

## Performance Trade-off in Distributed Simulation

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

### Abstract

*This paper extends our previous work on formalizing event orderings using partial order set and its application in space analysis in distributed simulation. We focus on the time and space trade-off in exploiting event parallelism. Event parallelism is divided into inherent (problem) parallelism, event ordering parallelism and effective event parallelism. Firstly, we analyze the performance cost of varying event ordering parallelism on memory requirement in open and closed systems. Secondly, we study the effects of interconnection topology of a physical system on exploitable event ordering parallelism. Measurements were obtained from a time-space analyzer that we have developed.*

### 1. Introduction

Parallel simulation offers the potential to speedup simulations, however the implementation overhead of event synchronization can be high [5]. Much research has been done to analyze the parallelism of a given real-world system. Berry and Jefferson proposed the concept of critical path analysis (CPA) to study the inherent parallelism [1]. Livny implemented a parallelism analyzer based on the CPA using a special simulation language DISS [7]. Lin proposed similar parallelism analyzers [6]. Three CPA algorithms had been developed based on different event scheduling policies. Teo et al proposed an analytical method of analyzing the inherent parallelism using operational laws and mean value analysis [12]. Wong et al proposed an algorithm to predict the memory consumption of specific CMB protocol by measuring the event population list [18].

Different from other parallelism and memory analyses, we focus on the effect of event orderings. A

simulation protocol adopts a specific event ordering and different event orderings produces different degree of parallelism. The objectives of our approach are to derive a tighter upper bound on parallelism and a lower bound on memory requirement, and with the aim of analyzing the parallelism (time) and memory (space) trade-off.

We proposed a methodology to analyze memory requirement in discrete-event simulation in [13][14]. This paper focuses on event parallelism analysis and quantifying the degree of event parallelism. We apply our analysis to four synthetic benchmarks representing different queuing network topologies. The result shows that the amount of event parallelism exploitable by parallel simulation is dependent on the topology of the physical system.

The rest of this paper is organized as follows. Section 2 reviews the simulation modeling process. We formalize our proposed parallelism analysis in section 3. The implementation of our analyzer is presented in section 4. Section 5 discusses our experimental result. We conclude our paper in section 6.

### 2. Memory Requirement

There are three major components in simulation modeling, i.e. *physical system*, *simulation model*, and *implementation*. The real-world system being emulated is called physical system. A simulation model is a logical model of a physical system that defines the input, parameters, output, system states, and other physical system components to be simulated. Subsequently the simulation model is translated into either a sequential or parallel/distributed implementation (simulator program) on a certain execution platform. The physical system and simulation model are independent of the implementation.

Based on the simulation modeling paradigm, we divided the memory required to support a simulation into three main components [13][14], namely, memory to model the states of the physical system ( $M_{prob}$ ), memory required by the event list to schedule event execution based on the selected event ordering ( $M_{ord}$ ), and memory overhead to implement the synchronization protocol on a specific execution platform ( $M_{sync}$ ).  $M_{prob}$  is dependent on the problem size, and  $M_{ord}$  is dependent on the event ordering used. The size of  $M_{sync}$  depends on the efficiency of the implementation. Hence, the total memory requirement of implementing a simulation model on real machines using a certain protocol is  $M_{prob} + M_{ord} + M_{sync}$ .

### 3. Event Parallelism

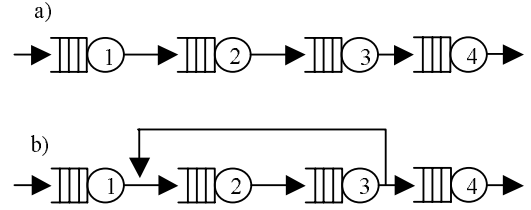
The time analysis of a distributed simulation can also be studied at three different levels, namely: physical system, event ordering, and implementation. In the physical system level, events may happen concurrently. Hence, physical system has parallelism which is called the *inherent event parallelism* ( $\Pi_{prob}$ ). Simulation has the ability to compress time by applying a certain event ordering. Different event ordering exploits different degree of event parallelism. We refer this as *event ordering parallelism* ( $\Pi_{ord}$ ). At the implementation, maintaining a certain event ordering on a specific execution platform requires synchronization overhead, hence the implementation may reduce  $\Pi_{ord}$ . We call this parallelism the *effective event parallelism* ( $\Pi_{sync}$ ).

#### 3.1 Inherent Event Parallelism

Inherent event parallelism ( $\Pi_{prob}$ ) refers to the parallelism that exists in the physical system. In a physical system, some service centers may act independently. This independency suggests that the physical systems have some degrees of parallelism. The degree of parallelism in the physical system depends on the degree of dependency between events.

The interconnections among service centers (topology) in physical system may affect the event dependency. For example, events in service center 3 in the physical system a in figure 1 will never affect events in service center 2, but by adding a feedback as shown in physical system b, events in service center 3 may affect the events in service center 2. Hence, physical system b has a higher degree of

event dependency. Consequently, physical system b has less  $\Pi_{prob}$  than physical system a.



**Figure 1. Effect of Topology on Inherent Event Parallelism**

#### 3.2 Event Ordering Parallelism

Events in the physical system happen chronologically based on the physical time. Physical time refers to the real time in the physical system [5]. An event with a smaller physical time occurs before another event with a larger physical time. Before parallel computer was introduced, sequential simulation must maintain this ordering when it sequentially executes events in order to produce a correct simulation result. This ordering is too strict in a sense that even though events may happen concurrently in the physical system, it will be executed sequentially in the simulation. In the advent of parallel computer, it is possible to use other event ordering which yield correct simulation result, so long as it does not violate the event causality.

Different event orderings introduce different degree of parallelism. The degree of event ordering parallelism depends on the strictness of event ordering rules. Stricter rules restrict the number of concurrent events [13]. Hence, stricter event ordering exploits less event parallelism.

In this paper we focus on four event orderings representing four different degrees of parallelism, i.e. total, timestamp, time-interval and partial event orderings. The main difference among them lies in the definition of concurrent event. This work can be extended to include other event orderings.

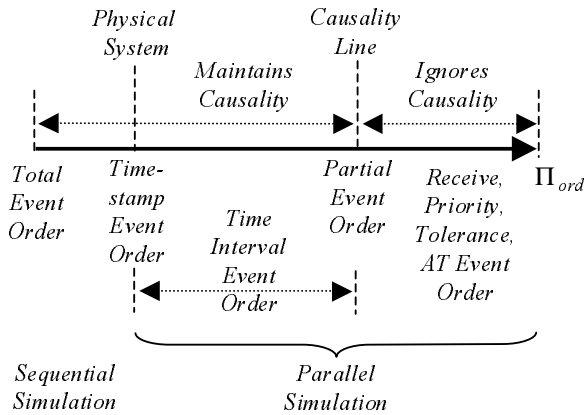
*Total event ordering* executes event sequentially even though events may happen concurrently in the physical system. How total event ordering arranges events in the physical system has been studied in [9][11][17]. Total event ordering does not have any parallelism ( $\Pi_{ord}=1$ ).

Concurrent events in *timestamp event ordering* is defined as the set of events with the same timestamp,

hence they can be executed in parallel in the implementation.

*Time-interval event ordering* provides another alternative to sequence events [13]. Each event is stamped with a time-interval  $[\text{start}(e), \text{end}(e)]$  indicating that event  $e$  may occur at simulation time between  $\text{start}(e)$  and  $\text{end}(e)$ . Event  $e_1$  is executed before  $e_2$  if  $\text{end}(e_1) < \text{start}(e_2)$ . Thus, any events in which their intervals overlap can be executed concurrently. The definition of concurrent event in timestamp event ordering is less strict, therefore on average it has more concurrent events.

The next event ordering is *partial event ordering* [13]. Partial event ordering does not assume any concept of time attached to each event. Instead, partial event ordering arranges events based on event causality only. Event  $e_1$  is executed before  $e_2$  if  $e_1$  causally affects  $e_2$ . Partial event ordering is similar to time-interval event ordering with infinite interval size. When the interval size is infinite, the only information available for arranging events is causal-effect relationship among events. Two or more events are concurrent if they do not causally affect each other. Among the event ordering that maintains the event causality, this is the least strict definition of concurrent event. Therefore partial event ordering produces the largest number of concurrent events.



**Figure 2. Spectrum of Event Orderings and Event Parallelism**

Based on our event ordering approach, we establish the relationship between physical system, different event orderings and simulation implementation as shown in figure 2. All event orderings to the left of causality line (including partial event ordering) maintain event causality and on the other side ignores event causality.

The total event ordering has the least parallelism and as we relaxed the ordering we can exploit more parallelism.

Event orderings that ignore event causality trade-off event parallelism for simulation correctness. Some researchers argued that to exploit more parallelism it is acceptable to sacrifice correctness as long as the error is statistically insignificant [8][10][15]. Receive order [3], priority order [3], approximate time order [4], tolerance order [8] are examples of event ordering that ignores event causality.

### 3.3 Effective Event Parallelism

At the implementation level, additional overhead is necessary to maintain event ordering synchronization across the processors. The overhead affects the event ordering parallelism. For example, to implement a partial event ordering, before executing an event, a processor has to check whether the antecedent of this event has been executed. The antecedent may be on another processor, hence a couple of messages must be sent and received. While the processor is waiting for acknowledgement, it cannot execute any events and remains idle. Hence, it reduces the number of concurrent events and the amount of event ordering parallelism. In addition to this, the execution platform such as the number of processors will also affect the exploitable event parallelism. We called this as the effective event parallelism ( $\Pi_{sync}$ ) of a simulation problem.

$\Pi_{sync}$  is less than or equal to  $\Pi_{ord}$  depending on the efficiency of the implementation. There are too many factors affecting  $\Pi_{sync}$  so that it is very difficult to carry out full factorial experiments. This paper concentrates on the event ordering level, we will discuss the measurement of  $\Pi_{sync}$  in a separate paper.

## 4. Measurement

The main aim of our experiment is to determine the trade-off between parallelism and memory requirement for various event orderings. A simulator is required to generate a set of events. We chose CSIM as the simulator. CSIM is a C simulation library for modeling and developing sequential discrete-event simulations [16]. Next, an analyzer is required to order the set of events according to a certain event ordering and to measure the parallelism ( $\Pi_{ord}$ ) and memory requirement ( $M_{ord}$ ). We developed Time and Space Analyzer (TSA) in C++. It supports the analysis of four

event orderings, i.e. total, timestamp, time-interval and partial event orderings. TSA library can be downloaded from [www.comp.nus.edu.sg/~rpsim/MSG/tsa/index.html](http://www.comp.nus.edu.sg/~rpsim/MSG/tsa/index.html).

Figure 3 shows the main loop in a sequential simulator with TSA instrumentation. The simulator invokes TSA for every event that is removed from the event list (line 5). TSA then simulates different event orderings as detailed in figure 4. Line 6 computes the parallelism and memory requirement.

```

1. While (simulation is in progress)
2.   remove event e with the smallest time
   stamped from event list
3.   simulation_clock = ts(e)
4.   execute_event_handler(e)
5.   TSA(e)
6.   TSA_report()

```

**Figure 3. Sequential Simulation and TSA Instrumentation**

Figure 4 shows TSA executive routine. Assume there are  $n$  logical processes (LP). Each LP maintains three lists, i.e.  $L$  stores the unprocessed events produced by the sequential simulator,  $Q$  stores the events waiting for resources, and  $E$  stores the number of future events scheduled. Initially these lists are empty. Variables  $\max Q$  and  $\max E$  are the maximum size of  $Q$  and  $E$  respectively. Variable  $\text{totPAR}$  measures the average number of concurrent event. Wall-clock time measures the time to execute the simulation program [5].

In line 2, an event  $e$  produced by the sequential simulator (line 5 in figure 3) is stored in the respective unprocessed event list. Line 3 checks if it is possible for TSA to proceed. Because if at least one of the unprocessed event is empty, TSA does not have enough information to decide whether event  $e$  can be executed or not. After enough information has been collected, TSA simulates the event execution by assuming that every event takes a fixed unit of wall-clock time and each LP is mapped onto a processor. Hence the wall-clock time is increased by one unit in line 4.

For each LP, TSA checks whether the antecedent of the earliest unprocessed event (i.e. event  $h$ ) has been executed so as to maintain the event causality. For event orderings other than partial event ordering, TSA also checks whether executing the earliest unprocessed event does not violate any additional rules since a stricter event ordering has more rules.

In line 8,  $h$  has been executed; hence it is removed from  $L$ . The execution of  $h$  may produce a set of events waiting for resources ( $Q'$ ), a set of internal events ( $IE'$ )

and a set of external events ( $EE'$ ). An internal event is an event scheduled by another event on the same LP. An external event is an event scheduled by another event from different LP. Line 9 adds  $Q'$  to  $Q$ ,  $IE'$  to  $E_i$  and  $EE'$  to  $E_j$ . Variable  $\text{busy}$  in line 12 counts the number of concurrent events on each iteration. This variable will be used to measure  $\Pi_{ord}$ .

Line 13 and 14 calculates the new maximum  $Q$  and  $E$  size respectively. If the current total of  $Q$  size for each LP is larger than the previous maximum  $Q$  size, then line 13 will update the maximum  $Q$  size. Line 14 does the same thing for maximum  $E$  size. Line 15 accumulates the number of concurrent events.

Finally, in line 17-19,  $\text{TSA\_report}$  (see also line 6 on figure 3) produces the memory required ( $M_{prob}$  and  $M_{ord}$ ) and the average event parallelism ( $\Pi_{ord}$ ). The  $M_{prob}$  is defined as the maximum queue size and  $M_{ord}$  is defined as the maximum size of event lists [13]. Line 19 calculates the average number of concurrent events by dividing variable  $\text{totPAR}$  with the total wall-clock time required to complete the simulation run.

Initially:  $\forall i L_i, Q_i, E_i = \emptyset$ ;  $\max Q = \max E = 0$ ;  
 $\text{totPAR} = 0$ ;  $\text{wall\_clock} = 0$ ;

```

1. TSA(e)
2.   Li = Li ∪ {e}
3.   if ∃ Li ≠ ∅
4.     busy = 0; wall_clock++;
5.     for each Li
6.       h = head(Li)
7.       if ante(h) has been executed && h does
         not violate event ordering rules
8.         Li = Li - {h}
9.         Qi = Qi ∪ Q'
10.        Ei = Ei ∪ IE'
11.        Ej = Ej ∪ EE', where j=0..(n-1), j≠i
12.        busy++
13.        maxQ = max(maxQ, ∑i |Qi|)
14.        maxE = max(maxE, ∑i |Ei|)
15.        totPAR = totPAR + busy;
16. TSA_report()
17.   Mprob = maxQ;
18.   Mord = maxE;
19.   Pord = totPar / wall_clock;

```

**Figure 4. TSA Executive Routine**

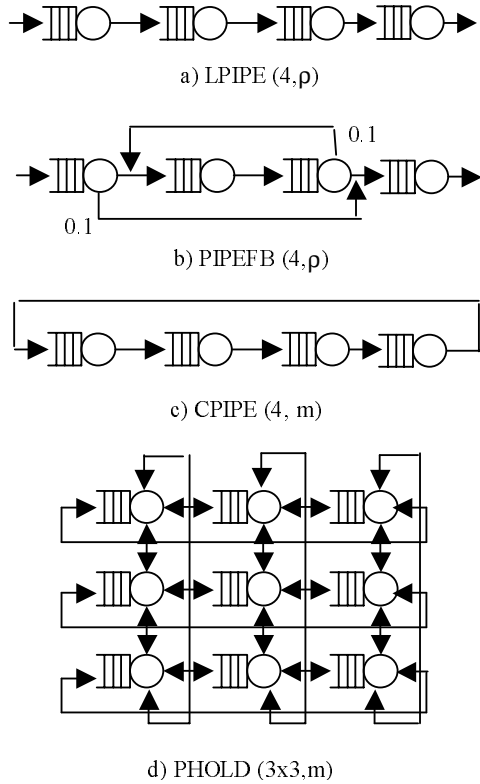
## 5. Experimental Result

We selected and implemented four synthetic benchmarks representing basic queuing network topologies (figure 5).

a) *Linear Pipeline* (LPIPE) is an open system with service centers connected in a chain-like topology.

The parameters are: number of nodes ( $n$ ) and traffic intensity ( $\rho$ ) which is the ratio between arrival rate and service rate.

- b) *Pipeline with Feedback* (PIPEFB) is an open system but with feedback connections among server centers. The parameters are the same as LPIPE.
- c) *Circular Pipeline* (CPIPE) has a ring-like topology representing a (simple) closed system with single feedback. The parameters are the number of nodes ( $n$ ) and job density ( $m$ ) which is the average number of jobs in each node.
- d) PHOLD has a torus topology and is commonly used in parallel simulation studies [2]. It represents a closed system with multiple feedback connections. The parameters are similar to CPIPE.



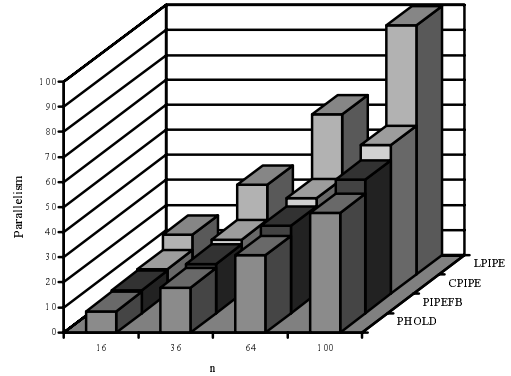
**Figure 5. Synthetic Benchmarks**

### 5.1 Effect of Event Orderings and Topology

The first experiment shows the effects of varying the event ordering on event parallelism ( $\Pi_{ord}$ ). The experiment results are based on an average of three replications, and simulation duration of 10,000 timestamp units per run. For time interval event ordering, we used an interval size of 0.3. The result is summarized in appendix B.

Total event ordering (TOT) corresponds to sequential simulation with  $\Pi_{ord}$  of one.  $\Pi_{ord}$  of time-stamp event ordering (TS) is two. Because the time-stamp of a departure event and its corresponding arrival event on the receiving node has the same timestamp, i.e., no event traveling time between service centers. Time-interval event ordering (TI) with interval size of 0.3 means that any events with time-stamp difference less than 0.3 can be executed concurrently so long as they are not causally dependent. This can exploit more  $\Pi_{ord}$  than time-stamp event ordering. Partial event ordering (PAR) considers only event causality and thus exploits more  $\Pi_{ord}$  than time-interval event ordering.

$\Pi_{ord}$  of timestamp event ordering for all benchmark is approximately the same. However, as we relax the event ordering to a partial event ordering, the level of  $\Pi_{ord}$  exploitable is different. Figure 6 shows the different  $\Pi_{ord}$  exploited by partial event ordering for each benchmark. This shows that different topologies affect the amount of  $\Pi_{ord}$  exploitable by event orderings in parallel simulation



**Figure 6.  $\Pi_{ord}$  - Partial Event Ordering**

The simple topology of LPIPE which is linear without any feedback enables partial event ordering to exploit  $\Pi_{ord}$  of 100. PIPEFB is an LPIPE with one feedback channel. This structure results in the simulation time in the leftmost node (figure 5b) advancing further ahead as compare to the remaining nodes. This node can execute events without any limitation. The node on its right must synchronize its two input channels and the simulation time advancement range between the two channels is very wide. Hence, most of the time it has to wait for the feedback channel to send the next event before it can execute events from its left node. Similar case also happens to the rightmost node. Partial event ordering exploits  $\Pi_{ord}$  of 48.

CPIPE is more complex than LPIPE because it has one feedback loop connecting the last node to the first node. Different from PIPEFB, all nodes in CPIPE has exactly one input channel, hence every events arrived from the input channel can be processed directly.  $\Pi_{ord}$  is around 60. The interwoven feedback channels in PHOLD make it the most complex topology in this experiment. Partial event ordering can exploit  $\Pi_{ord}$  of 48 only. Hence, it can be concluded that feedback channel reduce  $\Pi_{ord}$  that can be exploited by relaxing the event ordering, i.e. the physical system limits the amount of  $\Pi_{ord}$  exploitable by parallel simulation (unless we ignore the causality).

## 5.2 Time and Space Trade-off

Faster implementation may require more memory. Hence, it is important to analyze the trade-off between time and space performance. Figure 7 shows the parallelism ( $\Pi_{ord}$ ) and the memory requirement ( $M_{ord}$ ) of four benchmarks using four event orderings. The number of nodes is 100 and on average 60 events occurs on each node within one timestamp unit.

In general, for open system (LPIPE and PIPEFB), a stricter event ordering requires less  $M_{ord}$ . Teo et al has developed the memory requirement model for each event ordering [13][14]. Detail analysis and analytical proof can be seen from the same paper. In general, less strict event ordering increases the number of scheduled events which have to be stored in the event list. However, contrary to our intuition, relaxing the event ordering in closed system does not increase  $M_{ord}$ , because a stricter event ordering imposes more ordering rules than necessary to maintain event causality. Hence, the parallelism gain can be achieved only by relaxing the event ordering rules. The detail explanation can be found in the appendix A.

The result also shows that PIPEFB requires more  $M_{ord}$  than LPIPE. After decomposing the memory requirement, it turns out that the feedback contributes the biggest portion of memory required by PIPEFB. The node which has more than one input channel has to synchronize all its input channels. Backward feedback widens the simulation time gap between input channels, hence most of the time the node has to wait for the feedback channel to send the next event before it can execute events from other nodes. During which, a lot of memory is required to store the event sent from other channels. This synchronization also reduces parallelism,

hence PIPEFB has less parallelism than LPIPE. Time interval event ordering controls the simulation time advancement. Hence, it can reduce  $M_{ord}$ . However, less  $\Pi_{ord}$  can be exploited.

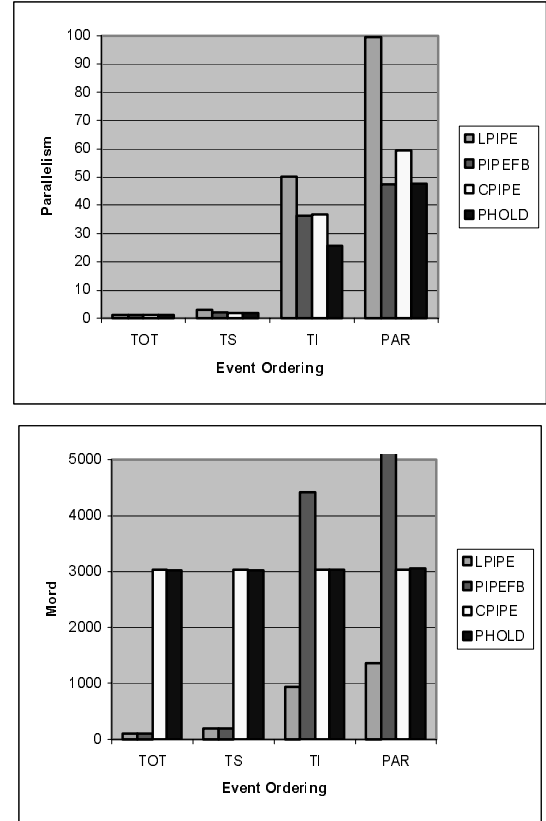


Figure 7. Time ( $\Pi_{ord}$ ) and Space ( $M_{ord}$ ) Trade-off

CPIPE and PHOLD require the same amount of memory  $M_{ord}$ . However, time interval and partial event ordering can exploit more  $\Pi_{ord}$  from CPIPE than PHOLD. Each node in CPIPE has exactly one input channel as compare to PHOLD which has four input channels. Hence, PHOLD has to synchronize its input channels resulting in less parallelism. However, since it is a closed system the memory requirement is constant.

## 6. Concluding Remarks

This paper presented a framework for studying time and space trade-off in distributed simulation. Event parallelism is classified into inherent event parallelism ( $\Pi_{prob}$ ), event ordering parallelism ( $\Pi_{ord}$ ), and effective event parallelism ( $\Pi_{sync}$ ). Sequential

simulation applies a very strict event ordering such that no parallelism is exploited ( $\Pi_{ord} < \Pi_{prob}$ ). Parallel simulation uses less strict event orderings to exploit more parallelism than  $\Pi_{prob}$  so as to achieve speed-up ( $\Pi_{ord} > \Pi_{prob}$ ). The effective event parallelism measures the event parallelism for a simulator implementation. The difference between  $\Pi_{ord}$  and  $\Pi_{sync}$  denotes the implementation overhead including the synchronization to support for a particular event ordering and the cost of the execution platform.

Our experiment shows that for open systems, i.e. LPIPE and PIPEFB, a less strict event ordering exploits more parallelism ( $\Pi_{ord}$ ) but requires more memory ( $M_{ord}$ ). Counter to intuition, for closed systems, i.e. CPIPE and PHOLD, increasing the degree of event parallelism does not require more memory ( $M_{ord}$ ). This is because a stricter event ordering imposes more ordering rules than necessary to maintain event causality, hence relaxing the event ordering can exploit more  $\Pi_{ord}$  without requiring more  $M_{ord}$ .

The experiment using four synthetic benchmarks suggests that the topology of physical system limits the amount of parallelism ( $\Pi_{ord}$ ) exploitable by parallel simulation. The existence of feedback reduces the parallelism of PIPEFB to approximately 50% of the LPIPE's parallelism. Similarly, a more complex feedback structure reduces the parallelism of PHOLD to 80% of the CPIPE's parallelism.

## References

- [1] O. Berry and D. Jefferson, "Critical Path Analysis of Distributed Simulation", *Proc. of SCS Multiconference on Distributed Simulation*, 1985, pp.57-60.
- [2] R.M. Fujimoto, "Performance of Time Warp under Synthetic Workload", *Proc. of the SCS Multiconference on Distributed Simulation*, 22, 1, 1990.
- [3] R.M. Fujimoto and R.M. Weatherly, "Time Management in the DoD High Level Architecture", *Proc. of the 10th Workshop on Parallel and Distributed Simulation*, 1996, pp. 60-67.
- [4] R.M. Fujimoto, "Exploiting Temporal Uncertainty in Parallel and Distributed Simulations", *Proc. of the 13th Workshop on Parallel and Distributed Simulation*, 1999, pp. 46-53.
- [5] Fujimoto, R.M. *Parallel and Distributed Simulation Systems*. John Wiley & Sons, Inc., 2000.
- [6] Y.B. Lin, "Parallelism Analyzers for Parallel Discrete Event Simulation". *ACM Trans. on Modeling and Computer Simulation*, 2 (3), 1992, pp. 239-264.
- [7] M. Livny, "A Study of Parallelism in Distributed Simulation", *Proc. of SCS Multiconference on Distributed Simulation*, 1985, pp.94-98.
- [8] P. Martini, M. Rumeckasten, and J. Tolle, "Tolerant Synchronization for Distributed Simulations of Interconnected Computer Networks", *Proc. of the 11th Workshop on Parallel and Distributed Simulation*, 1997, pp. 138 -141.
- [9] H. Mehl, "A Deterministic Tie-breaking Scheme for Sequential and Distributed Simulation", *Proc. of the 6th Workshop on Parallel and Distributed Simulation*, 1992, pp. 199-200.
- [10] D.M. Rao, N.V. Thondugulam, R. Radhakrishnan, and P.A. Wilsey, "Unsynchronized Parallel Discrete Event Simulation", *Proc. of the Winter Simulation Conference*, vol: 2, 1998, pp. 1563-1570.
- [11] R. Ronngren and M. Liljenstam, "On Event Ordering in Parallel Discrete Event Simulation", *Proc. of 13th Workshop on Parallel and Distributed Simulation*, 1999, pp. 38-45.
- [12] Y.M. Teo, H. Wang, and S.C. Tay, "A Framework of Analyzing Parallel Simulation Performance". *Proc. of the 32nd Annual Simulation Symposium*, 1999, pp. 102-109.
- [13] Y.M. Teo, B.S.S. Onggo, and S.C. Tay, "Effect of Event Orderings on Memory Requirement in Parallel Simulation", *Proc. of the 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2001, pp. 41-48.
- [14] Teo Y.M., and B.S.S. Onggo, *A Methodology for Space Analysis of Discrete-event Simulation*. Technical Report / TRA01, Department of Computer Science, National University of Singapore, June, 2001.
- [15] N.V. Thondugulam, D.M. Rao, R. Radhakrishnan, and P.A. Wilsey, "Relaxing Causal Constraints in PDES", *Proc. of the 13th International and 10th Symposium on Parallel and Distributed Processing*, 1999, pp. 696 -700.
- [16] Watkins, K. *Discrete Event Simulation in C*. McGraw-Hill, 1992.
- [17] F. Wieland, "The Threshold of Event Simultaneity", *Proc. of the 11th Workshop on Parallel and Distributed Simulation*, 1997, pp. 56-59.
- [18] Y.C. Wong, and S.Y. Hwang, "Prediction of Memory Consumption in Conservative Parallel Simulation", *Proc. 9th Workshop on Parallel and Distributed Simulation*, 1995, pp. 199-202.

## Appendix A: Snapshot of TSA Execution

Figure below shows a snapshot of a simple example of two LPs sending a message to each other. A tuple, 1.87 A 00, denotes arrival event 0 occurring at simulation time 1.87. Numbers in the bracket is the event list (FEL) size for LP0 and LP1. Each line represents one wall-

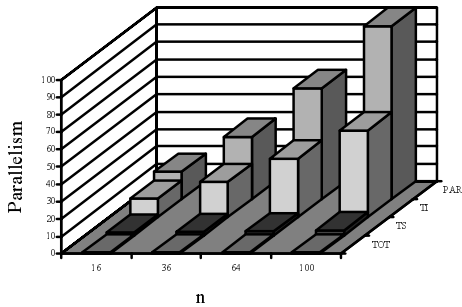
clock time unit. Assume at the beginning, there are five events scheduled for arrivals for each LP. To reach the same state, partial event ordering requires less wall-clock time than time-stamp event ordering. From the number of idle steps (denoted by dashed lines), it can be seen that partial event ordering has more event parallelism than time-stamp event ordering, but the memory required by

both ordering are equal as shown by the FEL size. It is clear from snapshot below that time-stamp event ordering imposes more restrictions. For example at wall-clock time 1 and 2, both events are independent, therefore they could have been executed at the same time as partial event ordering did in wall-clock time 1. This can be done without any additional memory.

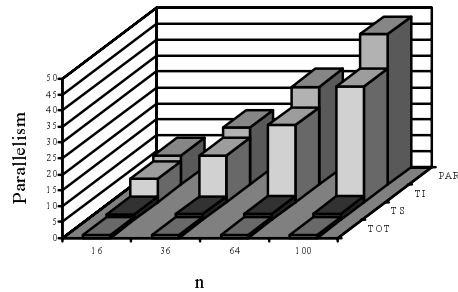
PARTIAL EVENT ORDERING				TIME-STAMP EVENT ORDERING		
WALL_CLK	LP 0	LP 1	FEL	LP 0	LP 1	FEL
1	1.87 A 00	1.25 A 05	{05,05}	-----	1.25 A 05	{05,05}
2	1.89 A 01	2.59 A 06	{04,04}	1.87 A 00	-----	{05,05}
3	2.93 A 02	-----	{03,04}	1.89 A 01	-----	{04,05}
4	3.08 D 00	3.08 A 00	{03,04}	-----	2.59 A 06	{04,04}
5	3.12 A 03	3.13 D 05	{03,04}	2.93 A 02	-----	{03,04}
6	3.13 A 05	-----	{02,04}	3.08 D 00	3.08 A 00	{03,04}
7	3.44 D 01	3.44 A 01	{02,04}	3.12 A 03	-----	{02,04}
8	3.73 A 04	3.74 A 07	{01,03}	3.13 A 05	3.13 D 05	{02,04}
9	4.26 D 02	4.26 A 02	{01,03}	3.44 D 01	3.44 A 01	{02,04}
10	4.46 A 06	4.46 D 06	{01,03}	3.73 A 04	-----	{01,04}
11	4.47 D 03	4.47 A 03	{01,03}	-----	3.74 A 07	{01,03}
12	4.74 D 05	4.55 A 08	{01,03}	4.25 D 02	4.25 A 02	{01,03}
13	-----	4.74 A 05	{01,02}	4.46 A 06	4.46 D 06	{01,03}
14	5.37 A 00	5.37 D 00	{01,02}	4.47 D 03	4.47 A 03	{01,03}
15				-----	4.55 A 08	{01,02}
16				4.74 D 05	4.74 A 05	{01,02}
17				5.37 A 00	5.37 D 00	{01,02}

## Appendix B: Experimental Result

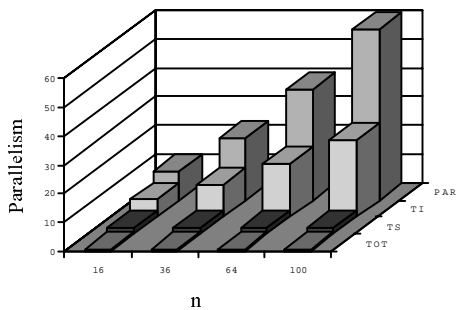
PIPE(n, 0.6)



PIPEFB(n, 0.6)



CPIPE(n, 30)



PHOLD (n, 30)

