Conservative Simulation using Distributed-Shared Memory

Y.M. Teo, Y.K. Ng and B.S.S. Onggo
Department of Computer Science
National University of Singapore
3 Science Drive 2
Singapore 117543
email: teoym@comp.nus.edu.sg

Abstract

This paper focuses on conservative simulation using distributed-shared memory for inter-processor communication. JavaSpaces, a special service of Java Jini, provides a shared persistent memory for simulation message communication among processors. Two benchmark programs written using our SPaDES/Java parallel simulation library are used. The first program is a linear pipeline system representing a loosely-coupled open system. The PHOLD program represents a strongly-connected closed system. Experiments are carried out using a cluster of Pentium II PCs. We used a combination of Wood Turner carrier null, flushing and demand-driven algorithms for null message synchronization. To optimize message communication, we replace SPaDES/Java inter-processor communication implemented using Java's Remote Method Invocation (RMI) with one JavaSpace. For PHOLD (16x16, 16) running on eight processors, this change reduces simulation runtime by more than half, null message overhead reduces by a further 15%, and event rate more than doubled. Based on our memory analysis methodology, the memory cost of null message synchronization for PHOLD is less than 9% of the total memory needed by the simulation.

1. Introduction

Computer simulation is a useful tool to study the behavior of many types of application problems, from military field stratagems in the olden days to commercial and industrial resources to the performance analysis of superscalar processor systems in modern days. Interest in the research domain of parallel discrete-event simulation (PDES) has been fuelled by the notion of exploiting execution parallelism in a number of domains including network design and configuration, personal communication systems, parallel programs, digital battlefields, and digital circuits [2]. The increasing sophistication and scalability of newer systems has motivated researchers to investigate techniques in exploiting event parallelism so as to reduce simulation runtime.

Event synchronization is an essential part of parallel simulation. In general, synchronization protocols can be categorized into two different families: conservative and optimistic. Conservative protocols fundamentally maintain causality in event execution by strictly disallowing the processing of events out of timestamp order. Examples of conservative mechanisms include Chandy, Misra and Byrant's null message (CMB) protocol [7] and the Moving Time Windows [1]. In the past decade, PDES research has developed variants of the null message protocol, with the objective of reducing the high null message overhead. For example, PARSEC [3] attempts to simplify the communication topology in a given problem, and the Cai-Turner carrier-null scheme [6] tries to resolve the problem of transmitting redundant null messages due to low lookahead cycles in digital circuit simulations on up to 24 processors.

This paper discusses a dual approach in improving the performance of conservative simulation using null messages, and presents an alternative measure of the costs of parallel simulation, i.e., in terms of memory usage. With respect to SPaDES/Java [23], we present various optimizations to enhance the efficiency of null message synchronization, emphasizing on the reduction of inter-processor communications overhead through a shared persistent memory architecture implemented using Java-Jini's

JavaSpaces [14] service. Section 2 discusses these various optimizations that we have incorporated in SPaDES/Java. The two benchmarks used are linear pipeline and PHOLD. Section 3 summarizes our experimental results. Section 4 concludes this paper.

2. Performance Issues

SPaDES/Java (Structured Parallel Discrete-event Simulation in Java) is a parallel simulation library [21][23]. It adopts a *process-oriented* world-view whereby entities in the real world are mapped into processes in the conceptual model, and these processes are represented as either logical processes (LPs) or dynamic processes that have a shorter life span than the simulation duration. SPaDES/Java supports both sequential and parallel simulation. Parallel simulation is supported in SPaDES/Java using the null message protocol. Event lists and queues in SPaDES/Java are implemented as binary minheaps [8], which are more efficient than the vectors provided in Java. SPaDES/Java implements LPs as threads, while dynamic processes are modeled as normal Java objects.

This paper addresses two main performance issues: reduce null message synchronization and reduce inter-processor communication among remote LPs with the use of JavaSpaces, with greater focus on the latter. In addition, we analyze the memory cost of conservative simulation.

2.1 Null Message Synchronization

Many implementations of conservative simulation adopt the null message approach, originally developed by Chandy and Misra [7], for the synchronization of simulation events among LPs in parallel simulation. However, the CMB protocol is rather naïve, because each LP at the end of a simulation pass sends null messages to all LPs, even if it is not necessary to do so. The amount of null messages required in a simulation is extremely high even for relatively small problem sizes.

The Wood-Turner carrier-null scheme [24] aims to reduce the null message overhead caused by the existence of cyclic topologies in parallel simulation. The flushing mechanism reduces the growth of null messages by preventing an LP from sending null messages with the same timestamp value to other LPs. When an LP schedules a new null message in its output channel, all null messages with a timestamp value smaller than the new one are flushed [18][20]. Fujimoto proposed an alternative approach to the conventional deadlock avoidance mechanism, by sending null messages on a *demand-driven* basis [3][9]. An LP sends out a null message when it receives a null message *request* from another LP. We combined these three techniques, and implemented in SPaDES/Java an improved *demand-driven null messaging with flushing* algorithm, which we use for our performance study.

2.2 Inter-processor Communication

Inter-processor communication is required for LPs to pass event and null messages. There are two main performance issues in message communication among processors: *message size* and *frequency of transmission*.

We reduce the message size before it is sent and reconstruct the message at the receiving LP. In SPaDES/Java, an event message is a user-defined class, extended from the `SProcess` class. We observe that a large part of the information is globally defined within the simulation environment. Sending only the user-defined portion of the message, which is not inherited from the library, reduces the size of a message and thus reduces the serialization delay caused by the transmission of each message.

2.2.1 RMI and JavaSpaces

SPaDES/Java was originally implemented using Java's Remote Method Invocation (RMI) to serialize and transmit event and null messages across processors. RMI works on a distributed memory architecture, allowing objects, including full code, to be passed between different hosts in the network. However, message serialization incurs a considerable amount of overhead. In simulations (especially those with larger granulariries), the serialization of large messages will cause a bottleneck due to the accumulated delays required for each message to be transmitted across the network. Recently, Sun Microsystems introduces Java Jini, a component-based technology for providing a higher level of abstraction in distributed systems programming [14]. A core Java Jini service, JavaSpaces adopts a distributed-shared memory architecture, providing *distributed data persistence*. Much of the simplicities of Jini systems are enabled by this ability to move code, encapsulated as objects, around the network.

Table 1 summarizes the key differences between RMI-based and Jini-based systems.

JavaSpaces's abstract space operations offer excellent opportunities to reduce inter-processor communication overheads. Instead of having many point-to-point communications, messages are now sent to a centralized memory pool, which coordinates their transfers to the relevant processors. JavaSpaces acts like a medium that reserves a certain abundant capacity of memory solely for inter-processor interactions by matching the attribute types and corresponding values of entry pairs. Technically, each LP is relieved of making remote calls to and interacting directly with a remote LP at the other end of the network, and this can potentially reduce the overall communication latency. RMI, on the other hand, does not guarantee a fixed memory space for remote interactions, and therefore its performance is heavily dependent on the network traffic. To investigate the latency of RMI and JavaSpaces communications, we conducted a Ping Pong simulation involving 1,000 messages being sent around four processors. We observed that the average communication latency for RMI is 10.5 seconds, while that for JavaSpaces is 9.1 seconds. The difference is due mainly to JavaSpaces' dedicated memory for communications.

Jini-based systems	**RMI-based systems**
Lookup directory service	Registry directory service
Facilitates spontaneous (voluntary) computing environment.	Unable to support spontaneous computing environment.
Client interacts with required service via a service object.	Client interacts with remote program by invoking methods via stub.
Network protocol for client interaction is kept transparent from client.	RMI is the sole interaction protocol to be used for communications.

Table 1: Differences between Jini and RMI Systems

Though JavaSpaces can accelerate the pace at which the LPs carry out their simulation over RMI, on the flip side, it can pose a performance problem when the space becomes over-utilized. JavaSpaces is merely a memory pool, which can be exhausted if too many entries are being written to it at any one time. In the extreme case of a fully connected [8] simulation model, each LP is directly connected to every other LP as shown in Figure 1.

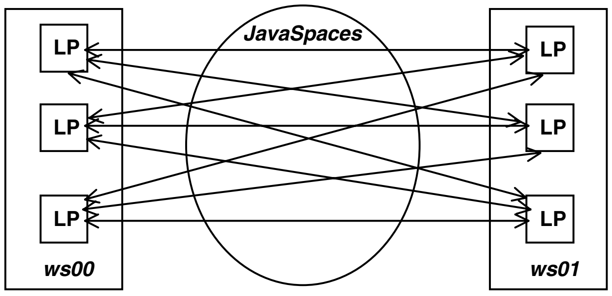

Figure 1: A Fully-connected Simulation Model Distributed on Two Processors

In this scenario all LPs write both events and null-messages into space. The message population in space can increase quickly if the rate at which messages are being transmitted is greater than the rate at which used messages are being removed from space. This results in disk thrashing when the used memory limit of the JavaSpaces has exhausted and the garbage collector does not sufficiently clean up the JavaSpaces. The consequence is a performance bottleneck. However, this is a not threatening problem, because this is subjected to the speed and memory capacity of the processor that maintains the JavaSpaces. RMI, on the other hand, is bounded by the performance of the associated local network.

Figure 2 shows SPaDES/Java distributed simulation communicating through *one* JavaSpaces, with seven LPs mapped onto four processors: ws00 to ws03.

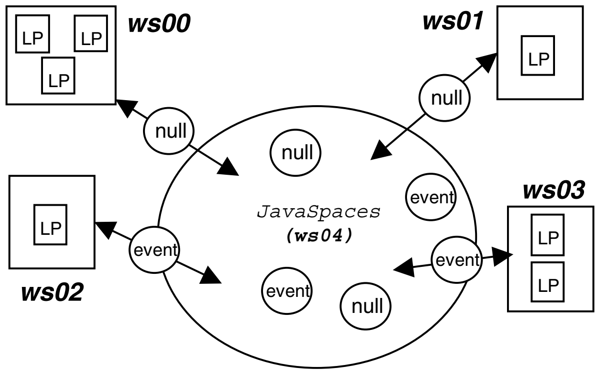

Figure 2: Inter-processor Communication using JavaSpaces

In the implementation, an LP sends a message to another LP residing on a different processor by writing a copy of the message (`write` operation) into the JavaSpaces.

2.2.2 Communicating Event Messages

During the initialization phase of the simulation, an LP invokes the *notify* command on the JavaSpaces to indicate that it is ready to accept event messages from remote LPs. During the simulation, an LP sends an event message to an LP on another processor by depositing the message, tagged with the sender and receiver IDs, into the JavaSpaces. The JavaSpaces will siphon the message to the correct processor based on the value of the receiver ID. Once the message has been delivered, the JavaSpaces will perform garbage collection to remove it from memory, using the `take` operation. Figure 3 shows an example of how transmission of an event message is facilitated by a JavaSpaces.

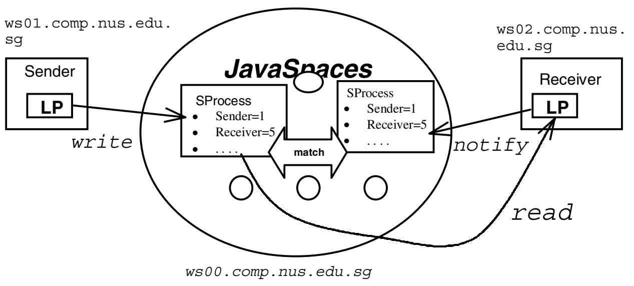

Figure 3: Sending and Receiving Event Messages through a JavaSpaces

2.2.3 Communicating Null Messages

Based on our combined null message algorithm, when an LP demands a null message from a remote sender LP, it places a request, tagged with the sender ID and its own ID, into JavaSpaces via the *notify* operation. The JavaSpaces then notifies the relevant processor containing the sender LP that a null message request has been made to it. Consequently, the sender LP deposits a null message into JavaSpaces, tagged with the requester's ID, and whichever LP that had requested for a null message from this sender can then read off the deposited null message. As with the event messages, all used null messages will be cleared from space. If used null messages are not removed, it will lead to *memory leakage* of the JavaSpaces, which can potentially cause the simulation run to crash.

JavaSpaces fits the analogy of a mailbox that centrally coordinates the distribution of letters, and LPs send letters (messages) to remote LPs by depositing the letters in the mailbox. Also, more than one processor may retrieve a null message that has just being deposited into space with LPs making null message demands from the same sender LP. This attempts to avoid redundancy in transmitting null messages between remote processors.

3. Performance Analysis

The objectives of our experiments are to study how our implementation using the combined null-message algorithm and distributed-shared memory affects overall performance, in terms of null message overhead and simulation runtimes. A cluster of PCs connected via a 100Mbps Ethernet switch and running RedHat Linux 6.0 operating system were used in the experiments. Each node in the cluster is a Pentium II 400 MHz processor with 256MB of memory. Each experimental result is the average over three replications using different random number seed values. The two benchmarks used are a linear pipeline representing an *open* system, and PHOLD [11], which is a *closed* system.

The linear pipeline consists of *n* service centers connected sequentially as shown in Figure 4.

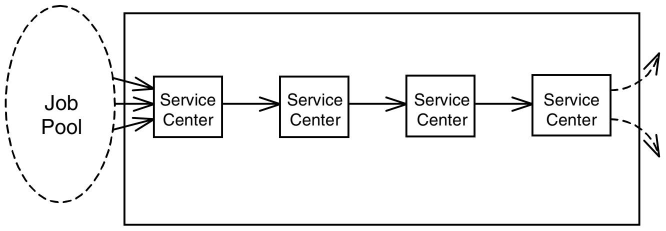

Figure 4: Linear Pipeline (4, ρ)

Job arrival rate is exponentially distributed with a mean of 10 jobs per second. Each service center has a service rate that is exponentially distributed with a mean of 50 jobs per second. The traffic intensity is denoted by ρ. As illustrated above, the linear pipeline comprises of LPs that are *loosely coupled*. The interactions between LPs in the program are minimal, since there are no feedback loops in the topology and hence, message communications proceed in a forward, one-directional manner. It can be used to model open systems such as car wash and supermarket express queues.

PHOLD [11] consists of a network of *N* x *N nodes* and each node is interconnected to four neighboring nodes. Every node is initialized with *m* jobs at the start of simulation. The service time at each node is exponential (0.9) plus a lookahead of 0.1 [11]. Figure 5 shows a 3x3 PHOLD comprising of nine nodes.

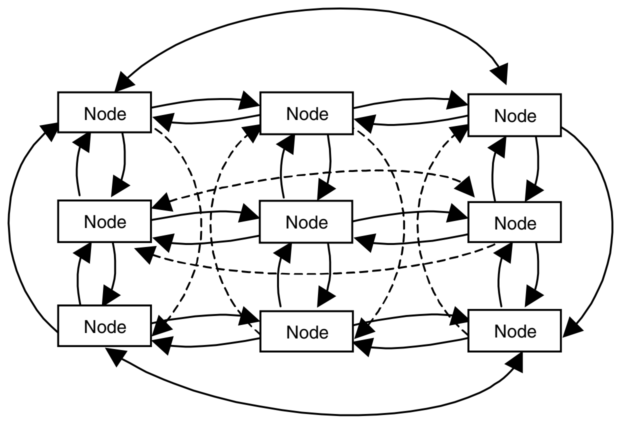

Figure 5: PHOLD (3x3, *m*)

In contrast to the linear pipeline, PHOLD has a strongly connected, *tightly coupled* topology, with many feedback paths. It is one of the most commonly used benchmark for parallel simulation. Linear pipeline and PHOLD are general application problems with two extreme characteristics, and therefore, the simulation results can be applied to many real-world problems.

We define null message ratio (NMR) as the ratio of the *total number of null messages to total messages (event plus null)* in the simulation. We designed our experiments to first investigate the extent to which our combined null message algorithm has improved simulation performance, by executing the parallel simulation on one processor, and comparing the accummulative effects of each optimization step (i.e. carrier null, carrier-null + flushing, carrier-null + flushing + demand-driven null messaging). Next, we vary the number of processors to study the effects of using JavaSpaces and RMI (see

Table 2). "All (4 procs/RMI)" denotes the cumulative effect of the combined null message algorithms on four processors using RMI for inter-processor communication.

In the case of the linear pipeline, cycles are absent in the topology, so the carrier-null NMR overhead is similar to that of the CMB protocol. When the LPs are mapped to one processor and run in parallel, the results show that null message flushing reduces the NMR from 0.94 to about 0.7 and the cumulative effect of flushing, demand-driven null messaging and the use of reduces the NMR further to an average of 0.6. The accumulative effect of the optimizations reduces runtime by more than a half.

For PHOLD, we ran parallel simulations for different problem sizes over simulation duration of 10,000 time units over one, four and eight processors. The reduction in NMR is significant. The combined effect of carrier null, flushing and demand-driven null message further reduces NMR to 0.4 for the case with one processor. Again, the reduction in runtime is more than half for a larger problem size. This illustrates that even on one processor alone, where inter-processor communications are not in play, there are considerable reductions in NMR and achievements in speedup due to the combined null message algorithm. Flushing reduces null message growth from an exponential rate to a linear rate, with the elimination of some proportion of null messages at each LP each time the channels are scanned. The demand-driven mechanism further reduces null messages because now a condition is being laid to invoke the sending of null messages, with the initiative being rested on the destination LP instead of the sender LP. There is no improvement in NMR when the simulation is distributed over four and eight processors respectively using RMI.

Algorithm		Pipeline (16, 0.8)		PHOLD (16x16, 16)	
		NMR	Runtime	NMR	Runtime
1 proc	CMB	0.94	3770	0.99	13990
	+ carrier null	0.94	3775	0.69	5651
	+ flushing	0.70	2917	0.57	4580
	+ demand-driven	0.61	1989	0.44	1563
All (4 procs/RMI)		0.60	1715	0.44	1288
All (8 procs/RMI)		0.59	1578	0.44	1006

Table 2: Null Message Optimizations

#procs	Runtime		NMR		Event rate	
	RMI	JavaSpaces	RMI	JavaSpaces	RMI	JavaSpaces
1	1563	1572	0.44	0.44	49127	48846
4	1288	899	0.44	0.34	59616	85412
8	1006	487	0.44	0.29	76327	157670

Table 3: PHOLD (16x16, 16) – RMI versus JavaSpaces

We observe reductions in NMR and greater speedup when JavaSpaces is used to coordinate the transmission of null messages over more than one processor. For PHOLD(16x6,16) as shown in

Table 3, simulation runtime improves from 1006 seconds (RMI) to 487 seconds (JavaSpaces), NMR reduces from 0.44 (RMI) to 0.29 (JavaSpaces), and event rate doubles. Event rate is defined as the total number of event messages (excluding null) over the simulation runtime.

Figure 6 summarizes the improvement in NMR using JavaSpaces over RMI, for the PHOLD program. The reduction in NMR from using JavaSpaces for inter-processor communication is due to the greater amount of flushing and more efficient null message coordination being done centrally in space. The accompanying decrease in simulation runtimes is caused by the reduction in NMR as well as the lower communications latency involved in JavaSpaces over RMI, as mentioned in Section 2.2.1. The more processors involved in the simulation, the greater the effect of the savings in communications latency.

Teo et. al has developed a methodology based on partial order set theory to study the memory requirement of computer simulation [22]. The methodology defines three components of memory involved. M^{prob} is dependent on the characteristic of the problem and is constant for the same problem size. M^{ord} is dependent on the conservative protocol event ordering but independent of the null message protocol implementation. M^{sync}, on the other hand, is dependent on the implementation of the null message protocol.

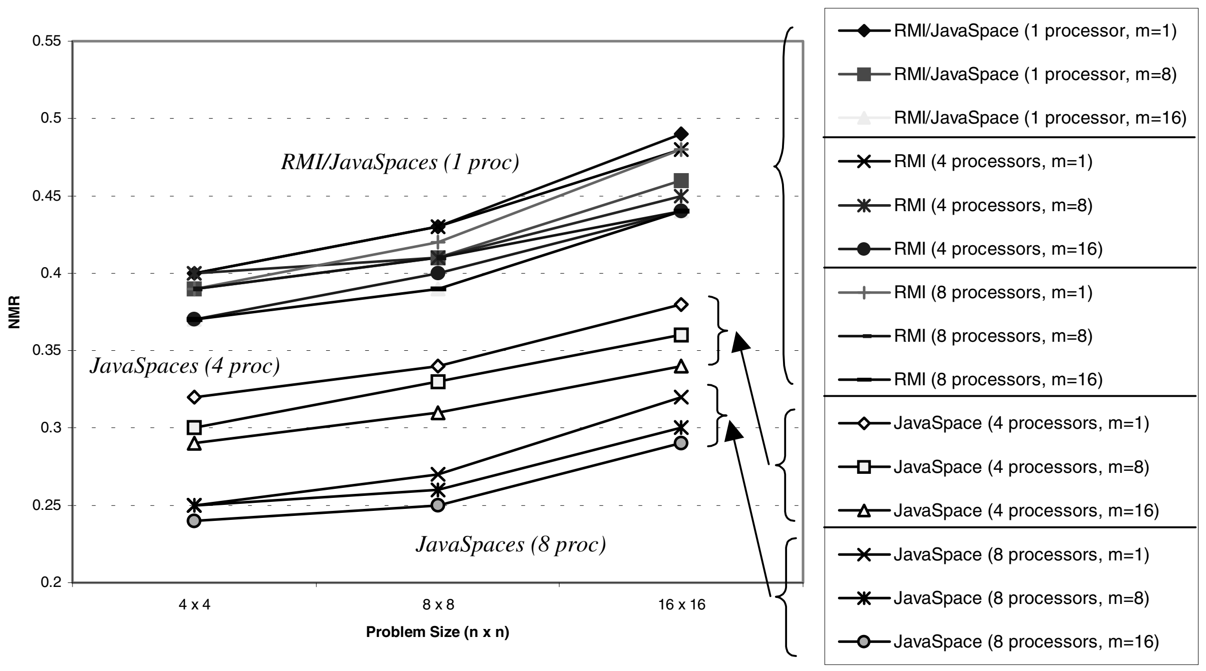

Figure 6: PHOLD (*n*× *n*, *m*) – Varying Number of Processors

For linear pipeline, the flushing algorithm reduces M^{sync} by 36%. Adding demand-driven, null message reduces M^{sync} by 50% with respect to CMB. In terms of the total memory required in a simulation, null message synchronization overhead accounts for 31%. For PHOLD, carrier null message reduces M^{sync} by 45%. Flushing with demand-driven null-message reduces M^{sync} by 65% as compared with CMB, and null-message synchronization overhead accounts for only 13% of the total memory required.

Table 4 compares the memory cost of the demand-driven null message with flushing algorithm for various problem sizes, between using RMI and JavaSpaces as the inter-processor communications protocol on 8 processors.

Space Usage	Pipeline(16, ρ)				PHOLD (16x16, *m*)		
	ρ				*M*		
	0.2	0.4	0.6	0.8	1	8	16
M^{prob}	98	192	320	740	256	2048	4096
M^{ord}	50	52	54	56			
M^{sync} (RMI)	331	341	348	352	665	651	638
M^{sync} (JavaSpaces)	305	308	311	312	347	332	317
M (RMI)	479	585	722	1148	921	2699	4734
M (JavaSpaces)	453	552	685	1108	603	2380	4413

Table 4: Profile of Memory Usage – 8 Processors

The values for M^{sync} above demonstrate that JavaSpaces requires a lesser amount of memory for event synchronization, as compared to RMI, due to JavaSpaces' ability to perform flushing and null message coordination more efficiently. Notice that the values of M^{sync} for PHOLD(16x16, *m*) using JavaSpaces to communicate are approximately half that for the case of RMI.

4. Conclusions and Future Works

The performance of parallel simulation is highly dependent on two major issues, namely the event synchronization overhead, and the cost of inter-processor communications. Inter-processor communication is bound by the network traffic, thereby limiting performance improvements from optimizing the conservative synchronization algorithm to a certain extent. We have briefly discussed three cumulative optimization techniques to reduce null message overhead. This paper focuses on reducing inter-processor communication by comparing RMI with Java-Jini/JavaSpaces. We conducted experiments on two programs of opposite nature: linear pipeline (a loosely-coupled, open system) and PHOLD (a tightly-coupled, closed system), varying the problem sizes and number of processors used. For PHOLD(16x6,16), JavaSpaces reduces the runtime by more than two-thirds, NMR reduces to 0.29, and event rate is almost four times higher using eight processors. We analyze the memory cost of the conservative null message approach based on the memory usage methodology we have developed [22]. We observe that for our optimized null message algorithm using JavaSpaces, the memory cost for supporting null message synchronization (M^{sync}) is less than 9% for PHOLD (16x16, 16) on eight processors.

We are extending SPaDES/Java distributed simulation library to operate on a grid computing environment, where processors are geographically scattered and network latency is significantly larger than on a cluster. To further study the effects of inter-processor communication, we are developing communication-intensive applications such as mobile communication network simulation.

Acknowledgements

This research is supported by the Ministry of Education (Singapore) and PSA Corporation under grants R-252-000-020-112 and R-252-000-020-490.

References

[1] R. Ayani, and H. Rajael, "*Parallel Simulation Using Conservative Time Windows*", Proceedings of the Winter Simulation Conference, pp. 709-717, 1992.

[2] R. L. Bagrodia, "*Perils and Pitfalls of Parallel Discrete-Event Simulation*", Proceedings of the Winter Simulation Conference, pp. 136-143, 1996.

[3] W. L. Bain, and D. S. Scott, "*An Algorithm for Time Synchronization in Distributed Discrete Event Simulation*", Proceedings of the SCS Multiconference on Distributed Simulation, 19, 3 (February), pp. 30-33, 1988.

[4] L. Bajaj, R.Bagrodia, and R. Meyer, "*Case Study: Parallelizing a Sequential Simulation Model*", Proceedings of the 13th Workshop on Parallel and Distributed Simulation, pp. 12-19, 1999.

[5] P. Bizarro, L. M. Silva and J. G. Silva, "*JWarp: A Java Library For Parallel Discrete-Event Simulations*", Proceedings of the ACM Workshop on Java for High-Performance Network Computing, 1998.

[6] W.T. Cai, and S.J. Turner, "*An Algorithm for Distributed Discrete Event Simulation – The Carrier Null Message Approach*", Proceedings of the SCS Multiconference on Distributed Simulation, pp. 3-8, 1990.

[7] K. M. Chandy and J. Misra, "*Distributed Simulation: A case study in design and verification of distributed programs*", IEEE Transactions on Software Engineering, SE-5:5, pp. 440-452, 1979.

[8] Cormen, Leiserson and Rivest, *Introduction to Algorithms*, McGraw Hill, 1989.

[9] A. Ferscha, *Parallel and distributed simulation of discrete event systems*, a chapter in the Handbook of Parallel and Distributed Computing, McGraw-Hill, 1995.

[10] R. M. Fujimoto, "*Parallel Discrete Event Simulation*", Communications of the ACM, vol. 33, pp. 31-52, 1990.

[11] R.M. Fujimoto, "*Performance of Time Warp under Synthetic Workload*", Proceedings of the SCS Multiconference on Distributed Simulation, 22, 1, 1990.

[12] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*, Wiley Series on Parallel and Distributed Computing, Wiley-Interscience, pg. 51-95, 2000.

[13] F. W. Howell, P. E. Heywood, and R. N. Ibbett, "*Hase: A flexible toolset for computer architects*", Computer Journal, vol. 38, pp. 755-764, 1995.

[14] S. Hupfer, *The Nuts and Bolts of Compiling and Running JavaSpaces Programs*, Java Developer Connection, Sun Microsystems, Inc., 2000.

[15] D. R. Jefferson, "*Virtual Time, ACM Transactions on Programming Languages and Systems*", vol. 7, pp. 404-425, 1985.

[16] D. R. Jefferson and H. Sowizral, "*Fast concurrent simulation using the Time Warp mechanism*". Tech. Rep. N-1906-AF, RAND Corporation, 1982.

[17] L. Lamport, "*Time, Clocks, and the Ordering of Events in a Distributed System*", Communications of the ACM, 21, pp. 558-565, 1978.

[18] J. Misra, "*Distributed Discrete Event Simulation*", Proceedings of the ACM Computing Survey, vol. 18, pp. 39-65, 1986.

[19] D. M. Nicol, "*Parallel discrete-event simulation of FCFS stochastic queueing networks*", SIGPLAN Notice, vol. 23, pp. 124-137, 1988.

[20] S. C. Tay, *Parallel Simulation Algorithm and Performance Analysis*, PhD Thesis, Department of Computer Science, National University of Singapore, 1998.