or a single node of a shared or distributed memory parallel architecture. Each SDES consists of one or more LPs and uses a sequential synchronization algorithm (for example, the Global Event List algorithm) to schedule local events in their correct timestamp order.

Some mechanism must be defined to allow the SDES models to communicate with each other. These constructs may be provided in the form of explicit messages, remote procedure calls, or other mechanisms that depend on the conceptual framework that is used by the language. In this paper, we do not make any assumptions about the language or notation used by an analyst to describe the SDES. For simplicity and uniformity of exposition in this paper, we will assume that SDES models communicate with each other using time-stamped messages, and any simulation algorithm must execute the model as if all events were executed in their timestamp order.

We assume that the following set of variables are defined for each SDES (Jha and Bagrodia 1994):

- Earliest Output Time (EOT): For a given SDES *s*, EOT_s refers to the (lower bound) on the future time at which *s* will cause an event to be scheduled at another SDES. For some s, if EOT_s is infinity, the remaining SDES can be executed independently of *s*. A sink process is an example of such an SDES.
- Earliest Input Time (EIT): EIT_s of an SDES *s* refers to the earliest (future) event that may be scheduled on *s* by another SDES. (An SDES may contain unprocessed messages with timestamp smaller than its EIT). The EIT of an SDES is infinity if all events scheduled on that SDES are generated locally; a *source* process is an example.
- Lookahead: At any simulation time t, the lookahead of an SDES is a lower bound on the duration after which it will schedule an event at another SDES. The lookahead is used by an SDES to compute its EOT.

We use a simple closed queuing network model to illustrate the preceding concepts. The network consists of three FIFO servers. A job arriving at a server, waits for service, and after completion of the service proceeds with equal probability to any of the other servers in the network (Figure 1). We are interested in measuring the average and maximum time that is spent by a job at any server. Each server may be viewed as an SDES; interesting events in the system are the arrival of a job at a server (henceforth called the arrival event) and departure of a job from a server. The jobs in the system are abstracted via the events and need not be modeled as explicit logical processes. The arrival (and departure) events are modeled by sending messages among the corresponding servers.

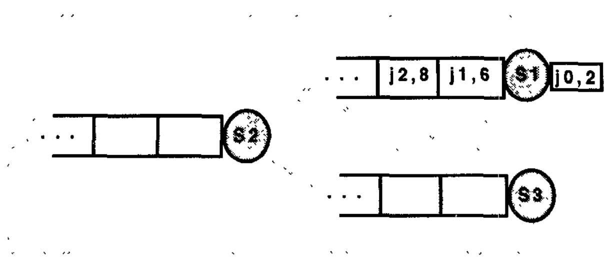

Figure 1: Closed Queuing Network

Assume that the service time at each server is 6 time units. Consider a server that is busy servicing a job: its EOT is the departure time of the current job. For install server, in Figure 1, *s1* is busy servicing job j0. EOT_{s1} is 2 + 6 = 8 (we assume that the job began service as soon as it arrived at the server). If the server is idle, its EOT will be its EIT plus its lookahead.

2.2 Synchronization

Traditionally, a PDES model was either optimistic (all SDES executed in the optimistic mode) or conservative (all SDES executed in a conservative mode). A conservative SDES cannot tolerate causality errors; hence it will only process events with timestamps less than its EIT. An optimistic SDES may additionally process events with timestamps greater than its EIT; however, the underlying synchronization protocol must detect and correct violations of the causality constraint. The simplest mechanism for this is to have the SDES (or 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. 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. All checkpoints timestamped earlier than GVT can be reclaimed. Using our model, it is sufficient for an optimistic SDES to preserve at least one checkpointed state with a timestamp smaller than its EIT. (The minimum of the EIT of all optimistic SDES is a reasonable lower bound on the GVT of the model). Given appropriate mechanisms to advance the EIT and EOT of the conservative or optimistic SDES, it is possible to implement a PDES model which contains sub-models that are executed using *optimistic, conservative, or sequential synchronization* mechanisms (Jha and Bagrodia 1994).

For a given SDES *s*, we define two sets IN_s and OUT_s respectively as the set of SDES from which s can either receive or send event messages (in the case when the communication topology is not known precisely, the sets can be constructed conservatively; in the default case, each set may contain every SDES). SDES s computes EIT_s as the minimum of the EOT of all SDES in IN_s. Given its EIT, an SDES can compute its EOT locally. The frequency and method used by an SDES to compute its EIT and the frequency with which it communicates its EOT to other SDES can have a significant impact on the performance of the PDES model.

In general, any of the GVT computation algorithms, conservative algorithms, or even a combination of the preceding algorithms can be used by a PDES to compute the EIT of each SDES (regardless of the execution mode of the individual SDES in the model). The choice of a specific algorithm for a given scenario is an efficiency rather than a correctness issue. We outline an aggressive null message based scheme: whenever the EOT of an SDES, say s, changes, EOT_s is sent using a null message to all SDES in OUT_s; the null message may of course be piggy backed on a regular message when feasible. On receipt of a new EOT from an SDES in IN_s, the SDES s recomputes its EIT (and perhaps its EOT) and propagates changes to OUT_s. It is easy to show that given a model with no zero delay cycles, such an algorithm will eventually advance the EIT of every SDES.

Consider the queuing network example. For a server process s, OUT_s and IN_s both consist of all server processes in the system, other than s. Consider the computation of EIT for a server using the null message scheme described earlier. The EOT of a server changes every time a job departs from a busy server or a job arrives at a server. When a server say *s1* forwards a job to *s2* it also sends a *null message* to *s3* (the null message to *s2* can be piggy backed with the *job* message) possibly allowing it to advance its EIT as explained earlier. Other techniques to improve the computation of EIT and lookahead are discussed subsequently.

3 SIMULATION MODEL DESIGN

A number of factors govern the choice of the specific programming language that is used by an analyst to describe the model. Among many others, these include the *conceptual framework* or *world view* adopted by the language (Evans 1988), the availability of, or familiarity of the analyst with specific languages, and perhaps the simulation libraries and support facilities provided by the language. From the perspective of this paper, an important concern in choosing a simulation language is its support for parallel execution of the model. Henceforth we refer to a language that supports parallel execution of the model as a Parallel Simulation Language (or PSL). (Bagrodia 1994) is a recent survey of languages and software to develop PDES models.

3.1 Parallel Simulation Languages

Most extant PSLs provide a set of DES primitives (scheduling events, advancing simulation time, etc.) together with a set of parallel programming primitives for thread (or object) definition, creation and interprocess (or inter-object) communication and synchronization. We use *thread* or *process* to refer to the basic unit of parallelism that is provided by the language.

A parallel language (and by extension a PSL) can use either a shared-nothing or a shared-everything model. In the former case, the processes do not have access to any shared variables; communication is entirely via messages or procedure calls. In the latter paradigm, every data item declared in the program is assumed to be accessible by every process. Note that a shared-everything programming model may be implemented on a distributed memory machine and similarly a shared-nothing model may be implemented on a shared memory architecture. Some parallel languages may also use a hybrid model, with restricted forms of data sharing. A majority of existing PSLs use the shared nothing programming model; henceforth we will use the term MP-PSL to refer to such a language. Shared-everything and hybrid models are active areas of research in PSL design. In particular, some PSLs distinguish between processes on the same and different processors allowing the former to access shared variables.

Some PSLs additionally provide constructs to modify the attributes of a model that can affect the synchronization overheads and hence its parallel performance. For instance, the Apostle (Wonnacott and Bruce 1996) simulation language provides support for granularity control of a simulation object to reduce overheads for event handling, SPEEDES (Steinman 1991) requires user-directed checkpointing as a way to provide incremental state saving, Maisie (Bagrodia and Liao 1994) and U.P.S. (Nicol and Heidelberger 1996) support user-specified lookahead specification, and Maisie also provides a variety of other constructs to monitor and optimize parallel performance including dynamic topology information, a set of constructs to reduce rollback costs, and modifying the state saving interval or time window.

PSLs also differ in their support for specific synchronization protocols. Many existing PSLs support only optimistic protocols — Sim++ (Baezner, Lomow, and Unger 1990), ModsimII (West and Mullarney 1988), Apostle, and SPEEDES to name a few. Relatively few support only conservative protocols; and even fewer, like Maisie, support conservative, optimistic, and mixed protocols. Even though conservative implementations are considerably easier to develop and have relatively few operating parameters to tune, optimistic protocols are generally perceived to be more widely applicable. However, with aggressive exploitation of lookahead in models, it has been possible to extract significant speedups from a variety of applications using conservative protocols.

3.2 Parallel Simulation Overheads

There are three primary sources of overheads in the parallel execution of a simulation model:

- Partitioning related overheads: Specific decompositions of a model into its component SDES modules sets an upper bound on the speedup that can be obtained from parallel execution of the model. The allocation of LPs to SDES determines the contribution to the overhead by load (im)balance, message communications, and related factors.
- Synchronization protocol overheads: These are the overheads that are contributed by the specific protocol that is used to synchronize the execution of the SDES. For instance, a conservative protocol introduces null message or blocking overheads whereas an optimistic protocol may introduce rollback or checkpointing overheads.
- Target architecture overheads: These costs include architecture-specific costs like message latency and context switching overheads that, in many cases, are beyond the direct control of the modeler.

As described in the next section, the analyst must recognize and reduce these overheads for successful parallelization of the DES model.

4 PARALLEL MODEL EXECUTION

The use of a PSL is a necessary, though hardly sufficient, condition for the corresponding model to run efficiently on a parallel architecture. In this section, we outline common pitfalls that must be overcome by an analyst to design a efficient parallel implementation. As with any parallel program, there is no guarantee that parallel execution of a model will yield performance benefits; however following the guidelines in this section will certainly increase those chances.

Pitfall 1 – Shared Variables: Even though a MP-PSL uses a shared-nothing programming model, it may not be possible (or efficient) for the compiler to detect uses of shared variables. Shared variables may not pose a problem for sequential implementations, but can have a serious consequence when porting the model to a parallel architecture. Consider the implementation of global variables: as the language assumes a shared-nothing model, no attempt is made to maintain consistency among the copies of the global variables stored on multiple processors.

In general, shared variables may be eliminated from a program by transforming it such that the variables are stored in one or more SDES, and all read and write operations to the corresponding data are implemented via messages or other mechanisms provided by the language for interprocess communication. However such a transformation might create a significant bottleneck in the program. In many cases, efficient alternatives are available. For instance, a common use of global variables is as write-once variables that are used to store input data like connectivity matrices or boundary values that are input to the model and are not modified subsequently. In this case, global variables can be eliminated by replicating the initialization code on each processor or by suspending all SDES until the initial values have been broadcast.

Some languages like Maisie allow the programmer to use write-once globals by providing an explicit library routine for synchronous broadcasts of initialized global variables. Other languages like Apostle allow variables to be shared among processes on the same processor. However, indiscriminate use of shared or global variables in a DES model is likely to present serious obstacles in its migration to a parallel architecture.

Pitfall 2 – Pointer Data Structures: The use of pointer data structures is similar to the issue of shared variables. An MP-PSL must not pass pointers among SDES (or logical processes) as there is no logical sharing of data between them. For PDES models that are synchronized using optimistic algorithms, even local use of pointer data structures may complicate checkpointing. Many optimistic systems use copy-state checkpointing where the entire state of a process is copied and stored in a time-stamped queue. Dynamic data structures like queues and trees make it hard to use copy-state checkpointing.

Elimination of pointers in interprocess communication typically requires that all data be passed by value. For large structured data types, this might significantly increase the communication overhead and hence program execution time. A common solution is to allow pointer passing among processes mapped to a common processor, although this adversely affects maintainability.

Pitfall 3 – Zero Delay Cycles: A model is said to contain a cycle if there is a sequence of SDES e_i, e_{i+1}, e_{i+2}, ..., e_{i+n}, e_i, n >= 0, such that each SDES may send a message to the next SDES in the sequence. The model is said to contain a zero-delay cycle if the sequence of messages exchanged in the cycle have the same timestamp. A simple example of a model with a cycle is shown in Figure 2. The merge process simply outputs messages received from its inputs in increasing timestamp order. If a path exists in the model from the merge process to a server process, and the server process has a lookahead of zero, the model could include a zero-delay cycle. A PDES model cannot contain zero-delay cycles as they can cause deadlocks in a conservative simulation (Misra 1986) or instabilities in an optimistic system.

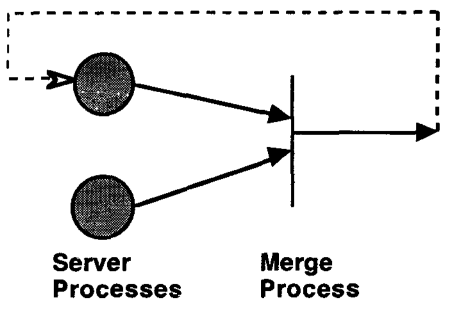

Figure 2: Merging Processes

Some parallel simulation environments (e.g. TWOS) avoid zero-delay cycles by requiring that the receive timestamp of a message be strictly greater than the (simulation) time at which it is sent (Jefferson, Beckman and Wieland 1987). This might be a significant restriction on the modeler particularly because sequential languages do not impose such constraints. A somewhat less restrictive approach is to ensure that at least one process in every cycle in the model has a non-zero lookahead (Misra, 1986).

Pitfall 4 – Poor Lookahead: Assume that a message with timestamp t_s arrives at an SDES at physical time t_r and is processed by the SDES at physical time t_p, $t_p >= t_r$ (for optimistic systems, t_p refers to the time at which the corresponding message was processed and not canceled subsequently). The duration $[t_r, t_p]$ represents simulation overhead if the corresponding SDES was conservative and its processor was idle in the duration, or if it was optimistic and the computation performed during the duration was subsequently canceled. It follows that to reduce synchronization overheads, the EIT of an SDES should be as high as possible. The EIT of an SDES is determined by the number and EOT of its predecessors. Given a model with a specific connectivity, we first examine techniques that improve the lookahead (Fujimoto 1988) for each component of the model. Subsequently we look at techniques to improve the performance of a model by reducing its connectivity. Traditionally, superior lookahead is assumed to impact the performance of only conservative SDES; however if an optimistic SDES processes a message with a timestamp smaller than its EIT, it can assert that corresponding messages with a lower timestamp can never be canceled and hence reduce its checkpointing overheads.

A commonly used technique to improve lookahead for stochastic models is to presample random distributions that are used to model various temporal intervals (Nicol 1988). For instance, consider a FIFO server that is idle at simulation time t_I. In the absence of presampling, its EOT will only be $t_I + \varepsilon$, where ε is some minimum value that can be generated by the corresponding random distribution. However, if the idle server presamples the next random value from the specified distribution, it can use that to advance its EOT. This technique is also used to improve the performance of a priority-preemptible server which would otherwise have a poor lookahead even when it was busy serving a low priority job. Language level facilities have been developed to allow the runtime system to extract the lookahead from the model or to allow a programmer to directly express the lookahead in the model (Jha and Bagrodia 1993, Nicol and Heidelberger 1996, Cota and Sargent 1989).

Compile and run time analyses of a model have also been shown to be useful in improving the application lookahead. Consider the simulation of asynchronous parallel programs where the tasks or threads communicate with each other using explicit send and receive commands. The lookahead for these applications is typically the time between two consecutive communication commands, referred to as the local block time (LBT). A common approach to estimating the LBT is by direct execution where the physical time needed to execute the code is measured by the processor clock and used to advance the local simulation clock. Techniques like predictive barrier scheduling (Legedza and Weihl 1996) use compile and run-time analysis of the program to estimate a lower bound on the time between successive communication statements in a program. Further, semantic information can be used to identify the source of an incoming message in the receive statement allowing a thread to proceed as soon as the corresponding message is available locally rather than waiting for all messages to be synchronized (Prakash and Bagrodia 1995).

Pitfall 5 – High Connectivity: Improving the connectivity information available for each SDES can improve the performance of a conservative SDES. By default, the runtime system assumes that the model is fully connected, requiring each SDES to send information about its EOT to all other SDES in the system, considerably increasing the synchronization overhead. By providing dynamic connectivity information to the model, this overhead can be reduced significantly (Jha and Bagrodia 1993).

It may also be possible to tailor the decomposition of the entities in a model to improve its connectivity. For instance, it is often possible to use clustering in network and VLSI simulations to collapse strongly connected sub-models into a single SDES and reduce the connectivity among the SDES (Gerasoulis and Yang 1993).

Pitfall 6 – Load Imbalance: Traditional techniques to decompose parallel programs (and hence parallel models) typically try to balance the computation among the partitions while minimizing the message communications among them (Sarkar and Hennessy 1988). The assignment of processes to processors may be accomplished using either *static* or *dynamic* algorithms. The primary advantage of static scheduling algorithms is the absence of any run time overheads in distributing the computation. The primary drawback of static algorithms is the difficulty in accurately estimating the computation and communication costs for each LP at compile time. For instance, in a gate-level circuit simulation, the computation load may be approximated by the number of gates in each partition and the communication load by the number of 'nets' that are cut between two partitions. A static partitioning algorithm may use these metrics to generate a statically-balanced decomposition. Clearly the level of dynamic circuit activity (which is a function of the input vectors that are fed to the circuit) among the different partitions may not be correlated well with the number of gates.

In the case of dynamic partitioning algorithms, the load at each processor is monitored at runtime. When the load is determined to be unbalanced based on the criteria specified by the system (e.g. CPU utilization, number of LPs in the scheduled queue, etc.) LPs are selected for transfer based on a specific *transfer* policy. The selected LPs are subsequently moved from one SDES to another at runtime based on specific *placement* policies that may be specified for the system. The primary advantage of dynamic techniques is that they can be much more responsive and accurate in identifying and rectifying load imbalance in a computation. However the techniques may impose a significant overhead for monitoring the load and transferring LPs. Because of the significant runtime overhead of migrating objects, extant PDES systems generally use static model decompositions to assign LPs to an SDES.

Although a significant amount of research has been undertaken in the area of load balancing for parallel and distributed computing (Shirazi, Hurson, and Kavi 1996), there has been significantly less exploration of this issue in the context of PDES (Wilson and Nicol 1996). A majority of the load balancing research in PDES has been dedicated to VLSI simulations (Bailey, Briner, and Chamberlain 1994). A parallel program is typically partitioned in a manner that minimizes message communications among the components. However, for a parallel simulation other factors may be more important: for instance, the communication topology has been found to have a significant impact on performance where models with an acyclic communication topology perform significantly better than ones that contain cycles (Cong, Li, and Bagrodia 1994). In addition, although it may not be possible to eliminate cycles completely from a model, reductions in the number of cycles can also have a significant impact on the performance. Thus from a PDES perspective, it may be preferable to use a partitioning that has a higher total communication cost that another provided that the former also has a smaller number of cycles among the partition.

Pitfall 7 – High Message Traffic: This factor is related closely to the load imbalance issue in that the decomposition of a model should be aimed at reducing the message traffic among the partitions. In addition it is often possible to aggregate messages from one SDES to another, particularly when multiple processes may be mapped to an SDES. As a large number of small messages typically contribute greater communication overhead than a small number of large messages, appropriate use of message aggregation or piggy-backing can reduce this overhead.

Pitfall 8 – Low Event or Computation Granularity: Context-switching costs can be a significant contributor to simulation overheads when a large number of processes are mapped to a single SDES. A number of factors including cache behavior, task switching times, and processes scheduling costs contribute to this overhead. Some of these costs, e.g. the scheduling policy, are determined by the design choices that have been made by the language designer and may be beyond the control of the programmer. For instance, the traditional Global Event List (GEL) algorithm schedules events across different LPs mapped to an SDES in the order of their timestamps. It is typically possible to reduce the scheduling overheads by using a parallel simulation algorithm: compute the EIT for each LP; select an LP that has the largest number of pending safe events (i.e. events with timestamps smaller than its EIT) and process all *safe* events for that LP before switching to another LP. Although this policy will cause some events to be scheduled out of their timestamp order, it is easy to show that causality will not be violated. For queuing network and network simulation benchmarks (Jha and Bagrodia 1993), this form of local scheduling has been shown to significantly improve overall execution times for a model.

In general, the overhead costs tend to increase with the number of LPs. However it is often possible to *hide* the communication latency of a parallel program by multiprocessing the LPs mapped to an SDES. This behavior has been observed for parallel programs in many applications and has led to the design of numerous thread packages. The support for multiple processes at an SDES can similarly improve the performance of a PDES model.

Pitfall 9 – Low Inherent Parallelism: It may sometimes be the case that the parallel implementation of an application fails to yield significant performance improvements because the application itself or a specific decomposition contains a low degree of inherent parallelism. As an example of such a scenario consider the simulation of a network containing a large number of nodes but a very small number of jobs. As the parallel activity is determined by the processing associated with each job, there will be insufficient parallelism in the model. In general, it is a non-trivial problem for an analyst to identify the extent of parallelism that is present in a PDES model.

Theoretical metrics like *critical path* have been used to provide a loose lower bound on the parallel execution time of a model (Lin 1990). The critical path has limited practical utility because it ignores many common and often unavoidable sources of overhead including message latency, load imbalance etc. Recently, the notion of an *Ideal Simulation Protocol* or *ISP* (Jha and Bagrodia 1996) has been suggested to experimentally compute a realistic lower bound on parallel execution time. This protocol includes all overheads that arise in the parallel execution of a model *other than those that are directly attributed to the synchronization protocol.* In other words ISP allows an analyst to compute the speedup that could be obtained from the parallel execution of a PDES model, if the synchronization overheads were zero. ISP could thus be used by an analyst to determine the parallelism potential of a model (or a specific decomposition) before expending significant effort in trying to tune other simulation-specific parameters.

Pitfall 10 – High Checkpointing Overheads: Checkpointing is used in optimistic protocols to support rollbacks to cancel incorrect computation. The simplest technique is copy-state checkpointing which copies the entire state of an SDES or LP before processing each event. It is often more efficient to use *interval checkpointing* where the state of an SDES is saved after processing multiple events. Periodic or interval checkpointing reduces the total state saving time; however it increases the computation time because some correct events must now be recomputed. The checkpointing frequency that will yield optimal performance is a tradeoff between the preceding factors and is typically application-dependent. An analytical formulation to select an optimal checkpointing interval has been described in (Lin et al. 1993). The preceding paper also describes an algorithm that can be used to select an 'optimal' checkpoint interval during the execution of a model. Subsequent work has extended these results to use adaptive checkpointing intervals.

For many applications including circuit and battlefield simulations, the state of an SDES or even an LP can be very large. Further it is often the case that only a small fraction of its total state space is modified when an event is processed. For such applications the use of copy-state checkpointing can increase the synchronization overheads sufficiently to offset any performance benefits of parallel execution. An alternative is to use *incremental* state saving, where only the portion of the object's state that is modified by an event is saved. Incremental state saving can either be programmer-directed or system-directed. In the latter case, two possibilities exist: the run-time system can explicitly compare the old and new states of an object and only save the modified portions, or save a history of all modifications as they are made to an object. Incremental state saving can reduce checkpointing costs but considerably increase rollback costs because the previous state of the object must now be reconstructed using the modification history or the incrementally saved states. Comparison of the two methods for checkpointing is an ongoing research area (Palaniswamy and Wilsey 1993).

5 CONCLUSIONS

The increasing complexity of many DES models has led to an increased the demand for PDES. However, the process of developing an efficient parallel model, even in languages that support PDES, remains a challenging task. This tutorial described some of the more common pitfalls that analysts need to be aware of when embarking on the journey to parallelize a DES model.

ACKNOWLEDGMENTS

This work was supported by the U.S. Advanced Research Projects Agency ARPA-CSTO, Air Force Contract F-30602-94-C-0273, 'Scalable Systems Software Measurement and Evaluation' and Dept. of Defense Contract DABT-63-94-C-0080, 'Transparent Virtual Mobile Environment.' The author would like to thank Vikas Jha, Yu-an Chen, Jay Martin, Rich Meyer, Brian Park and others in the UCLA Parallel Computing Laboratory, for their role in the many useful discussions that led to this paper.

REFERENCES

Bagrodia, R. 1994. Language support for parallel discrete-event simulations. In *Proc. 1994 Winter Simulation Conference*, ed. J. D. Tew, S. Manivannan, D. Sadowski, and A. Seila, 1324-1331.

Bagrodia, R. and W.-T. Liao. 1994. MAISIE: A language for the design of efficient discrete-event simulations, *IEEE Trans. Software Eng.*, 20(4), 225-238.

Baezner, D., G. Lomow, and B. W. Unger. 1990. Sim++: The transition to distributed simulation. In *Proc. 1990 SCS Multiconference on Distributed Simulation*, 211-218.

Bailey, M. L., J. V. Briner, Jr. and R. D. Chamberlain. 1994. Parallel logic simulation of VLSI systems.

ACM Computing Surveys (26) 3:255-294.

Cong, J., Z. Li, and R. Bagrodia. 1994. Acyclic multi-way partitioning of boolean networks. In *Proc. ACM/IEEE Design Automation Conf.*, 670-675.

Cota, B. A. and R. G. Sargent. 1989. Automatic lookahead computation for conservative distributed simulation. Technical Report CASE Center 8916, Simulation Research Group and CASE Center, Syracuse University, Syracuse, NY.

Evans, J. B. 1988. *Structures of discrete event simulation: an introduction to the engagement strategy*. Elis Horwood Ltd.

Fujimoto, R. 1988. Lookahead in parallel discrete event simulation. *Int'l Conference on Parallel Processing*.

Fujimoto, R. 1990. Parallel discrete event simulation. *CACM*, 33(10):30-53.

Gerasoulis, A. and T. Yang. 1993. On the granularity and clustering of directed acyclic task graphs. *IEEE Trans. on Parallel and Distributed Systems*, 686-701.

Jefferson, D. 1985. Virtual Time, *ACM TOPLAS*, 7(3):404-425.

Jefferson, D., B. Beckman, and F. Wieland. 1987. Distributed simulation and the time warp operating system. In *Symposium on Operating Systems Principles*.

Jha, V. and R. Bagrodia. 1993. Transparent implementation of conservative algorithms in parallel simulation languages. In *Proc. 1993 Winter Simulation Conference*, ed. G. W. Evans, M. Mollaghasemi, E. C. Russell, and W. E. Biles, 677-686.

Jha, V. and R. Bagrodia. 1994. A unified framework for conservative and optimistic distributed simulation. In *Proc. 8th Workshop on Parallel and Distributed Simulation*, ed. D. K. Arvind, R. Bagrodia, J. Y. Lin, 12-19, IEEE Computer Society Press.

Jha, V. and R. L. Bagrodia. 1996. A performance evaluation methodology for parallel simulation protocols. In *Proc. 10th Workshop on Parallel and Distributed Simulations*, 180-183, IEEE Computer Society Press.

Legedza, U. and W. E. Weihl. 1996. Reducing synchronization overhead in parallel simulation. In *Proc. Tenth Workshop on Parallel and Distributed Simulation*, 86-95, IEEE Computer Society Press.

Lin, Y. 1990. Understanding the limits of optimistic and conservative parallel simulation. PhD thesis, University of Washington, Seattle.

Lin, Y., B. R. Preiss, W. M. Loucks, and E. D. Lazowska. 1993. Selecting the checkpoint interval in time warp simulation. In *Proc. Seventh Workshop on Parallel and Distributed Simulation,* ed. R. Bagrodia and D. Jefferson, 3-10. IEEE Computer Society Press.

Liu, W., C. Chiang, H. Wu, V. Jha, M. Gerla, and R. Bagrodia. 1996. Parallel simulation environment for mobile wireless networks. In *Proc. 1996 Winter Simulation Conference*, ed. J. Charnes and D. Morice.

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

Nicol, D. M. 1988. Parallel discrete event simulation of FCFS stochastic queuing networks. In *Parallel programming: experience with applications, languages and systems*, 124-137. ACM SIGPLAN.

Nicol, D. M. and P. Heidelberger. 1996. On extending more parallelism to serial simulators. In *Proc. Tenth Workshop on Parallel and Distributed Simulation*, 202-205, IEEE Computer Society Press.

Palaniswamy A. C. and P. A. Wilsey. 1993. An analytical comparison of periodic checkpointing and incremental state saving. In *Proc. Seventh Workshop on Parallel and Distributed Sim.*, ed. R. Bagrodia and D. Jefferson, 127-134. IEEE Computer Society Press.

Prakash, S. and R. Bagrodia. 1995. Parallel simulation of data parallel programs. In *Proc. Eighth Workshop on Languages and Compilers for Parallel Computing*.

Sarkar V. and J. Hennessy. 1988. Compile-time partitioning and scheduling scheme of parallel programs. In *Proc. SIGPLAN '88 Symposium on Compiler Contruction*, 17-26.

Shirazi, B., A. R. Hurson, K. Kavi. 1996. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press.

Steinman, J. 1991. SPEEDES: synchronous parallel environment for emulation and discrete event simulation, In *Advances in Parallel and Distributed Simulation*, 95-103. SCS Multiconference, Anaheim, CA.

West, J. and A. Mullarney. 1988. ModSim: a language for distributed simulation. In *Proceedings of 1988 SCS Multiconference on Distributed Simulation*, San Diego, CA, 155-159.

Wilson, L. F. and D. M. Nicol. 1996. Experiments in automated load balancing. In *Proc. Tenth Workshop on Parallel and Distributed Simulation*, 4-11, IEEE Computer Society Press.

Wonnacott, P. and D. Bruce. 1996. The APOSTLE simulation language: granularity control and performance data. In *Proc. Tenth Workshop on Parallel and Distributed Simulation*, 114-123, IEEE Computer Society Press.

AUTHOR BIOGRAPHY

RAJIVE L. BAGRODIA is an Associate Professor of Computer Science at UCLA. He obtained a Bachelor of Technology in Electrical Engineering from the Indian Institute of Technology, Bombay, in 1981, and the M.A. and Ph.D. degrees in Computer Science from the University of Texas at Austin, in 1983 and 1987 respectively. His research interests include distributed algorithms, parallel languages, programming methodology and performance evaluation.