Structured Parallel Simulation Modeling and Programming

Yong Meng TEO, Seng Chuan TAY and Siew Theng KONG
Department of Information Systems & Computer Science
National University of Singapore
Lower Kent Ridge Road
SINGAPORE 119260
`email: teoym@iscs.nus.edu.sg`

Abstract

Parallel discrete-event simulation research has focused mainly on designing efficient parallel simulation protocols. However, the exploitation of parallel simulation technology in real-life applications has been hindered mainly by the lack of simulation support tools. This paper describes the design of `SPaDES` *(Structured Parallel Discrete-Event Simulation), a parallel simulation environment for developing portable simulation models, and a platform for design experimentation of parallel simulation synchronization protocols. An implementation of the environment,* `SPaDES/C++`*, cleanly separates simulation modeling and programming from the details of parallelization such as parallel simulation synchronization and parallel programming. For ease of portability and modular design,* `SPaDES/C++` *is implemented as a parallel simulation library. A comparison of* `SPaDES/C++` *with* `CSim` *and* `Simscript` *using two application examples is discussed.*

Keywords: *structured parallel simulation, simulation environment, process-oriented modeling, simulation programming*

1 Introduction

Parallel simulation is an important technology that exploits compute intensive simulation using parallel computers. Parallel discrete-event simulation (PDES), also called distributed simulation, refers to the execution of a single discrete-event simulation program on a parallel computer [6, 11]. As parallel simulation is a hard synchronization problem, automatic parallelization of the sequential simulation program does not expose enough concurrency. In PDES, synchronization refers to the action necessary to ensure that simulated events are processed in a non-decreasing simulation time sequence. Time synchronization, however, is complicated by many factors such as processor capability, communication latency and load balancing.

Research in the last fifteen years focuses on developing efficient parallel algorithms for synchronizing parallel activities [5, 9]. However, inadequate attention has been devoted to the programmability problem in parallel simulation, i.e., to reduce the effort and expertise required to develop efficient parallel simulation models [7]. Typically, in developing a parallel simulator the simulationist is confronted with two main problems in translating the conceptual model of the real world system (hereafter call the *physical system*) into a simulation program: the program must adhere to the programming model of the particular parallel computer, and to incorporate the selected parallel simulation synchronization protocol. Our objective is to provide a structured parallel simulation environment that facilitates simulation modeling and programming.

The rest of the paper is divided into seven sections. Section 2 discusses modeling and parallel simulation programming. Section 3 presents an overview of SPaDES design including its process-oriented modeling methodology, set of simulation primitives, simulator structure, etc. Section 4 discusses an implementation of SPaDES called SPaDES/C++. In section 5, we compare the performance of SPaDES with CSim and Simscript. Section 6 gives an overview of related work, and section 7 presents the concluding remarks.

2 Modeling and Programming

Simulation is the process of modeling a proposed or real dynamic system and observing its behavior over time. A *model* is a representation of the real system that includes entities of the system, and the behavior and interactions of those entities. A simulation environment must support some type of modeling framework to facilitate development and implementation of the simulator. The world view supported in a parallel simulation language should provide a framework for modeling the physical system. In addition, the world view should also be independent of the parallel simulation synchronization scheme, and the parallel programming model used in the underlying parallel computer. This paper adopts a modified process-interaction modeling view called *process-oriented* world view for structured parallel simulation modeling and programming.

Programming effort incurred in PDES is demanding. The programmers not only have to write codes to mimic a real-world applications, but also to program parallel simulation mechanisms to ensure the causality correctness of simulation result. To help the programmers concentrate on the problem domain only, a comprehensive parallel simulation environment is required. SPaDES provides a set of modeling primitives implemented as library routines to support parallel simulation and programming. In this paper, we illustrate this extension using C++; a favorite choice due to its object-oriented characteristics in addition to the efficient implementation of C. The major advantages of our SPaDES environment are that a new compiler is not required, thus increasing its portability, and the programming environment of the existing sequential language such as debuggers, graphical tools, etc. is readily available to the language designers as well as the programmers.

3 SPaDES Simulation Environment

The main objective of the SPaDES environment is to permit the users to develop a parallel simulator without being overly concerned with the execution environment, and achieve acceptable performance on a parallel machine. SPaDES exploits the similarity of processes in the process-interaction world view, objects in object-oriented programming, and logical process view in PDES by providing a consistent process-oriented modeling world view.

3.1 General Structure

The general structure of SPaDES is divided into three layers. The first layer consists of a conceptual model of the real-world problem being modeled that is constructed based on the process-oriented methodology. Entities and servers in the real-world are modeled as Processes and Resources respectively. For simplicity, we will term these as simulation processes (SPs). In the operational layer which is hidden from the user, these SPs are in turn mapped onto logical processes (or LPs) and messages on which the parallel simulation mechanism is based. LPs are mapped onto physical processes (PPs) that are then mapped onto the processors for execution. Each PP represents an independent thread of execution analogous to a run-time process.

Given a real world problem, a simulationist will formulate a conceptual model using the process-oriented methodology. Each of the SPs in the conceptual model has a direct mapping onto a logical process (see figure 1). SPaDES also provides mapping schemes to allow user to control the granularity of a physical process by allowing several LPs to be clustered in a PP. Similarly, several PPs may also be mapped onto the same processor to improve the processor utilization, and to reduce communication overhead.

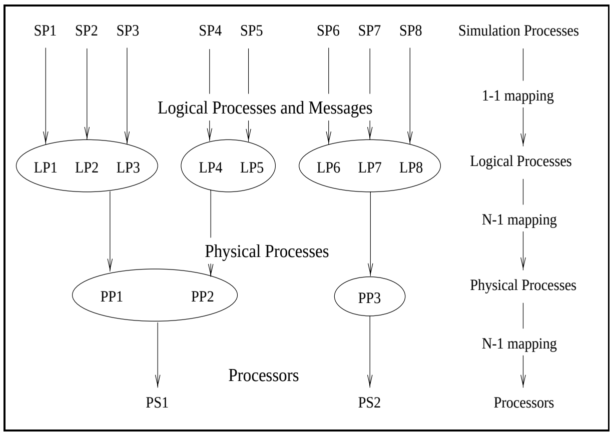

Figure 1: General Structure of SPaDES

SPaDES also places great emphasis on structured modeling and programming in the development of parallel simulators by providing the users with a set of well-defined modeling primitives, object classes and a programming template.

3.2 Process-Oriented Modeling

We categorize real-world entities into *permanent* and *temporary* entities. This is similar to the process-interaction world view adopted in Simscript and DEMOS that distinguish between active and passive components. A temporary entity does not exist throughout the simulation duration. Temporary entities are modeled as a set of processes. For example, *customers* in a queuing model are temporary entities. A SPaDES process routine models the sequence of events carried out by a temporary entity during its lifetime. Permanent entities correspond to entities that exist throughout the simulation duration. SPaDES simplifies the modeling of permanent entities through “resources”. An example of a permanent entities is a *server.*

3.3 Processes and Resources

A *process* models an active entity in the real-world system encompassing both the actions of the events and the passage of simulation time between events. In a typical simulation, many instances of a process are created, e.g., different customers that interact with each other in some way and eventually leave the system. A parallel simulation language (PSL) must provide basic primitives for manipulating processes such as creating and terminating processes, sending messages to processes, and waiting for messages and/or simulation time to elapse.

Figure 2 summarizes the possible states of a SPaDES process and the operations that trigger a transition from one state to another. The states include:

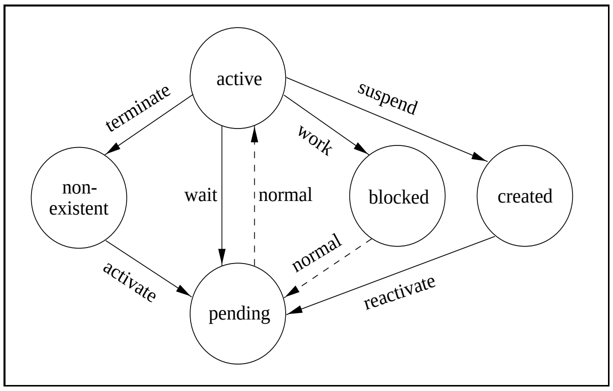

Figure 2: State Transition of a SPaDES Process

pending	- process is awaiting execution in the event list
active	- process is executing its process-routine
blocked	- process is waiting for resource in the current event list (CEL)
created	- process is not in any of the above three states
non-existent	- process is no longer in existence

Permanent entities that provide services are modeled as resources, i.e., they represent servers that can be acquired by temporary entities. Real-world examples of resources are checkout counters in a supermarket, tables in a restaurant or physicians in a health clinic. The functions that a resource performs are routine: examine its queue; if a process is available, service process otherwise set state of resource to idle. A new resource is modeled by giving it a name, and specifying the number of server units available.

3.4 Synchronization Protocols

To support the concurrent execution of simulation events, parallel simulation requires additional synchronization protocols to ensure the causality correctness of event execution. The current implementation uses the optimistic, also called Time Warp [9], mechanism to coordinate parallel event execution. A throttling scheme that better manage speculative event execution is also implemented [18]. In SPaDES, such a coordination is transparent to users.

3.5 Parallel Simulation Primitives

SPaDES provides two categories of primitives, namely (i) process manipulation primitives and (ii) simulation configuration primitives.

3.5.1 Primitives for Manipulating Processes

The modeling of processes and its state transitions is supported by *six* basic primitives: **activate**, **work**, **wait**, **suspend**, **reactivate** and **terminate**.

An *activate* primitive creates a new process instance and schedule it for execution at the specified server. It has the following form:

```
activate(process, time, phase)
```

where `process` represents the process to be activated, `time` is the activation time of the process, and `phase` specifies the starting phase with which the process is to execute upon activation. For example,

```
activate(job, exponential(5), 1)
```

creates a new process instance called job, schedules it for activation at a time equal to the current clock time plus the value returned by the exponential random variate generator, and start execution at the first phase of the process routine.

A *work* primitive performs *three* functions associated with a resource: request for resource, acquire a resource for service time, and release the resource on service completion. It is of the following form:

```
work(resource, time, units)
```

where `resource` denotes the type of resource requested, `time` represents the service time required, and `units` indicates the number of service units of `resource` requested. For example,

```
work(cpu, exponential(5), 1)
```

represents a request for one unit of a resource called cpu, for a service period sampled from an exponential distribution with a mean of 5. If the requested resource is unavailable (i.e. servicing another process), the process concerned is automatically placed into the queue associated with the resource. That process is suspended from execution and waits in the queue for the resource. The resource on completing service picks up the next waiting process in the queue for execution.

The next primitive called *wait* models the passage of time. It is used to suspend a process for a specified duration of simulated time. It has the following form:

`wait(time)`
where `time` models the amount of time. This primitive is used to model interarrival time of processes arriving at the system. For example,
`wait(exponential(8))`
suspends a process for a time period sampled from an exponential distribution with a mean of 8.

The *suspend* primitive is used to delay a process for an unknown amount of time. It has the following form:
`suspend()`
The process that is suspended must be reactivated explicitly by calling *reactivate* primitive.

The *reactivate* primitive is used to reactivate a suspended process. It has the following form:
`reactivate(process, time)`
where `process` is the process to be reactivated, and `time` is the time at which the process is to be activated.

A *terminate* primitive destroys a process that has completed execution. It has the following form:
`terminate()`

3.5.2 Primitives for Configuring Simulation

SPaDES permits users to control the mapping of SPs, LPs, and PPs onto processors via the primitives: **mapNode**, **mapHost** and **mapProcess**. A default mapping is provided but these primitives allow fine-tuning of the simulator to further improve its performance.

The *mapNode* primitives allows several LPs to be mapped onto a physical process. As LPs grouped together on the same physical process have direct access to the input queues of one another, communication overhead can be reduced. It has the following forms:
`mapNode(numLPs, Process, PP)`
`mapNode(numLPs, P0, [P1, P2, ... ] PP)`
where `numLPs` indicates the number of LPs mapped to the physical process, and `Process` is an array containing a corresponding number of LPs, and `P0, P1, P2` ... are individual LPs, and `PP` is the physical process itself.

The next primitive, *mapHost* allows one to control the location of the physical processes. That is, the physical processes can be spawned onto any particular processor as desired. However, this feature is only available if the SPaDES library is implemented with a message passing package that allows dynamic creation of processes during execution. The primitive has the form:
`mapHost(numPPs, PP1, [PP2, ...] hostname)`
where `numPPs` indicates the number physical processes, `PP1, PP2, ...` are the physical processes and hostname specifies the name of the machine where the processes are to be spawned.

As mentioned in section 3.1, each SP is an LP that is capable of autonomous execution. Each LP has it own input queue, local clock etc. Thus each SP is capable of starting its own thread of execution at simulation startup if there is already an event message present in the input queue. The third primitive *mapProcess* allows user to initialize a SP with a starting event message. It is of the form: `mapProcess(process, SP)` where `process` is an event message and SP is the simulation process whose input queue with which the event message is to initialize.

The set of primitives introduced is independent of the base language, and can be easily implemented as a library extension to general-purpose programming languages.

3.6 Simulator Structure

The conceptual structure of a SPaDES simulator and its relationship with the simulation library is shown in figure 3. The application program supplied

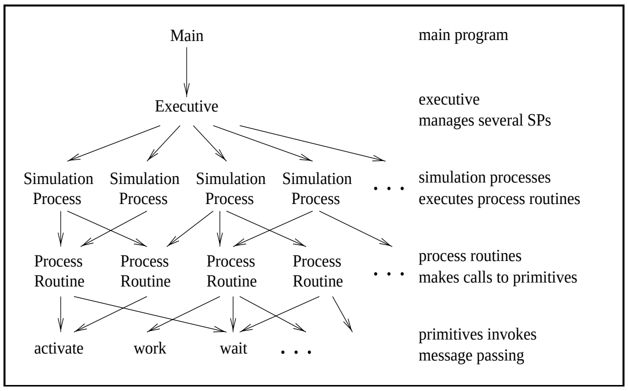

Figure 3: Structure of a SPaDES Simulator

by the user initializes and starts the simulation by passing control to the parallel simulation kernel. The simulation kernel consists of an Executive that manages several SPs (or LPs at the operational layer). Each of the SP is in turn scheduled for execution, i.e., executes the corresponding SP process routine. The execution of the process routines will generate calls to the primitives which results in messages being passed between each of the LPs.

4 Implementation of SPaDES/C$_{++}$

The SPaDES/C$_{++}$ library is structured as a hierarchy of classes, where the basic functionality is defined in low-level classes, while layers of more specific functionality are defined in derived classes. The functionality of the SPaDES library mainly comes from three basic classes, namely *Process*, *Executive* and *Resource* classes.

The *Process* class is for modeling entities in the physical system (see figure 4). A new process can be introduced by deriving from the base process class or from other derived class of the base class. Additional methods and attributes in the newly derived class can also be defined. The virtual method *executeProcess()* defines the main body of the process entity. This method contains the sequence of activities or events to be executed pertaining to a particular type of process entity, and is invoked by the simulation kernel each time an instance of this entity is active. Therefore, each time a new process type is to be constructed, the method *executeProcess()* must be redefined for that new process type. Polymorphism allows the simulation kernel to execute different types of process routines by invoking the same virtual method *executeProcess()*. The primitives `activate`, `work`, `wait`, `suspend`. `reactivate` and `terminate` supplied by the `Process` class works in the same way as described earlier.

```
class Process
{
public:

   // constructor of process
   Process(int size, int priority=0);
   virtual ~Process(){}

   // main body of a process, to be overrided
   // in a derived class
   virtual void executeProcess() {}

   // primitives
   // activate() and reativate() are macros for
   // _activate() and _reactivate()
public:
   void _activate(double, int);
   void _reactivate(double);
protected:
   void work(Resource*, double, int=1);
   void wait(double);
   void suspend();
   void terminate();

protected:
   // current phase of execution
   int phase;
};
```

Figure 4: SPaDES/C++ Process Class Interface

The *Executive* class provides the interface between the simulation application and the parallel simulation kernel (see figure 5). It manages several tasks such as the initialization of the simulator, schedules the next process for execution, controls the routing of messages and also to generate simulation output at the end of the simulation. However, users need only to concern themselves with the initialization of the simulator which are provided via interface functions in the *Executive* class.

```
class Executive
{
public:
   Executive();
   // simulation initialization functions
   void initialize(int, char**);
   void init();
   void startSimulation(double);
   void resetSimulation();

   // configuration functions
   void mapNode(int, Process**, int);
   void mapNode(int, ..., int);
   void mapHost(int, ..., char*);
   void mapProcess(Process*, Resource*);
};
```

Figure 5: SPaDES/C++ Executive Class Interface

The initialize() performs general initialization of the simulator that consists of operations that are hidden from the user. These include spawning of all the parallel processes and sending initialization information to them, etc. The init() method is left blank so as to allow user to fill in their own initialization of the starting state of the simulator. An example of init():

```
for (int i=0; i<15; i++)
{
    activate(terminal[i], 0, 1);
}
```

This example activates 15 terminal processes.

The routines *startSimulation()* and *resetSimulation()* are used to start the simulation and to reset the simulation respectively. The configuration functions are used to control the configuration of the simulator and functions in the same way as described in section 3.5.2.

The *Resource* class is used to model service stations of the real-world systems. It provides several features to help simplify the modeling of the service stations. They include: implicit queuing for single queues which are by default FCFS) and also a set of pre-defined statistics which will be automatically output to a file at the end of the simulation. The set of statistics include: utilization, average queue length, average response time etc.

A SPaDES/C++ programming template is divided into two main parts: the *declaration part* and the *implementation part* as with all C++ programs. The declaration part consists of the header file "spades.h" which contains all the declarations of the SPaDES simulation library. This is followed by declaration of all Process and Resource classes needed in the simulation derived from the base classes provided in the library.

The implementation contains the global variable declaration of the simulation processes involved in the simulation and all implementations of the methods in the declaration part. The method *executeProcess()* defines the sequence of events belonging to each `Process`. Next, the user needs to define his own initialization of the simulation using the *init()* method.

In the main program, initialization of the simulation is performed by invoking the *initialize()* method passing `argc` and `argv` as the arguments. Next, the *startSimulation()* statement starts the simulation, and the argument allows the simulationist to specify the duration of the simulation run.

5 Performance Evaluation

The SPaDES workbench is currently implemented on a network of workstations and on a 32-node Fujitsu AP3000 parallel computer using PVM for message-passing. For purpose of comparison, we have constructed the simulators for two examples: (i) time-shared computer system [8] (figure 6), and (ii) cafeteria chain (figure 7). In figure 7 each number shown in a server station indicates the number of servers positioned in the station.

The sequential simulators are programmed in CSim [19] and Simscript [14], while their parallel versions in SPaDES/C++. Due to space constraint, we present the performance obtained from the Fujitsu AP3000 parallel computer only.

Let p denotes the number of nodes used in parallel simulation. As shown in table 1, the source program code sizes required by SPaDES are the least among the three implementations for both examples. In terms of the executable file size, however, SPaDES requires the largest size due to its linkage to the libraries used for process synchronization and communication. The runtime of SPaDES implementations, not surprisingly, is much larger than the other two sequential implementations. The poor performance is due mainly to the fine granularity of simulation events which is application dependent. In both example applications, the workload of each event primarily consists of updating of queue length, server status (idle/busy), and the number of arrivals or departures, thus giving a small event grain size.

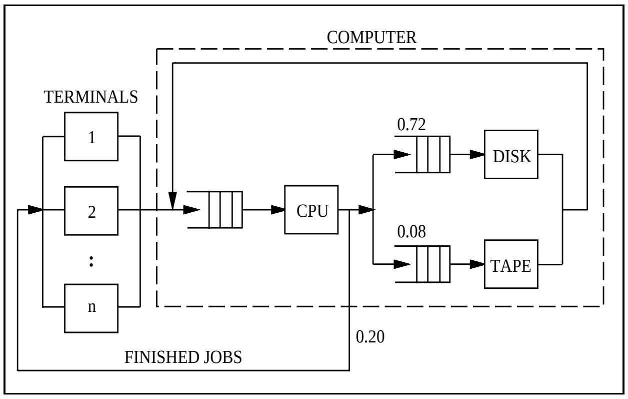

Figure 6: Time-shared Computer Simulation Model

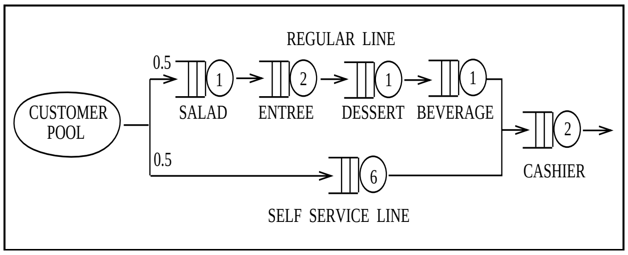

Figure 7: Cafeteria Chain Simulation Model

It is well known that coarse-grain process is necessary to exploit process-based parallelism. In a separate experiment we mimic coarse-grain events by increasing the workload of each simulation event to observe its effect. Let η be the additional workload per event. In this investigation η is simulated by a spin loop with runtime ranging from $1k$ to $8k$, where $k \approx 5$ milliseconds. Table 2 shows the runtime incurred by each example where the duration of simulation for the first example is 25,000 time units while that of the second example is 100,000. As the event granularity is increased, we observe a better elapsed time performance in SPaDES as compared to CSim and Simscript. This experimentation shows that SPaDES is scalable for coarse simulation event.

6 Related Work

Many sequential simulation programming languages (SSL) such as GPSS/H [15], Simscript [14], Simula [4] and ModSIM III [1] are available. For developing sequential simulators application specific packages such as SES/Workbench [16], AutoMod [13] and ProModel [3] are increasingly used. These packages promote ease of use through user-friendly graphical interface for inputting the model, and provide high-level modeling components. However, the development of parallel

program	implementation	measure			
		line of code	exec. size	comp/link time	exec. time
time-shared computer	CSim	296	49,385	4.8	2.0
	Simscript	114	475,168	12.6	1.6
	SPaDES ($p = 4$)	77	770,372	8.1	165.5
cafeteria chain	CSim	312	53,756	5.1	2.8
	Simscript	119	529,024	13.7	1.8
	SPaDES ($p = 7$)	100	770,548	6.3	14.3

Table 1: Performance Comparison

program	implementation	η (where k=5 milliseconds)			
		k	2k	4k	8k
time-shared computer	CSim	211.4	421.8	846.4	1687.9
	Simscript	191.9	381.7	761.5	1520.1
	SPaDES	180.7	202.9	250.2	358.4
cafeteria chain	CSim	149.9	278.5	456.0	800.2
	Simscript	111.2	236.6	427.4	758.2
	SPaDES	20.3	27.5	38.9	64.2

Table 2: Varying Event Grain Size

simulation languages is confined mainly to academia and lacks behind that of the SSL. To the best of our knowledge, current popular commercial simulation languages and packages have yet to extend their implementation to exploit parallel simulation and parallel computing technology. Table 3 compares SPaDES with six experimental parallel simulation languages namely TWOS [10], SPEEDES [17], SIMA [12], Maisie [2], MOOSE [20], and APOSTLE [21].

Five typical PSL features are included. The implementation column denotes whether the set of simulation primitives is implemented through OS calls, a compiler or as a simulation library. The last column distinguishes between implementations where message-passing is managed by the simulationist against those that is transparent. SPaDES supports the process-oriented world view while the other PSLs support either the event-scheduling or process-interaction view. With the exception of APOSTLE which is a new PSL, SPaDES uses the same design philosophy as most of the other PSLs by extending C or C++. A notable difference of SPaDES is that message-passing is fully transparent to the simulationist; this is in line with our objective of achieving total parallelism and synchronization transparency. All the other PSLs listed provide message-passing primitives accessible to the simulationists. While achieving total transparency imposes a higher performance penalty, it is a necessary trade-off to encourage wider scale adoption of parallel simulation technology. The historical development of SSLs bear evidence to our believe that this trade-off is essential.

7 Conclusions

While parallel hardware promised to speed simulation, parallel simulation has not achieved popularity. Research and development of modeling support tools and languages for parallel simulation lacks behind that of the plethora of sequential simulation languages and packages available. Another aggravating factor is that parallel programming is still very much an area of research. This paper introduces a structured parallel simulation modeling and programming methodology that relieve the simulationist from the intricacies of parallel programming and simulation parallelization. The consistence process view adopted from modeling to implementation allows a simulationist to concentrate on describing the process flow in terms of high level blocks or network constructs, while the interactions among processes and its implementation are handled automatically. Our examples application show that SPaDES requires fewer lines of code. One of the few widely accepted axioms of software engineering is that coding takes longer if you write more lines of code. Moreover, the reuse of libraries of pre-built objects holds out promise of real productivity gain in simulator development. The preliminary result shows that SPaDES is scalable if the grain size of simulation

PSL	world view	language	synchronization	implementation	message-passing
TWOS	event-oriented	C	optimistic	OS calls	user-defined
SPEEDES	event/data object	C++	both	OS calls	user-defined
SIMA	event-scheduling	C	conservative	library	user-defined
Maisie	process-interaction	C	both	compiler	user-defined
MOOSE	process-interaction	C++	both	compiler	user-defined
APOSTLE	process-interaction	new PSL	optimistic	compiler	user-defined
SPaDES	process-oriented	C++	both	library	transparent

Table 3: Typical Features of Parallel Simulation Languages

event is larger than the overheads incurred.

A visual graphical interface to support the development of parallel simulators including visualization and debugging, and a facility for automatic generation of SPaDES program codes are currently being developed. Performance loss in parallel simulation is an important issue. Ongoing works include parallel-event instrumentation and profiling in SPaDES, analytic performance model, process partitioning and mapping, etc. Understanding the parallelism profile of a parallel simulator is crucial to optimizing its runtime performance, and in popularizing the exploitation of parallel simulation technology.

References

[1] US Army, *"ModSim Reference Manual,"* Version 3, October 1989.

[2] R. L. Bagrodia and W. Liao, *"Maisie: A Language for the Design of Efficient Discrete-Event Simulations,"* IEEE Transactions on Software Engineering, Vol. 20(4), pp. 225-238, April 1994.

[3] S.P. Baird and J.J. Leavy, *"Simulation Modeling using ProModel for Windows,"* Proc. of Winter Simulation Conference, pp. 527-532, 1994.

[4] O. J. Dahl and K. Nygaard, *"SIMULA - An Algol Based Simulation Language,"* CACM, Vol 9. pp. 671-678, 1966.

[5] A. Ferscha, *"Parallel and Distributed Simulation of Discrete Event Systems,"* in Handbook of Parallel and Distributed Computing, McGraw-Hill, 1995.

[6] R. M. Fujimoto, *"Parallel Discrete Event Simulation, Communications of the ACM,"* Vol 23, No. 10, pp. 31-53, October 1990.

[7] R. M. Fujimoto, *"Parallel Discrete Event Simulation: Will the Field Survive?,"* ORSA Journal of Computing, Vol. 5, No. 3, pp. 213-230, Summer 1993.

[8] S. V. Hoover and R. F. Perry, *"Simulation - A Problem-Solving Approach,"* Addison-Wesley Publishing Company, 1989.

[9] D. R. Jefferson, *"Virtual Time,"* ACM Transactions on Programming Languages and Systems, Vol. 7, No. 3, pp. 404-425, July 1985.

[10] D. R. Jefferson, et al., *"Distributed Simulation and the Time Warp Operating System,"* ACM Operating Systems Review, Vol. 21(5), pp. 77-93, 1987.

[11] D. Nicol and R. M. Fujimoto, *"Parallel Simulation Today,"* Annals of Operations Research, Vol. 53, pp. 249-286, December 1994.

[12] Rajaei H., *"SIMA: An Environment for Parallel Discrete-Event Simulation,"* Proceedings of the 25th Annual Simulation Symposium, pp. 147-155, April 1992.

[13] M. Rohrer, *"AutoMod,"* Proc. of Winter Simulation Conference, pp. 487-492, 1994.

[14] E.C. Russell, *"SIMSCRIPT II.5 and SIMGRAPHICS Tutorial,"* Proc. of Winter Simulation Conference, pp. 223-227, 1993.

[15] T.J. Schriber, *"An Introduction to Simulation using GPSS/H,"* John Wiley, 1991.

[16] SES/Workbench - Sim Language Reference, Release 3.0, SES, 1995.

[17] J. Steinman, *"SPEEDES: A Multiple-Synchronization Environment for Parallel Discrete-Event Simulation,"* International Journal in Computer Simulation, pp. 251-286, 1992.

[18] S.C. Tay, Y.M. Teo and S.T. Kong *"Speculative Parallel Simulation with an Adaptive Throttle Scheme,"* 11th ACM/IEEE/SCS Workshop on Parallel and Distributed Simulation, pp. 116-123, Austria, June 1997.

[19] K. Watkins, *"Discrete Event Simulation in C,"* McGraw-Hill Book Company, 1993.

[20] J. Waldorf and R. Bagrodia, *"MOOSE: A Concurrent Object-Oriented Language for Simulation,"* International Journal in Computer Simulation, Vol. 4(2), pp. 235-257, 1994.

[21] P. Wonnacott and D. Bruce, *"The APOSTLE Simulation Language: Granularity Control and Performance Data,"* Proceedings of 10th Workshop on Parallel and Distributed Simulation, pp. 114-123, USA, May 1996.