Adaptive Techniques for BSP Time Warp

Malcolm Yoke Hean Low
Wolfson College

September 2002

Oxford University Computing Laboratory
Programming Research Group

Thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at the University of Oxford

Acknowledgements

I would like to thank my supervisor, Professor Bill McColl, for his advice and encouragement during the course of this research.

I would also like to thank Professor David Nicol of Dartmouth College for his contribution of ideas and advice to the work presented in Chapter 3.

Finally, I would also like to express my gratitude to the Oxford Supercomputing Center for the use of their parallel computing facilities.

This research is supported by a graduate scholarship from the Singapore Institute of Manufacturing Technology.

Abstract

Parallel simulation is a well developed technique for executing large and complex simulation models in order to obtain simulation output for analysis within an acceptable time frame. The main contribution of this thesis is the development of different adaptive techniques to improve the consistency, performance and resilience of the BSP Time Warp as a general purpose parallel simulation protocol.

We first study the problem of risk hazards in the BSP Time Warp optimistic simulation protocols. Successive refinements to the BSP Time Warp protocol are carried out to eliminate errors in simulation execution due to different risk hazards. We show that these refinements can be incorporated into the BSP Time Warp protocol with minimal performance degradation.

We next propose an adaptive scheme for the BSP Time Warp algorithm that automatically throttles the number of events to be executed per superstep. We show that the scheme, operating in a shared memory environment, can minimize computation load-imbalance and rollback overhead at the expense of incurring higher synchronization cost.

The next contribution of this thesis is the study of different techniques for dynamic load-balancing and process migration for Time Warp on a cluster of workstations. We propose different dynamic load-balancing algorithms for BSP Time Warp that seek to balance both computation workload and communication workload, optimizing lookaheads between processors, as well as manage interruption from external workload.

Finally, we propose an adaptive technique for BSP Time Warp that automatically varies the number of processors used for parallel computation based on the characteristics of the underlying parallel computing platform and the simulation workload.

Contents

Declarations

The following is a list of papers published during my D.Phil. research in Oxford on the field of adaptive techniques for BSP Time Warp optimistic simulation protocol.

- M.Y.H. Low. Adaptive BSP Time Warp. In *Proceedings of the Fifth UK Simulation Society Conference (UKSim 2001)*, pages 14–20, Cambridge, UK, 28-30 March 2001.
- M.Y.H. Low and D.M. Nicol. Consistent Modeling of Distributed Mutual Exclusion Protocol using Optimistic Simulation. In *15th Workshop on Parallel and Distributed Simulation (PADS 2001)*, pages 137–144, Lake Arrowhead, California, 15-18 May 2001.
- M.Y.H. Low. Dynamic Load-Balancing for BSP Time Warp. In *Proceedings of the 35th Annual Simulation Symposium*, pages 267–274, San Diego, California, USA, 14-18 April 2002.
- M.Y.H. Low. Manufacturing Simulation using BSP Time Warp with Variable Number of Processors. In *Proceedings of the 2002 European Simulation Symposium (to appear)*, Dresden, Germany, 23-26 October 2002.
- M.Y.H. Low. Managing External Workload with BSP Time Warp. In *Proceedings of the 2002 Winter Simulation Conference (to appear)*, San Diego, California, USA, 8-11 December 2002.

The following is a list of papers published prior to my D.Phil. research in Oxford. These works were carried out as a joint project between Singapore Institute of Manufacturing Technology and the School of Applied Science, Nanyang Technological University (Singapore). The project studied the use of conservative parallel simulation protocols in the modeling of virtual factories.

- C.-C. Lim, Y-H. Low, W. Cai, W.-J. Hsu, S.-Y. Huang, and S.J. Turner. An Empirical Comparison of Runtime Systems for Conservative Parallel Simulation. In *2nd Workshop on*

Runtime Systems for Parallel Programming (RTSPP 1998), pages 123–134, Orlando, Florida, USA, 30 March 1998.

- S.J. Turner, C.-C. Lim, Y.-H. Low, W. Cai, Hsu W.-J, and S.-Y. Huang. A Methodology for Automating the Parallelization of Manufacturing Simulations. In *12th Workshop on Parallel and Distributed Simulation (PADS'98)*, pages 126–133, Banff, Alberta, Canada, 26-29 May 1998.

- Y.-H. Low, C.-C. Lim, W. Cai, S.-Y. Huang, W.-J. Hsu, S. Jain, and S.J. Turner. Survey of Languages and Runtime Libraries for Parallel Discrete Event Simulation. *Simulation and Transactions of the Society for Computer Simulation (SCS), Joint Special Issue on Parallel and Distributed Simulation*, 72(3):170–186, March 1999.

- C.-C. Lim, Y.-H. Low, B.-P. Gan, S.Jain, W. Cai, S.-Y. Huang, and W.-J. Hsu. Performance Prediction Tools for Parallel Discrete Event Simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation (PADS'99)*, pages 148–155, Atlanta, Georgia, USA, 1-4 May 1999.

- S. Jain, C.-C. Lim, B.-P. Gan, and Y.-H. Low. Criticality of Detailed Modeling in Semiconductor SupplyChain Simulation. In *1999 Winter Simulation Conference (WSC'99)*, pages 888-896, Phoenix, Arizona, USA, 5-8 December 1999.

- Y.-H. Low, B.-P. Gan, S. Jain, W. Cai, W.-J. Hsu, S.-Y. Huang, and S.J. Turner. Parallel Discrete-Event Simulation of a Supply-chain in Semiconductor Industry. In *4th High Performance Computing (HPC) Asia 2000*, Beijing, China, 14-17 May 2000.

List of Figures

List of Tables

Main Abbreviations and Symbols

TW	Time Warp
BSP	Bulk-synchronous parallel model
PDES	Parallel discrete-event simulation
LP	Logical process
DLB	Dynamic load-balancing
GVT	Global virtual time
g, l	BSP parameters
WB	Computation load-imbalance
CB	Communication load-imbalance
n_e	Event limit for each superstep
n_g	Number of supersteps between GVT computations
n_s	Total number of supersteps in a simulation run
P_i	Processor i
$P_i.wl$	Computation workload of processor i
$P_i.cl$	Communication workload of processor i
$P_i.la$	Average system load of processor i
$P_i.wlpo$	Average computation workload per simulation object for processor i
λ	Number of GVT computations between migrations
ϕ	Stability threshold
ϵ	Load-imbalance threshold
η	Migration threshold
θ	Processor load threshold

Chapter 1

Introduction

1.1 Background

Simulation has long been used as a tool in studying complex systems. Over the years, it has slowly transformed itself from a supporting tool to a decision making tool. Simulation is used in many application areas. In the area of defense, simulation is used for both combat planning and troops training [95, 41]. In the area of manufacturing, simulation is used in predicting production throughput and identifying process bottlenecks [47, 48]. In the area of telecommunication, simulation is used for base-station capacity planning and evaluation of new communication protocols [9, 52].

Traditionally, simulation is executed using commercial simulators that run on stand-alone machines. Simulation of very complex and detailed systems can often take hours or even days to complete. With the increasing widespread use of simulation as a decision making tool, a simulation run must deliver results in an acceptable time-frame. A simulation model of a factory that runs for hours is useless to a production planner who needs the simulation output to generate hourly production schedules. As the use of simulation becomes widespread, the problems faced by simulationists increase in size. As the size of the simulation model grows, the limited memory capacity of a stand-alone machine may become too small for any simulation to be carried out.

Parallel Discrete Event Simulation (PDES) could provide solutions to these problems. The combined computing power of multiple processors working concurrently could deliver simulation results in a shorter time-frame as compared to running the simulation sequentially. The combined memory capacity offered by these multiple processors also allows the

simulation of large scale models.

Although PDES has been a well researched topic for many years, the number of application areas have however been limited to defense and telecommunication. Much of the work is still confined within the academic community [31, 75]. Examples of success in deploying PDES in real-world simulation model can be found in [5, 61]. The first paper described a parallel simulation of personal communication services network, while the second paper reported the modeling of a supply-chain for the semiconductor industry using a conservative parallel simulation protocol. However, in each of the case study, explicit domain knowledge and expertise in PDES need to be employed in order to achieve acceptable execution performance.

Very often, the success of a parallel simulation system depends largely on how the simulation workload is partitioned among the set of processors used in the computing environment as well as the workload of individual processor.

Existing studies of parallel simulation protocols have focused mainly on experiments that are conducted within a tightly controlled environment using dedicated parallel and distributed processing stations. Most existing simulation packages use a simple round-robin partitioning or simply rely on users to define the required partition.

However, complex simulation models inherently exhibit irregular workload characteristics that change over time. A good partition for the simulation model at the start of a simulation may become less effective over time as the simulation progresses.

Thus, static partitioning is not effective if a) the workload of a simulation model changes over time throughout a simulation run; or b) the availability of computing resources such as processor cycles and communication bandwidth fluctuate over the lifetime of a simulation run. To maximize the usage of computing resources as well as reducing the time taken to execute a parallel simulation run, the parallel simulation engine used should have a facility for dynamic load management so as to handle the changing requirement of simulation models as well as the dynamic behaviour of the environment the simulation will run in.

However, having dynamic load management in a parallel simulation protocol is still not sufficient to guarantee the successful execution of a simulation run. Optimistic parallel simulation protocols that employ "risk" in their operations suffer from the possibility of "risk hazards". While some of the "risk hazards" could be automatically recovered by the simulation engine itself, others could force the simulation system into an inconsistent state or even crash the system completely.

This is obviously an undesirable behaviour for simulation models that are involved in mission critical operations or require extensive amount of time to execute. A general purpose parallel simulation protocol should have facilities to eliminate "risk hazards" so as to allow consistent and safe execution.

1.2 Research Aim, Objectives and Scope

1.2.1 Aim

The aim of this research is to explore new adaptive techniques to improve the consistency, performance and resilience of BSP Time Warp (BSP-TW) so that it can be used as a general purpose optimistic parallel simulation protocol. To fulfil this aim, the specific objectives involve studying the following research areas:

- Risk Hazards;
- Adaptive Tuning;
- Dynamic Load-balancing;
- Managing External Workload;
- Variable Number of Processors.

1.2.2 Objectives

Risk Hazards

We first study the problem of risk in optimistic simulation protocols, using as example simulation of a distributed mutual exclusion protocol with strong consistency properties. The simulation model is augmented to detect model inconsistency errors resulting from risky optimistic simulation. While the model runs sequentially without consistency errors, errors occur when the model is executed in parallel optimistically using BSP-TW. Some of the errors entirely violate the fundamental mutual exclusion properties of the model itself. To address this problem we extend the BSP-TW protocol to eliminate these inconsistencies. The details of these extensions and the performance trade-off for adding them are discussed in Chapter 3. [60]

Adaptive Tuning

The BSP-TW algorithm aims to reduce the synchronization overhead by adaptively computing upper bounds to the event limit per superstep at regular interval. This allows a simulation run to be completed using the minimum number of supersteps. However, from the BSP cost analysis perspective, synchronization cost is only part of the overall cost of the algorithm. Other overhead such as computation load-imbalance and number of events rolled back are often the dominating factors that need to be considered. This is especially true in a shared memory environment. We discuss the impact of excessive rollback on the performance of the original BSP-TW algorithm and propose an adaptive scheme to the BSP-TW algorithm that automatically determines the number of events to be executed per superstep. The scheme seeks to keep the computation load-imbalance and rollback overhead to a minimum, at the expense of incurring a higher synchronization cost. [56]

Dynamic Load-balancing

To achieve guaranteed performance when running any arbitrary simulation model using a general purpose parallel simulation system, dynamic load-balancing techniques that adapt to the changing states of the simulation model need to be employed.

Different techniques for dynamic load-balancing and process migration for Time Warp are examined. A new dynamic load-balancing algorithm for BSP-TW is proposed. The new algorithm exploits the superstep properties of BSP-TW and is able to balance both computation workload and communication workload, as well as optimize lookaheads between processors. Experimental results on different simulation models show that this new algorithm achieves consistently better performance than the original BSP-TW. [57]

Managing External Workload

Most existing optimistic simulation protocols are designed to run efficiently on a dedicated system. However, in the real-world environment, computing facilities are often shared and jobs from different users of the system tend to compete for CPU cycles.

Using the BSP cost model, we show that a BSP-TW algorithm operating under such a non-dedicated computing environment needs to take into consideration interruption from external workload in order to execute a simulation efficiently.

Two different approaches to managing external workload are considered. The first ap-

proach evicts simulation objects from a processor whenever its average system load exceeds a given threshold. The second approach considers the CPU time-slices given to executing simulation workload on each processor and migrates part of the simulation workload out of those processors that are heavily loaded.

Experimental results show that both approaches give significant performance improvement over the original BSP-TW under different load configurations. [58]

Variable Number of Processors

Even if the computing resources are dedicated and free from any interruption from external workload, there are situations in which using fewer processors than the maximum available can yield better performance. This can happen when a simulation model has very low event granularity, or when the parallel platform used has very high system overhead in terms of communication and synchronization cost.

A refinement to the BSP-TW is carried out to allow the algorithm to dynamically vary the number of processors used for computation during runtime based on factors such as event granularities, event rollback ratio, computation overhead as well as synchronization overhead. Experimental results using the refined algorithm to execute a real-world semiconductor wafer manufacturing model are presented. [59]

1.2.3 Scope

The study carried out in this thesis is limited to the optimistic BSP-TW simulation protocol. This research does not include the study of conservative simulation protocols because in most cases simulation models with zero-lookahead could not be handled or need to be treated differently. We believe that the optimistic simulation protocol is a more viable option as a general purpose parallel simulation protocol.

We have also chosen to explore new adaptive techniques using BSP Time Warp as the target parallel simulation protocol. The reason is that the BSP model on which the BSP Time Warp protocol is based on, provides a general, simple and elegant theoretical framework for designing portable and scalable parallel algorithms [70, 71].

Other parallel programming frameworks that are commonly used for designing parallel simulation protocols include the Parallel Virtual Machine (PVM) [37] and the Message Passing Interface (MPI) [89]. Unlike BSP, none of these frameworks provides a simple

performance cost model to analyze parallel algorithms and very often the only way to evaluate performance of a parallel application developed on these frameworks is through experiment.

1.3 Outline of Thesis

This thesis is organized as follows. Chapter 2 provides some background information on the research areas of PDES. We also describe the original BSP-TW optimistic parallel simulation protocol that will be used in this thesis.

In Chapter 3, we explore the issue of risk hazards in optimistic simulation. Successive refinements to the original BSP-TW protocol are made to eliminate risk hazards from simulation runs.

Chapter 4 describes another refinement to the original BSP-TW. We employ the technique of adaptive tuning to reduce the large number of rollbacks often experienced using BSP-TW as well as keep computation load-imbalance to a minimum. The adaptive tuning is carried out by controlling the number of events each processor can execute in each superstep.

However, the adaptive tuning technique proposed in Chapter 4 is not effective for models that inherently exhibit imbalance in computation as well as communication workload. There are no facilities in the BSP-TW protocol to balance load by migrating objects between processors after the initial partitioning phase.

The study of dynamic load-balancing and load migration algorithm for BSP-TW is described in Chapter 5. We first discuss the different reasons for using dynamic load-balancing and load migration on BSP-TW. This is followed by a review of some research work carried out in this area by other researchers.

We initially focus on the study of dynamic load-balancing and load migration without considering external workload. A dynamic load-balancing cost model for balancing computation and communication workload is derived. Based on the dynamic load-balancing equation, we extend the original BSP-TW algorithm and propose a new BSP-TW DLB_{cc} algorithm that dynamically balances both computation and communication workload. We further extend the algorithm to carry out optimization on lookaheads between processors. The new algorithm is referred to as BSP-TW DLB_{ccl}.

In the second part of the study described in Chapter 6, we examine the effect of external

workload on the performance of simulation models running using the BSP-TW DLB_{ccl} algorithm. In order to better manage interruption from external workload, we further extend the BSP-TW DLB_{ccl} algorithm to account for the changing CPU loads of the processors participating in the computation. Two solutions are proposed. The first solution implemented in BSP-TW DLB_{ccle} protocol dynamically migrates simulation objects out of those heavily loaded processors and discard them from computation until the external workload is removed. The second solution implemented in the BSP-TW DLB_{ccls} protocol uses the concept of time-slicing and migrates only part of the simulation objects out of those heavily loaded processors, thus enabling these processors to still contribute a slice of their CPU cycles towards executing the simulation workload.

Chapter 7 presents another approach to adaptive tuning of the performance of a BSP-TW simulation. The number of processors used in the simulation is dynamically varied during runtime based on a performance cost model. The cost model works by considering the different factors such as event granularities, synchronization and communication overhead as well as the BSP architectural dependent parameters. Experiments are conducted using a real-world semi-conductor wafer manufacturing simulation model on both shared memory and distributed memory systems.

Finally in Chapter 8, we summarize the results achieved in this thesis and outline future work and research directions.

Chapter 2

Parallel Discrete Event Simulation

PDES has been a well researched subject for many years. The principal idea behind PDES is that events in a simulation can be executed concurrently by different processors in the PDES system, but the end results should be identical to that produced by a sequential simulation.

In a sequential simulation, a single event-list would normally be present. Events are extracted from the event-list one after another in time-stamp order to be simulated. New events generated are put into the event-list according to time-stamp order.

In a PDES system, the single event-list is replaced by multiple event-lists. Each event-list is managed by a logical process (LP). An LP in the simulation model represents a physical process in the physical system. LPs are executed in parallel and their simulation times may advance asynchronously. The LPs interact with one another using time-stamped event messages. To preserve the correct time ordering of events (i.e. causality constraint) in a PDES system, each LP must execute all incoming event messages in a non-decreasing time-stamp order using a synchronization protocol.

There are two broad categories of PDES synchronization protocols: conservative and optimistic. Conservative synchronization protocols [6, 8, 13, 73] strictly enforce the causality constraint while optimistic synchronization protocols [36, 49, 50] correct causality violations using rollback.

2.1 Conservative Protocol

In a conservative simulation protocol, an LP will only process an event e with time-stamp t_e if it is guaranteed that no other event with time-stamp smaller than t_e will be received subsequently. Conservative protocols can be further sub-divided into synchronous and asynchronous protocols.

Examples of synchronous conservative protocol include Bounded Lag [63], Conservative Time Windows [2] and YAWNS [74]. The structure of these synchronous algorithms is essentially the same. The protocols operate in a series of steps, each consisting of two phases: synchronization phase and computation phase. In the synchronization phase, a set of events W_i is identified for LP_i such that for every event $e_i \in W_i$, e_i is causally independent from any event $e_j \in W_j$ in LP_j, $j \neq i$. In the computation phase, LP_i executes all events in W_i sequentially.

An example of asynchronous protocol is the original work by Chandy and Misra [13] and Bryant [6] and is often referred to as the Chandy-Misra-Bryant (CMB) protocol. In this case, there is no global synchronization and each LP runs asynchronously and executes events that are "safe" based on information on the time-stamps of the events that are received.

2.1.1 Lookahead

The main drawback of using conservative protocols is the strong reliance on lookahead information between LPs to improve simulation performance. Lookahead LA_{AB} of a link from LP_A to LP_B is defined as the smallest time interval such that an event happening on LP_A can affect LP_B. Specifically, if LP_A is currently at time t_A, then any external event sent out by LP_A subsequently is guaranteed to reach LP_B at a time-stamp greater or equal to t_A+LA_{AB}. To extract lookahead information from a given simulation model often requires the use of model specific information. In the case where lookahead between two LPs is small or even zero, the performance of a conservative protocol degrades considerably. Techniques such as processing time pre-sampling [73] and partitioning [92] can be used to remove this performance bottleneck.

A recent work by Chen and Szymanski [15] introduces a new concept called *lookback*. Lookback is defined as the ability of a logical process to change its past (states) without affecting other logical processes. Specifically, a logical process at simulation time t with

a lookback of l is able to process any received event with time-stamp between t-l and t without sending out anti-messages. The time window [t-l,t] is referred to as the *lookback window* and a *lookback procedure* is used to process events falling into the lookback window. Two lookback-based protocols are proposed and experimental results on the Closed Queueing Network (CQN) simulation are presented. The authors also show that lookback based protocols allow conservative simulation to circumvent the speedup limit imposed by the critical path.

2.1.2 Deadlock Avoidance and Recovery

An asynchronous conservative protocol that uses the CMB protocol is prone to deadlocks. Deadlock avoidance and recovery algorithms are needed to ensure continuous simulation progression. Null-message [72] is one of the methods used to prevent deadlocks. Excessive null-messages can significantly degrade the performance of a conservative protocol. In [8], a special carrier-null-message is used to reduce the number of null-messages and also rapidly advance simulation time of LPs connected in a cycle. In [14], a deadlock detection/recovery mechanism for the CMB protocol that avoids the use of null-messages is also proposed.

2.2 Optimistic Protocol

An optimistic protocol, on the other hand, allows causality constraint to be violated, but corrects it periodically by rolling back the simulation. In this case, each LP optimistically executes events as they arrive. Whenever an event arrives with time-stamp smaller than those executed previously (straggler event), the LP will be rolled back and the state of the LP restored accordingly. Events that have been sent as a result of processing erroneous events have to be cancelled by means of anti-messages.

Optimistic protocol is often known as Time Warp (TW) simulation. TW was first proposed by Jefferson [49] and can also be classified into synchronous and asynchronous. Examples of synchronous TW simulation protocols include SPEEDES [90] and BSP-TW [68]. Asynchronous TW protocols are used in GTW [19] and TWOS [51].

	m^+ received by LP	m^- received by LP

$m.ts \geq$ LVT	**if** $m^- \in$ EL **then**
	discard m^+ and m^-
	else
	insert m^+ in EL
		if $m^+ \in$ EL **then**
		discard m^+ and m^-
		else
		insert m^- in EL

$m.ts <$ LVT	**if** $m^- \in$ EL **then**
	discard m^+ and m^-
	else
	insert m^+ in EL
	Rollback
		if $m^+ \in$ EL **then**
		discard m^+ and m^-
		Rollback
		else
		insert m^- in EL

Table 2.1: Rollback and Message Cancellation Rules in Time Warp: EL=event-list, m^+=normal message, m^-=anti-message, $m.ts$=time-stamp of m and LVT (local virtual time)=time of the last event processed by the LP.

2.2.1 Rollback and State-Saving

To enable rollback of an LP, the LP must keep a record of its states as the simulation progresses. This is often known as state-saving.

Let s_i be the state of an LP before processing event e_i, o_i be the set of events sent out by the LP as a result of executing event e_i, and $e_i.ts$ be the time-stamp of event e_i.

Suppose the LP has executed a series of k events $(e_1, e_2, ..., e_k)$ where $e_1.ts < e_2.ts < ... < e_k.ts$. The set of states recorded by the LP is $(s_1, s_2, ..., s_k)$ and the corresponding sets of events sent out being $(o_1, o_2, ..., o_k)$.

If the LP receives a straggler event e_m such that $e_1.ts < e_m.ts < e_2.ts$, then the state of the LP must be restored to s_2. For each event in the event set $(o_2, o_3, ...o_k)$, a corresponding anti-message is sent to the corresponding destination LP to undo the effect of the event. The LP then re-executes the set of events $(e_m, e_2, ...e_k)$. Table 2.1 shows the rollback and message cancellation rules in a TW protocol.

Different variations of state-saving strategies have been proposed. The straight forward and most commonly adopted method is to save the LP's state prior to executing each event. This is also known as copy state-saving [49]. This scheme is rather costly if events are relatively fine-grained compared to the cost of state-saving.

Two other schemes are also proposed. The first of which is to save the LP's state for every X number of events processed, where X is the state-saving interval. This is also known as sparse state-saving [55]. For example, if sparse state-saving is performed with

$X = 2$ in the above example, only states $(s_1, s_3, ...s_{n-2}, s_n)$ will be recorded. Since the straggler event e_m acts on state s_2, the state of the LP must be restored to s_1 and event e_1 must also be re-simulated in order to obtain state s_2. However, the set of events, o_1, generated as a result of re-processing e_1 need not be transmitted since the purpose of re-processing e_1 is just to restore the state of the LP to s_2.

The second approach is to save only the changes to the state due to each event execution. This is known as incremental state-saving [91]. For example, if $d_{i,j}$ represents the state change from state i to state j, then incremental state-saving saves only the set of state change $(d_{1,2}, d_{2,3}, ...d_{n-1,n})$. To restore state s_2 from a last checkpointed state s_1, it suffices to re-apply only the state change $d_{1,2}$ to s_1.

An interesting approach is reported in [11] whereby the technique of *reverse computation* is used to recover LPs' states. This technique tracks decision points taken in state modifications and attempts to recompute the state of an LP by executing the events in reverse order, executing code that undo the state modification code along the way. For example, if r_i represents the reverse action corresponding to event e_i, then restoring state s_2 involves applying the set of actions $(r_n, r_{n-1}, ...r_2)$ to state s_{n+1}.

2.2.2 Message Cancellation Policy

There are two approaches to sending anti-messages in order to correct erroneous external events. The original approach described in [49] is to simply send a corresponding anti-message for each erroneous external event sent previously. This is also known as the aggressive cancellation policy.

In contrast, lazy cancellation [36] takes a more relaxed approach and attempts to delay the sending of anti-messages. The idea is that re-simulation of a rolled-back event may result in exactly the same set of external events being sent. By delaying the sending of anti-messages until the external events generated differ from the ones recorded by the LPs before rollback, lazy cancellation avoids unnecessary cancelling of correct messages. Using the same example from section 2.2.1, lazy cancellation delays the sending of anti-messages for the output event set $(o_2, o_3, ...o_k)$ until after the re-simulation of each event in $(e_m, e_2, ...e_n)$. If the re-processing of event e_2 generates the same set of output event o_2, then anti-messages for o_2 need not be sent at all.

A variation of lazy cancellation can be applied to state computation as well. If the straggler event e_m does not alter the state of the LP, then state s_m is exactly the same as

state s_2. If event processing is repeatable, the re-processing of event e_2 from state s_2 should yield the same state change to give state s_3. An LP can exploit this fact and attempt to skip over the re-computation phase by comparing its current state with those states recorded before the rollback. This technique is known as lazy re-evaluation [77].

2.2.3 GVT Computation and Fossil Collection

As state-saving and book-keeping of simulated events can consume large amount of memory in a TW simulation, periodic freeing up of obsolete records (fossil collection) is needed so as to recover memory to allow further simulation progress. TW simulation usually incorporates a Global Virtual Time (GVT) algorithm that attempts to estimate the GVT, which represents a lower bound on the time-stamp of the current simulation progress of all the LPs in the systems. GVT is defined as

$$GVT = \min_{\forall i} \{e_i.ts\}$$

where $e_i.ts$ is the time-stamp of all unprocessed or partially processed event e_i. Since no LP can be rolled back beyond GVT, all events e_j, states s_j and output event set o_j with time-stamp smaller than GVT are considered committed and the corresponding memory used can be reclaimed.

Different GVT computation algorithms have been proposed. In [22], a GVT computation scheme that utilizes a centralized GVT manager to periodically compute a new GVT estimate is described. A distributed scheme which uses a distributed snapshot algorithm to approximate GVT is described in [69].

Fossil collection through GVT computation alone cannot prevent memory exhaustion in TW. Techniques such as message sendback [49] and Gafni's protocol [36] allow TW to recover from message exhaustion by freeing up local resources.

2.3 Repeatability and Random Number Generator

A simulation run is said to be repeatable if it produces the same results using the same starting random seed. This should hold true whether the simulation is executed on a sequential machine, a shared-memory machine with 64 CPUs or a PC cluster with 128 nodes.

To achieve repeatable simulation, a requirement is that the random number generator must be able to produce deterministic stream of random numbers given the same initial seed.

Also, the system must be able to obtain an ordering of the simultaneous events in the system so that deterministic tie-breaking is possible. In [94], the author examined the implications of simultaneous events on the logical correctness of a parallel simulation. In [85], some schemes to achieve a well-defined ordering of events and means to identify causally dependent and independent events with identical time-stamps are evaluated.

In [33], Fujimoto proposed a relaxed view of the simulation model, whereby an event is time-stamped with a time interval, rather than an exact time-stamp. The argument is that simulation is only an approximation of the real world, as such there is no single correct ordering of events for any given simulation model. Different event orderings can sometimes lead to the same results. Exploiting this fact could yield more efficient synchronization protocols. The impact of this alternative view to PDES has yet to be fully understood.

2.4 Performance Prediction

Constructing a parallel simulation model has always been known to be a daunting task. Very often, the parallel simulation model constructed could run an order of magnitude slower than a corresponding optimized sequential simulation without going through extensive fine-tuning of various PDES protocol parameters.

Performance prediction has come into focus in recent years. Performance prediction tools have the ability to allow a user to determine the kind of performance improvement to be expected by performing critical path analysis and predicting the parallel performance of a simulation model on a particular parallel simulation protocol [54, 97] using only information from sequential simulation output.

Certain performance prediction tools such as NMap [26] allows an abstract simulation model to be specified and "test run" on different synchronization protocols.

Performance prediction is also useful in predicting the scalability of a simulation protocol using a particular simulation model running on different system configurations. Information such as this is useful in determining the price performance ratio of the parallel system to be purchased.

2.5 Performance Tuning

Any developments on parallel computation systems need to consider the issues of partitioning and load-balancing. These problems are not new to the parallel simulation community and are often tackled using two partitioning approaches: static versus dynamic.

The static approach attempts to extract static information from the simulation model and tries to perform static partitioning of the parallel simulation system. The dynamic approach attempts to monitor run-time behaviour of the simulation system and tries to fine-tune the partition based on some accumulated runtime statistics.

The specific performance problems faced by the parallel simulation community can be classified as follows.

For the conservative protocol:

- small or zero lookahead between LPs;
- excessive number of null-messages;
- overhead in deadlock detection and recovery;
- imbalance of computation and communication load among LPs.

For the optimistic protocol:

- determine the optimal GVT computation frequency;
- determine the optimal state checkpoint frequency;
- management of memory usage;
- optimism control in event execution;
- event cancellation policy;
- imbalance of computation and communication load among LPs.

Various schemes have been proposed to reduce the effort and overhead involved in performing state-saving. In [29], Fujimoto described special purpose hardware that supports incremental state-saving. In [91], a software approach that performs transparent state-saving using constructs in C++ is also described.

Researchers have also devised schemes to deal with the issue of load-balancing. In [96], an automatic static load-balancing algorithm is studied. In [20], a survey of several adaptive performance tuning strategies (mostly based on LP migration) is presented. These dynamic schemes typically make decisions based on runtime statistical data collected. Recent development [27] includes the study of pro-active adaptive algorithm that can react to abrupt changes in both computational and communication resources availability.

With the increasing popularity of symmetric multi-processing machine (SMP) and cluster computing, PDES algorithms that attempt to exploit the strength of these platforms have also been proposed. In [28], a TW protocol that uses shared memory to efficiently perform message-cancellation is described. In another paper [34], an optimized algorithm for GVT computation on shared memory multi-processors is described. TW protocol that is optimized for a cluster of shared memory multi-processors (Clumps) has also been described in [87].

2.6 PDES Language and Library

To ease the learning curve for new users in the field of PDES, languages and libraries have been developed for PDES systems that allow rapid development of a simulation model in PDES. PDES languages and libraries provide users with high-level constructs to program simulation models. Users looking for further optimization in performance will also find compile-time switches or optional parameters that allow fine-tuning of the underlying simulation kernel.

In [62], a survey of several widely used PDES languages (APOSTLE [98], Parsec [3] etc) and libraries (TWOS [51], GTW [19] etc) are presented. The survey compares these languages and libraries in terms of features such as modeling capabilities, programming framework and system environment.

2.7 Visualization

Visualization of a PDES simulation can often help a developer identify performance bottlenecks and model inconsistency. This would allow performance tuning and model debugging to be carried out with ease. Users of a parallel simulation model can also monitor the progress of the simulation visually, allowing easy comprehension of the structure of the

model.

In [12], an experimental visualization environment developed for the PVM [37] network computing system is extended to support visualization of TW execution in a network computing environment.

In [79], a simulator-neutral interactive simulation framework is described. The framework aims to assist users to remotely collaborate and interact with parallel simulation over the Internet as well as view model specific run-time animations.

A prototype performance visualization system for a distributed discrete-event system using a TW protocol on a network of workstations is developed [40]. An evaluation of the system was conducted on a group of users to assess the effectiveness of visualization in identifying performance bottlenecks and aid in fine-tuning the performance of the simulation. The evaluation showed that visualization of the simulation allowed users to quickly locate and isolate performance bottlenecks and make effective changes to system parameters.

2.8 Other Developments in PDES

2.8.1 Updatable Simulation

In a study reported in [25], a techniques called *updatable simulation* which performs multiple execution of a simulation study by updating the results of a prior simulation run rather than re-executing the entire simulation. This technique requires the states of the simulation objects and the corresponding events be logged in the first (primary) run of the simulation study. Subsequent runs with different initial starting conditions or input parameters can employ a *reuse procedure* to determine which events in the primary run can be reused so that re-computation can be avoided. The success of this approach relies on the extent of similarities between the histories of two simulation runs with only a small change in their initial state or input parameters. Experimental results using this technique on a sequential and parallel packet level ATM multiplexer simulation shows that substantial reduction in the time required to complete multiple simulation runs can be achieved.

2.8.2 Simulation Cloning

A technique known as *simulation cloning* is first proposed in [44] where a running parallel discrete event simulation is dynamically cloned at decision points to explore different execution paths concurrently. This allows what-if and alternative scenario analysis to be carried out interactively or non-interactively. Experimental results on the PHold and a personal communication service (PCS) network model show that the technique is efficient in application where the effects of a message introduced by a logical process do not spread rapidly to other logical process. The work is further extended in [45] to examine the scalability of simulation cloning in detail. The experimental results reported indicate that cloning scales with the size of simulation in terms of the number of logical processes and the number of messages generated.

2.8.3 High Level Architecture

An interesting development in recent years is the United States Department of Defense (DoD) High Level Architecture (HLA) [17] effort to "provide a specification of a common technical architecture for use across all classes of simulations". In particular, the Run-time Infrastructure (RTI) component of the HLA provides a distributed operating system to support simulation interaction and management. A synchronization mechanism for event ordering is provided in the time management component of the HLA [35]. By expanding its services to include repeatable execution and ordering of simultaneous events [32], HLA provides a framework in which research in the PDES community can be readily applied on real-world systems.

2.9 SimBSP and BSP Time Warp

The bulk-synchronous parallel (BSP) model [93] was developed to be a general purpose approach to parallel computing. It has features such as simple programming interfaces, scalable performance and a simple cost model for performance prediction. A BSP model comprises a set of P processor-memory pairs that communicate with each other through messages using an interconnection network. A BSP program progresses in a series of supersteps, each ending with a barrier synchronization of all the processors. In each superstep, each processor performs computation on locally held data and sends messages to other processors. The messages sent in a superstep will only be available to the receiving

```
bsp_begin();
[A] Initialization
while GVT < SimEndTime
    [B] Receive external events and process rollback
    [C] Compute new GVT, perform fossil collection and compute new event limit n_e
        every n_g supersteps
    [D] Execute n_e events
    bsp_sync();
endwhile
bsp_end();
```

Figure 2.1: Algorithm for BSP Time Warp.

processors in the next superstep.

The SimBSP parallel simulation library is a set of APIs that allows easy construction of simulation models through object inheritance. The SimBSP parallel simulation library is based on the BSP Time Warp protocol described in [68].

The BSP Time Warp (BSP-TW) protocol is described in [68]. The algorithm for BSP-TW is shown in Figure 2.1. Each processor manages a group of logical processes (LPs) in the system. LP and simulation object are used interchangeably in this thesis. LPs in the same processor share a common event-list. Each processor executes a series of supersteps as indicated by the outer `while` loop and the `bsp_sync()` statement at the end of the loop.

An estimate of the global virtual time (GVT) is computed after every n_g supersteps, where n_g is a system defined constant. Memory for events or states in an LP with time-stamps smaller than GVT are reclaimed (fossil collected) after each GVT computation. Each processor executes the body of the loop till its GVT value is greater than the simulation end time. The BSP-TW protocol can be characterized by the following attributes:

Sequential Simulation

Events received externally from other LPs or generated internally by local LPs are kept in a sequential event-list. The BSP-TW protocol defines an adaptive upper limit, n_e, to control the number of events to be simulated per processor per superstep. This limit makes no distinction between newly received events and rolled-back events. The n_e events are taken from the sequential event-list and simulated sequentially.

Controlled Optimism

Since the event limit is imposed with no distinction between newly received events and rolled-back events, and the fact that a sequential event-list is used to process events chronologically, a form of controlled optimism on event execution is imposed on the processor. This has the effect of reducing the length and number of rollbacks that can take place. A processor that proceeds faster in simulation time than other processors in one superstep will find itself processing proportionally more rolled-back events in the subsequent supersteps, thus slowing down its progress relative to other processors.

Group Rollback

Each processor receives events from other processors at the start of each superstep. Some of these messages are straggler events (events with time-stamp smaller than the simulation time of the receiving LPs) and rollback of LPs will occur. As the stragglers are received, the respective LPs are put into a rollback priority queue that registers the earliest time of each rollback. The LPs are then rolled back by first processing the one with the earliest rollback time.

After all the external events have been received, each processor proceeds to simulate the least time-stamped event. Further rollbacks may take place at this stage and the similar group rollback mechanism will be used.

Silent GVT Calculation

GVT calculation is carried out periodically after every fixed number of supersteps. While other TW systems use dedicated synchronization steps to compute GVT, the GVT computation in the BSP TW protocol is carried out silently using the event processing steps without pausing the simulation.

Adaptive Superstep Event Limit

After each new GVT estimate is computed, the processor performs fossil collection to reclaim all unused memory space. At the same time, a new event limit per superstep, n_e, is computed.

The number of events that can be executed in the current superstep is set to this up-

per limit of n_e, with no distinction between new events and rolled-back events. Each LP optimistically executes n_e events from its event-list, and sends newly generated events or anti-messages to other LPs.

To adaptively determine the optimal number of events to be simulated in each superstep, each LP_i in the simulation system is tagged with a superstep counter $LP_i.SStep$. Each event e_i in the system is also tagged with a superstep counter $e_i.SStep$ indicating the earliest superstep at which the event may take place.

For each new event e_i sent by LP_i, if the event is destined for another LP in the same processor, $e_i.SStep$ is set to $LP_i.SStep$, since the event could potentially be simulated in the same superstep. If the event is destined for an external LP, $e_i.SStep$ is set to $LP_i.SStep$+1, as the external LP will only receive the event in the next superstep.

The superstep counter of each LP is initialized to 0 at the start of the simulation. For each event e_i received and executed on LP_i, $LP_i.SStep$ is updated to the maximum of $LP_i.SStep$ and $e_i.SStep$.

The superstep counter of an LP is considered part of its state and is state-saved accordingly. The new estimate to the optimal number of events to be simulated per superstep, $n_e(i)$, at GVT computation interval i in a processor with m LPs is determined using information from fossil collected events at each fossil collection phase using the following formula:

$$n_e(i) = \frac{C_i - C_{i-1}}{\max\limits_{\forall j}\{LP_{j,i}.SStep\} - \max\limits_{\forall j}\{LP_{j,i-1}.SStep\}}$$

where C_i is the accumulated total number of events committed in processor P in GVT computation interval i and $LP_{j,i}$, $1 \leq j \leq m$, is the superstep counter of LP_j just before the start of the GVT computation interval i.

2.9.1 Repeatability of Simulation

To achieve repeatable simulation results in an optimistic simulation engine, two important aspects of the simulation engine must be implemented carefully. The first involves random number generators, and the second involves the treatment of simultaneous events.

Random Number Generator

Given certain pre-defined initial random number seeds, a correct sequence of events in a simulation must obtain the same random number streams, no matter how many times the simulation model are rolled back in the course of simulation, so as to achieve repeatable simulation results.

This requires that the random number streams used in the simulation model be part of the LP's state and be state-saved accordingly. In the SimBSP library, each LP has an associated random number generator as part of its state. The random number generator is initialized using the ID of the LP and is state-saved accordingly during simulation run.

The state of a random number generator is often large and re-computing the random numbers in the re-simulation of events can incur high computation overhead. In the SimBSP implementation, all random numbers generated by an LP are stored in a list. The state of an LP corresponding to the random number generator is represented by an integer index on the list. Rollback of a random number generator involves only retrieving the random number at the corresponding index in the list. Old random numbers are freed from the list during fossil collection. This implementation reduces the state of the random number generator to the size of an integer, as well as eliminating the need for unnecessary re-computation of random numbers during the re-simulation of events.

Simultaneous Events and Zero Lookahead

To achieve repeatable simulation results, the simulation engine must be able to construct a total ordering of all simultaneous events. The event ordering heuristic provides a solution by enforcing a total ordering among simultaneous events. However, special care needs to be taken when simultaneous events are generated by external LPs, resulting in the problem of zero lookahead.

Zero lookahead in a simulation model arises when an event in one LP generates another event with the same time-stamp in another LP and the two LPs lie in different processors.

Consider the following rollback scenario between LP_i and LP_j. At superstep n, LP_i simulates event e_1 with time-stamp t and generates an event e_2 at the same time-stamp for LP_j. In the same superstep n, LP_j simulates event e_3 also with time-stamp t. In the next superstep $n+1$, event e_2 arrives at LP_j. As most TW protocols keep track of only the simulation time of the last event executed, LP_j has no way of knowing if event e_2 is a

straggler relative to event e_3 without actually rolling back e_3.

If however, the necessary information of the last event executed are kept by LP_j to allow it to deduce that event e_2 "happen after" event e_3, then no rollback will take place. This technique could be used to eliminate unnecessary rollback of simultaneous events and reduce the amount of rollback trashing between LPs.

2.10 Summary

In this chapter, we reviewed the current research focus in the field of PDES. The major issues in both the conservative and optimistic parallel simulation protocols are discussed. We also highlighted the different aspects of parallel simulation systems such as languages, libraries and visualization that will enable a wider adoption of the parallel simulation technique.

The SimBSP library and the BSP-TW protocol have also been described. While the BSP-TW protocol aims to be a general purpose optimistic simulation protocol, it does not address problems such as risk hazards, load-imbalance in simulation model and load-imbalance in computing resources. These issues will be further examined in subsequent chapters.

Chapter 3

Risk Hazards

3.1 Introduction

Optimistic simulation protocols use *aggressiveness* and *risk* to exploit parallelism in models. As described by Reynolds [83], *aggressiveness* allows events to be executed speculatively, while *risk* allows possibly incorrect state resulting from speculative execution to be exported from one processor to another. It is possible to execute aggressively but without risk, as demonstrated by the Breathing Time Bucket protocol [90]. Most optimistic systems, such as GTW [19] employ both *risk* and *aggressiveness*.

Potential hazards of *risk* in optimistic simulation protocols were examined in [76] by Nicol and Liu. They argue that optimistic simulation systems built by linking a library to native language source code are vulnerable to risk, in that execution behavior which is perfectly legitimate in an optimistic simulation can crash a simulation or silently corrupt the simulation state. They go on to identify two hazards of *risk* with respect to consistency, "rollback inconsistency" and "stale state". Rollback inconsistency occurs when "a completed rollback makes two sampled states inconsistent", and stale state occurs when "an initiated but as yet uncompleted rollback chain makes two sampled states inconsistent". The paper proves that both hazards can be eliminated by requiring a message acknowledgement protocol both for normal event messages and anti-messages.

Nicol and Liu neither give examples of how the consistency errors they describe could occur in a semantically consistent model, nor do they study implementation problems that might arise in implementing their solutions.

In this chapter, we do both of these things. Our objectives are to (a) verify that a seman-

tically consistent model such as the modeling of distributed mutual exclusion protocol can exhibit inconsistent behaviors when executed using an optimistic simulation protocol such as BSP-TW; (b) describe various extensions to the BSP-TW protocol to eliminate rollback inconsistency and stale state; (c) determine the trade-off between performance and consistency in the protocol.

The rest of this chapter is organized as follows. We first describe the concept of rollback inconsistency and stale state, as well as the solution proposed in [76] to solve both these issues in section 3.2. Section 3.3 gives a brief description of the distributed mutual exclusion model used in this study, together with a description of various errors that could arise in an optimistic simulation and are semantically inconsistent with the model itself.

Section 3.4 explains how errors due to rollback inconsistency can occur in the BSP-TW optimistic simulation. In section 3.5, we extend the BSP-TW with explicit anti-message in order to eliminate rollback inconsistency. In section 3.6, we explore the use of extended barrier in BSP-TW to achieve implicit anti-message acknowledgement. The BSP-TW is further extended with normal message acknowledgement in section 3.7 in order to eliminate stale states. A proof for the correctness of BSP-TW with extended barrier is presented in section 3.8. Section 3.9 highlights some related work and section 3.10 summarizes the work carried out in this chapter.

3.2 Rollback Inconsistency and Stale State

In this section, we describe the concepts of rollback inconsistency and stale state in the context of optimistic simulation that uses risk. We also present the solution proposed in [76] to solve both these risk hazards.

We first associate with each LP in the simulation model a dependency vector (DV) of length n, where n is the number of LPs in the simulation model. The j component of DV for LP_i denotes the latest state of LP_j known to LP_i. Each of the j component of DV consists of a time-stamp in which the state of LP_j is sampled, followed by a list of time-stamp in which LP_j was rolled back prior to sampling the state. Each external message sent by an LP is tagged with the LP's DV.

Suppose LP_i receives a message m from another LP. Each j component of the DV of LP_i is compared with the corresponding j component of the DV of message m.

Let the j component of the DV of LP_i be (t_1, L_1) and the j component of the DV of m

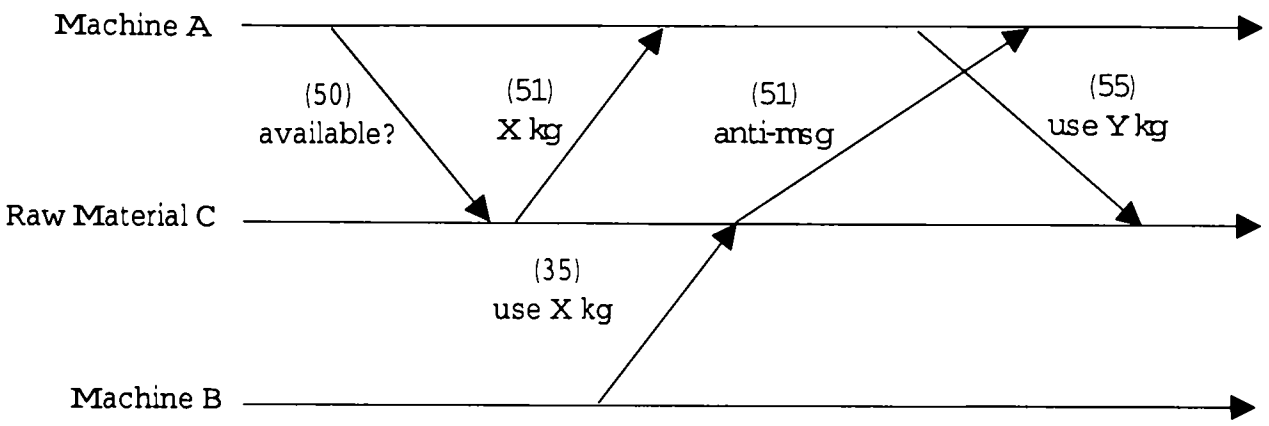


Figure 3.1: An Example of Rollback Inconsistency.

be (t_2, L_2) respectively, where t_1 and t_2 are the simulation times in which the state of LP_j are sampled and L_1 and L_2 are ordered lists of rolled-back times of LP_j.

If L_1 and L_2 are of the same length and $t_1 < t_2$, then the j component of the DV of LP_i is updated with the DV from message m. Otherwise, nothing needs to be done as the j component of the DV from message m reflects an older version of the state of LP_j compared to the DV in LP_i.

If L_1 and L_2 are of different lengths, then one of the lists is necessarily a proper subset of the other. Suppose L_1=(L_2, L_1'). If there exists a rolled-back time t_3 in L_1' such that $t_3 < t_2$, then the two DVs are said to be rollback inconsistent. The state of LP_j sampled at t_2 was made invalid by the rollback at t_3, but the effect of the rollback has not yet affected LP_i. If t_2 is smaller than all the rolled-back time in L_1', then the state sampled at t_2 was not affected by any of the rollbacks that happened subsequently and the two DVs are said to be rollback consistent.

Stale state occurs when an initiated but as yet uncompleted rollback makes two sampled states inconsistent. Suppose LP_j sends out a message m_1 to LP_k, which initiated a rollback at LP_k that will ultimately roll back another LP_i. Immediately after sending out message m_1, LP_j sends another message m_2 to LP_i. Upon processing message m_2, the j component of the DV of LP_i is stale since the message m_2 processed by LP_i is destined to be rolled back due to the transmission of message m_1 by LP_j.

Figure 3.1 illustrates a scenario in which rollback inconsistency occurs. Suppose machines A and B are both using a shared pool of raw material C. At time 50, machine A sends a query message to check how much raw material is available. C sends a reply at time 51 stating that X kg of raw material is left.

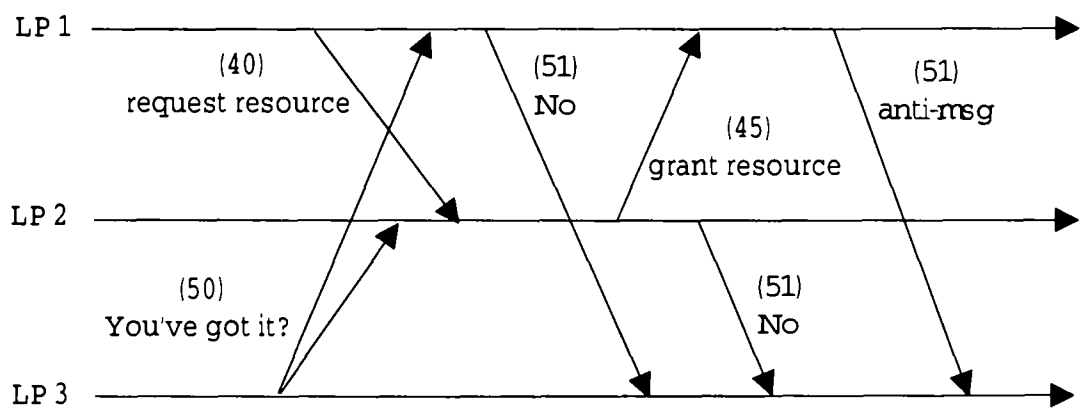


Figure 3.2: An Example of Stale State.

However, if machine B has sent a message to acquire X kg of raw material at time 35 but the message did not arrive until after raw material C has sent the reply to A; this forces C to be rolled back and an anti-message to be sent to A for the reply message sent at time 51. Immediately after processing the message from machine B, C has no raw material.

Before the anti-message arrives at machine A, A acts on C's reply and immediately sends a message to C to use Y kg of raw material. Since C has no more raw material left, the message from A is inconsistent to its current state.

Figure 3.2 shows another example where stale state occurs. Suppose a shared resource is being held at any one time by one of the three LPs. Originally the resource is held by LP_2. At time 40, LP_1 sends a request to LP_2 to ask for the resource. This request is granted by LP_2 at time 45.

At time 50, LP_3 sends out two messages to check which LP is holding the resource. Since LP_1 is still waiting for the reply from LP_2, it replied immediately that it did not have the resource.

Since LP_2 sends out the message granting the resource to LP_1, it also replied to LP_3 that it did not have the resource. However, the message from LP_2 to LP_1 is destined to roll back LP_1 and ultimately triggers a rollback in LP_3. Thus the state of LP_3 is stale with respect to the reply message received from LP_2.

As described in [76], rollback inconsistency errors can be eliminated simply by acknowledging anti-messages. By further acknowledging ordinary messages, the simulation will be free from stale state. In the case of anti-message acknowledgement, an LP receiving an anti-message, e_i, will acknowledge the anti-message if no rollback occurs or no anti-messages are sent as a result of any rollback. If some anti-messages are sent, then

the LP needs to wait for the acknowledgement of all the anti-messages before sending the acknowledgement for e_i.

Similarly for normal message acknowledgement, an LP receiving a normal message e_i will immediately send an acknowledgement for e_i if no rollback is triggered or no anti-messages are sent as a result of any rollback. If e_i triggers a rollback and some anti-messages are sent as a result, then all the acknowledgement of the anti-messages must be received before the acknowledgement for e_i can be sent.

Consider the same example in which LP_i receives a message m from another LP. Assume that the j component of LP_i's DV is rollback inconsistent with respect to the j component of m's DV. If LP_i's DV is (t_1, L_1) and m's DV is (t_2, L_2) and assume that $L_1 = (L_2, L_1')$, the two DVs can only be rollback inconsistent if there exist a rolled-back time t_3 in L_1' such that $t_3 < t_2$. This means that after sending out a message at time t_2, LP_j is rolled back to time t_3 and another message time-stamped at t_1 is sent at some point in simulation time after the rollback.

If LP_i is required to wait for anti-message acknowledgement, the complete causal chain of events (including message m) originating from the transmission of the event at time t_2 by LP_j would have to be cancelled before LP_j can send out the message time-stamped at t_1. Thus rollback inconsistency cannot occur with anti-message acknowledgement.

Similarly, we consider the case for the occurrence of stale state. Using the example for stale state described above, suppose LP_j waits for normal message acknowledgement before sending out another message. Following the transmission of message m_1 to LP_k, LP_j will not send out message m_2 until an acknowledgement for m_1 is received from LP_k. Upon receiving m_1 from LP_j, LP_k is rolled back and proceeds to send out anti-messages that will eventually roll back LP_i. The acknowledgement for message m_1 will only be sent back to LP_j when all the acknowledgements for anti-messages are received by LP_k. LP_i will thus be rolled back before message m_2 is transmitted by LP_j.

3.3 Distributed Mutual Exclusion Model

The distributed mutual exclusion model used in this study consists of an n by n grid with each element in the grid being a node or a resource. During the course of the simulation, a node will acquire a resource for a certain period of time. A "radius-of-usage" parameter, r, is defined for the resources in the simulation model to restrict the set of nodes a resource

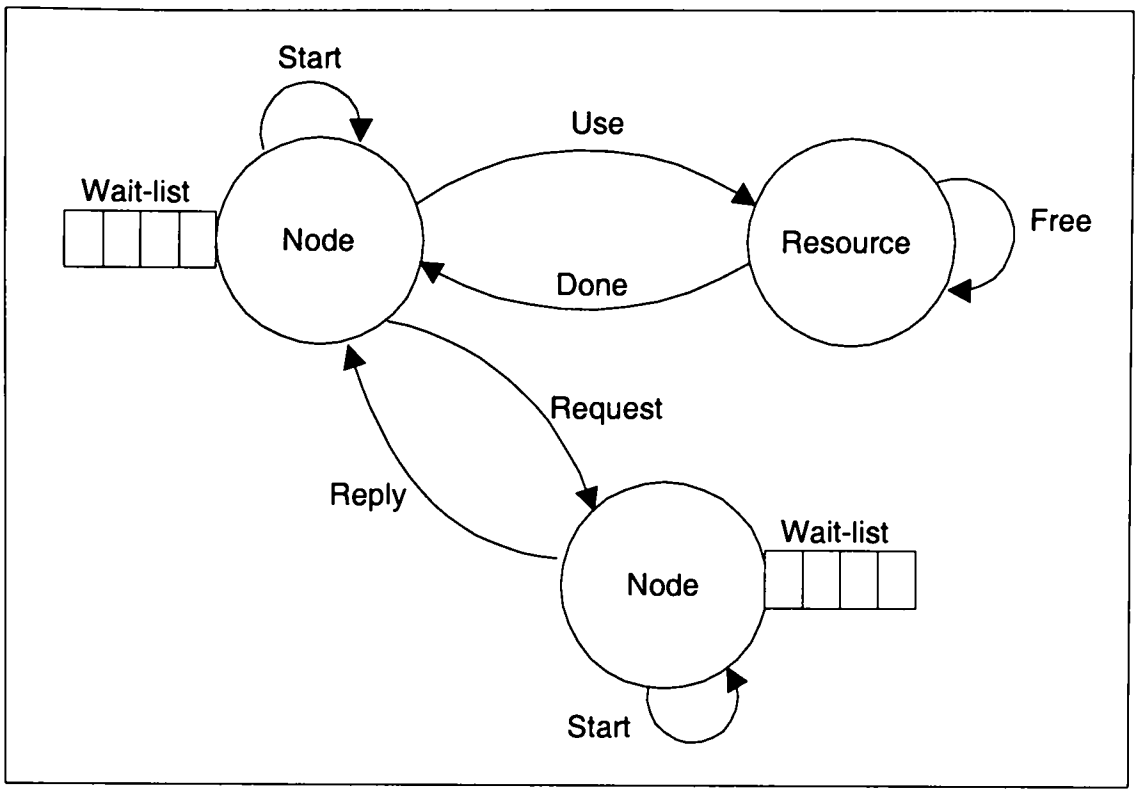


Figure 3.3: Events Relationship between Node and Resource in a Distributed Mutual Exclusion Model.

will serve. A resource R_i at grid coordinate (x,y) will service any node N_i with grid coordinate (s,t) if $|s - x| \leq r$ and $|t - y| \leq r$. When the radius-of-usage of two resources overlap, nodes that are within the overlapping regions will be randomly serviced by any of the two resources.

A resource is used exclusively by a node during the service period denoted by the service time t_s which is sampled from an exponential distribution with mean 1.0. The distributed mutual exclusion algorithm by Ricart and Agrawala [84] is used to achieve exclusive access to the resources by the nodes. Figure 3.3 shows the events used to implement the model. The actions of the events in the distributed mutual exclusion algorithm are as described in the following section.

3.3.1 Distributed Mutual Exclusion Algorithm

The state of a node N_i consists of a queue of waiting nodes, $N_i.queue$, a state variable, $N_i.state$, a server identity variable, $N_i.server$, and a request-time variable $N_i.rTime$. The state of a resource R_i consists of a state variable, $R_i.state$, and a node identity variable, $R_i.node$.

At the start of the simulation, the state of each node is initialized to `RELEASED` and the state of each resource is initialized to `UNLOCK`.

A START event is scheduled for each node at the beginning of the simulation to start the resource acquisition process. This event is again re-scheduled for a node after the node has released a resource. On executing the START event, a node N_i randomly chooses to acquire a resource R_i from the set of resources accessible to it and set $N_i.server$ to R_i. The state of node N_i, $N_i.state$, is first set to WANTED. Assume T_i is the current simulation time of node N_i, the request-time variable $N_i.rTime$ is set to T_i. Suppose there are a total of m nodes within the radius-of-usage of resource R_i. Node N_i then sends REQUEST events, $request < T_i, N_i, R_i >$, to all the other $m-1$ nodes within the radius-of-usage of resource R_i.

Node N_i waits for a total of $m-1$ replies before assuming ownership of resource R_i. The state of the node N_i is then set to HELD and N_i proceeds to use the resource R_i.

A node N_j receiving a REQUEST event, $request < T_i, N_i, R_i >$, will immediately reply to the message if it is not in a HELD or WANTED state, by sending a REPLY event to node N_i. If node N_j's state is either HELD or WANTED and the resource it is currently interested in or having control on is resource R_j, it first checks if the requested resource from node N_i matches R_j. If $R_i \neq R_j$, a reply is immediately sent to node N_i.

If $R_i = R_j$ and the state of N_j is HELD, this implies that node N_j is currently using resource R_j. The request from node N_i will be queued without replying. If the state of N_j is WANTED, node N_j must decide if it has a higher priority in using R_j by comparing its request-time for the resource, $N_j.rTime$, with the request-time of node N_i, T_i. The request from node N_i will be queued if $N_j.rTime$ is smaller than T_i; otherwise node N_i has a higher priority in using the resource and a reply will be sent immediately to node N_i.

A node N_i that has successfully acquired control of a resource R_i sends it a USE event. Upon receiving a USE event from node N_i, a resource R_i sets its state $R_i.state$ to LOCK and the variable $R_i.node$ to N_i to keep track of the node currently using it. Resource utilization is modeled by resource R_i scheduling a FREE event to itself with a service time t_s. Upon executing a FREE event, resource R_i sends a DONE event to node N_i to notify it of the completion of resource usage.

Upon receiving a DONE event, node N_i sets its state to RELEASED and schedules a START event for itself. It also sends replies to all queued requests.

Error Type	Description of Error
1	A node receives a START message but the state of the node is not RELEASED
2	A node receives a REPLY message but the state of the node is not WANTED
3	A node receives a DONE message but the state of the node is not HELD
4	A node receives a DONE message from a wrong server
5	A server receives a USE message but the state of the server is LOCK
6	A server receives a FREE message but the state of the server is UNLOCK

Table 3.1: Description of Errors in Model Consistency.

3.3.2 Semantically Inconsistent Errors

Several types of errors that are semantically inconsistent with the distributed mutual exclusion protocol model are listed in Table 3.1. These errors will not occur when the model is executed using a sequential simulator or a parallel simulator running a conservative simulation protocol. However, these errors do arise when the model is executed under an optimistic protocol that uses risk.

The first four types of errors occur only with nodes. Error type 1 occurs when a node receives a START event and finds that its state is not RELEASED. This error is inconsistent with the model by considering the following.

A START event is only scheduled for a node at the start of the simulation, and whenever the node releases a resource. In the first case, all nodes have their states initialized to RELEASED at the beginning of the simulation. In the second case, the state of a node is set to RELEASED following the receipt of a DONE event from a resource and before the scheduling of a new START event to itself.

Error type 2 arises when a node's state is not WANTED but it receives a REPLY message from another node. Similarly, error type 3 occurs when a node receives a DONE event from a server but the state of the node is not WANTED. For error type 2, a node would only be expecting a REPLY event if it has started the process of acquiring a resource, set its state to WANTED and sent out REQUEST events to other nodes. For error type 3, a node would be expecting a DONE message from a resource if it has gained exclusive usage of a resource, set its state to HELD, and sent a USE event to the resource. Error type 4 occurs when a node receives a DONE event from a resource different from the one it had last sent a USE event.

The last two errors occur at the resources. Error type 5 occurs when a resource receives a USE message from a node but the state of the server is LOCK. The resource is currently being used by a node but another node is also trying to use the resource as well. This is clearly a violation of the semantic of the distributed mutual exclusion protocol which

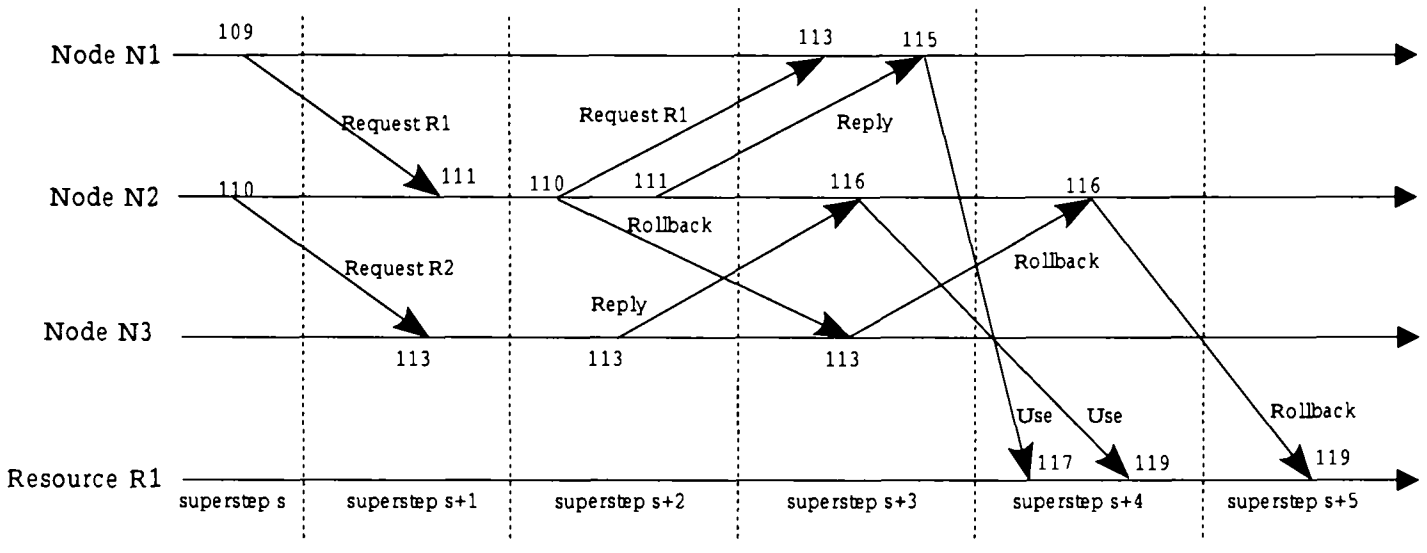


Figure 3.4: Consistency Error with Two Nodes Accessing a Resource at the Same Time Due to Erroneous REPLY Message.

guarantees that a resource will only be used by a node at any one time, and hence it should not receive a USE event from other nodes. The last error simply says that a resource should not expect to receive a FREE event if its state is not LOCK. This is true since a resource will only schedule a FREE event to itself after it receives a USE event from a node and set its own state to LOCK.

3.4 Rollback Inconsistency Errors in BSP-TW

In this section, we discuss how rollback inconsistency errors can occur in the distributed mutual exclusion model when the BSP-TW protocol is used. This will be followed by a discussion of different ways of extending the BSP-TW protocol to eliminate rollback inconsistency and stale state in the subsection sections.

To illustrate how rollback inconsistency can happen in BSP-TW with the distributed mutual exclusion model, let us consider the consistency error type 5 described in Table 3.1. Figure 3.4 shows a scenario that leads to the occurrence of this error.

Assuming resource $R1$ is shared by nodes $N1$ and $N2$ and resource $R2$ (not shown in the diagram) is shared by nodes $N2$ and $N3$. In superstep s, nodes $N1$ and $N2$ decide to use resources $R1$ and $R2$ respectively. Following the steps for acquiring a resource in the distributed mutual exclusion protocol, node $N1$ sends a REQUEST event with request-time 109 to node $N2$. Similarly, node $N2$ sends a REQUEST event with request-time 110 to node $N3$. Note that the request-time of a REQUEST event also corresponds to the send-time of

the event.

In superstep $s + 1$, both events arrived at the respective destination with a receive-time of 111 and 113 respectively. Both events are not simulated in superstep $s + 1$.

Suppose that in superstep $s + 2$, node $N2$ receives a straggler event that rolls back its simulation time to 110. Node $N1$ now has to cancel the REQUEST event it has sent to node $N3$ in superstep s by forwarding the corresponding anti-message to node $N3$. In the same superstep, node $N2$ re-simulates the rolled-back event at time 110 and decides to use resource $R1$ instead of $R2$ (due to changes in random number stream). It then forwards a REQUEST event to node $N1$ with request-time 110. Also in the same superstep, node $N2$ services the request from node $N1$. Since node $N1$ has a lower request-time for resource $R1$ than node $N2$, node $N2$ replied immediately to the request.

In the same superstep $s + 2$, node $N3$ services the request of node $N2$. Assuming that node $N3$ is not using any resources or is using a resource other than $R2$, it replies immediately to node $N2$'s request. However, the request from node $N2$ is destined to be rolled back in the subsequent supersteps.

In superstep $s + 3$, node $N1$ services the request from $N2$. Since it has a lower request-time than node $N2$, the request from node $N2$ is queued. Node $N1$ then receives the REPLY event from node $N2$ and proceeds to use the resource $R1$ by sending a USE event.

In the same superstep $s + 3$, node $N2$ receives a REPLY event. However, the event originates from $N3$ and is a reply for an erroneous request for resource $R2$. Taking the event as a reply for using resource $R1$, node $N2$ also sends a USE event to resource $R1$. In the same superstep, node $N3$ sends out an anti-message to cancel the erroneous REPLY event.

In superstep $s+4$, both USE events from $N1$ and $N2$ arrive at resource $R1$. $R1$ services the request from node $N1$ at time 117. Assuming that the resource will be occupied until time 120, the USE event from node $N2$ that arrives at time 119 forces resource $R1$ into an inconsistent state.

Obviously the most straight forward way to fix this source of inconsistency is to check if a reply matches the correct request. Indeed, a quick change to the code in the simulation model by adding a resource identity tag to all reply messages did reduce the number of consistency errors in the simulation model significantly. However, some consistency errors still exist and in particular, the consistency error that involves mutual exclusion of a resource is not completely eliminated. After analyzing event traces from a simulation run

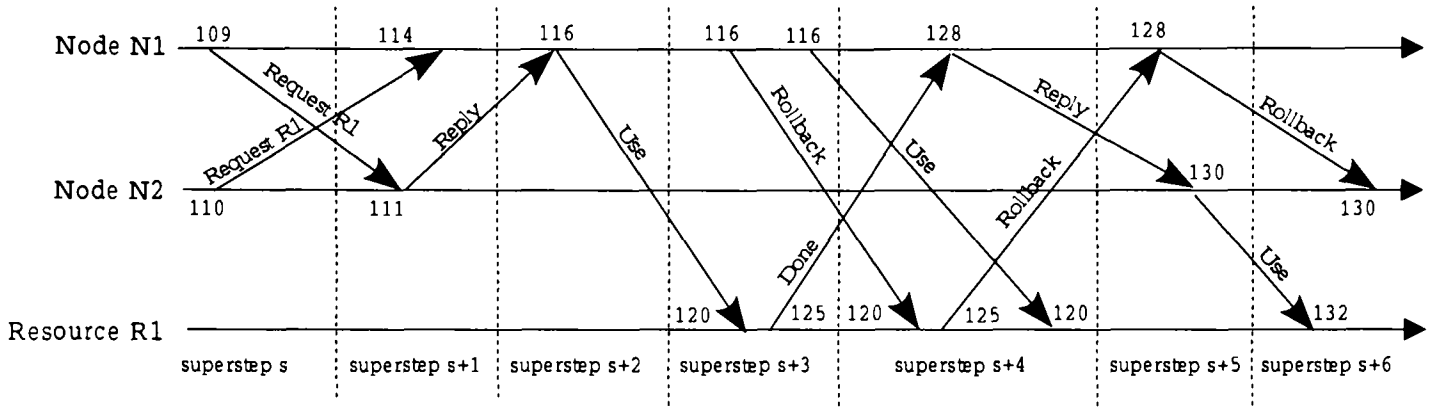


Figure 3.5: Consistency Error with Two Nodes Accessing a Resource at the Same Time Due to Erroneous DONE Message.

with this error, we discover another scenario that results in the error. Figure 3.5 shows the series of supersteps that lead to this error.

Assume again resource $R1$ being shared by nodes $N1$ and $N2$. In superstep s, both nodes decide to acquire control over resource $R1$ and send each other a REQUEST event. In superstep $s+1$, node $N1$ determines that it has a lower request-time and queued $N2$'s request. In the same superstep, node $N2$ determines that node $N1$ has a lower request-time and replied immediately to the request. The reply from node $N2$ reaches node $N1$ in superstep $s+2$. Node $N1$ proceeds to send a USE event to resource $R1$.

In superstep $s+3$, resource $R1$ receives the request from $N1$ at time 120. Assume that the resource is used for 5 time unit. Resource $R1$ sends a DONE event to $N1$ at time 125 in the same superstep. Suppose node $N1$ receives a straggler event in superstep $s+3$ that causes it to roll back to time 116. Node $N1$ sends an anti-message to undo the USE event it has sent to resource $R1$ in the previous superstep. In the same superstep, re-simulation of the REPLY event at time 116 generates the same USE event destined for resource $R1$.

In superstep $s+4$, node $N1$ receives the DONE event from resource $R1$ and proceeds to send a reply to node $N2$, granting it exclusive usage of resource $R1$. In the same superstep, resource $R1$ receives the two events from node $N1$. The first event causes a rollback and an anti-message to be sent to node $N1$ to cancel the DONE message sent in the previous superstep. The second USE event from $N1$ locks the resource for exclusive use by $N1$. Suppose $R1$ is utilized for a total of 15 time unit this time round, $R1$ will only be free at time 135.

In superstep $s+5$, node $N2$ receives the reply from $N1$ and sends a USE message to $R1$. In the same superstep, the anti-message from $R1$ arrives at $N1$, causing it to roll back

Description of BSP TW Algorithms	
BSP-TW1	Original BSP-TW without anti-message acknowledgement
BSP-TW2	BSP-TW1 with LP suspension and anti-message acknowledgement
BSP-TW3	BSP-TW2 with 25% throttling on event limit
BSP-TW4	BSP-TW1 with only anti-message acknowledgement

Table 3.2: Description of BSP-TW Algorithms: BSP-TW1, BSP-TW2, BSP-TW3 and BSP-TW4.

and send an anti-message to cancel the REPLY event it sends to $N2$ in superstep $s + 4$.

In superstep $s + 6$, the USE event from $N2$ arrives at $R1$ with a receive-time of 132. Since $R1$ will be used by $N1$ until time 135, this event drives resource $R1$ into an inconsistent state.

3.5 Anti-Message Acknowledgement

The examples described in the previous section suggest that it is not feasible to simply add user code segments to detect and eliminate rollback inconsistency errors. In [76], Nicol proposed a message acknowledgement scheme that eliminates both rollback inconsistency errors and stale states. By requiring an LP to wait for an anti-message to be acknowledged before proceeding to the next event, the scheme guarantees safety from rollback inconsistency errors. Furthermore, by also requiring an LP to wait for acknowledgement of a normal external message before sending out another external message, the scheme also eliminates the problem of stale states.

Table 3.2 gives a description of the four variations of BSP-TW that will be discussed in this section. We first implemented the anti-message acknowledgement scheme in the original BSP-TW protocol. The solution to normal message acknowledgement will be discussed in section 3.6. For the rest of this chapter, we will refer to the original BSP-TW protocol without anti-message acknowledgement as BSP-TW1, and the original BSP-TW protocol with anti-message acknowledgement as BSP-TW2. The necessary changes for adding anti-message acknowledgement to BSP-TW1 are as follows:

Suspension of LPs

- An LP is suspended if any event e_i that has been rolled back on the LP sends out anti-messages. Associated with each LP, LP_i, is a temporary event buffer, `ackBuffer`,

holding all such events waiting for anti-message acknowledgement.

- Associated with each LP is another temporary event buffer, `suspendBuffer`, that temporarily holds all events yet to be simulated but destined to arrive at LP_i.

- An event e_i is removed from `ackBuffer` when all the anti-messages it sent out are acknowledged. The event is then placed into the `suspendBuffer`.

- While an LP is suspended from execution, other LPs in the same processor continue to process events so long as the event limit imposed by the BSP-TW protocol is not reached. Events destined for a suspended LP is placed immediately into its `suspendBuffer`.

Resumption of LPs

- An LP is resumed when there are no more events in the buffer `ackBuffer` waiting for anti-message acknowledgement.

Event cancellation and anti-message acknowledgement

- Event cancellation mechanism (when a positive message meets its anti-message) in the processor event-list is extended to immediately generate and dispatch an anti-message acknowledgement. Message cancellation also extends to the temporary event buffer, `suspendBuffer`, to ensure immediate dispatch of anti-message acknowledgement.

- The events in the event buffer `suspendBuffer` for all LP_i that are suspended are removed and reinserted into the processor event-list at the beginning of each superstep. Since the processor event-list is shared between LPs local to the processor, merging the events for suspended LPs into the event-list has the effect of reducing the local optimism of the processor to simulate events for other LPs that are not suspended.

GVT computation

- Since some of the events for suspended LPs will remain in the buffers `ackBuffer` and `suspendBuffer`, the GVT computation is modified to take into account events in these buffers as well.

	Low Connectivity				High Connectivity			
	Error Type				Error Type			
	2	3	4	5	2	3	4	5
4 Processors								
R=10%	19	2	0	1	3096	43	0	109
R=30%	252	10	0	24	2087	29	0	46
R=50%	352	21	1	27	1516	25	0	22
9 Processors								
R=10%	254	30	1	8	3677	59	0	117
R=30%	684	30	0	62	3759	50	0	77
R=50%	816	51	0	37	3033	47	0	37
16 Processors								
R=10%	339	45	0	30	4055	71	0	116
R=30%	1201	46	5	97	4910	72	0	62
R=50%	1237	80	8	99	3655	56	0	56

Table 3.3: Number of Errors Detected in the Distributed Mutual Exclusion Model using BSP-TW1 Protocol.

3.5.1 Experiments with Anti-message Acknowledgement

The experiments are conducted using 4 processors on a 86-node SGI Origin-2000 system. The Oxford BSP Toolset [78] is used as the runtime library for this set of experiments and all subsequent experiments in this thesis. The distributed mutual exclusion model has a grid size 100 by 100. Let R be the percentage of resources in the grid cells. We experiment with different values of R as well as two different radius-of-usage values, 1 and 2. A radius-of-usage of 1 represents low connectivity in the network, as each resource will be connected to a maximum of 8 nodes. A radius-of-usage of 2 represents high connectivity with each resource connecting to a maximum of 24 nodes. A spinloop of 25μs is used to model fine event granularity.

The simulation length for all the runs are fixed at 1000 time unit. A GVT computation interval of 40 supersteps is used throughout the experiments. Nodes and resources in the simulation are modeled as simulation objects in BSP-TW. A two-dimensional block partitioning is used to assign grid cells to processors.

Table 3.3 shows a breakdown of the different errors detected for the model executed using the BSP-TW1 protocol. Error types 1 and 6 are not detected in all the runs. The high connectivity model shows a significantly higher number of errors than the model with low connectivity.

The same model is then executed using the BSP-TW2 protocol. As expected, no error

	BSP-TW1	BSP-TW2	BSP-TW3	BSP-TW4
No. of supersteps	1189	2993	2993	1189
No. of events rolled back	517851	15335534	313985	517851
No. of remote messages	469400	897615	436068	469400
No. of remote anti-messages	67457	495672	34125	67457
Average time (sec.)	222.5	469.2	226.0	212.9
Barrier time (sec.)	233.6	527.1	388.8	224.308

Table 3.4: Performance Comparison between BSP-TW1, BSP-TW2, BSP-TW3 and BSP-TW4 Protocols on 4 Processors for High Connectivity Model with R=50%.

is detected from these runs. However, the addition of the anti-message acknowledgement mechanism has a significant impact on the performance of the simulation protocol. Table 3.4 shows a comparison on some optimistic simulation statistics using the BSP-TW1 versus BSP-TW2 protocols for the high connectivity simulation model with R=50%.

The BSP-TW2 protocol has significantly higher number of rollback events and anti-messages as compared to the BSP-TW1 protocol. In terms of the number of supersteps needed to complete the simulation, the BSP-TW1 protocol requires far fewer supersteps than BSP-TW2.

The overhead in processing the rolled-back events in BSP-TW2 is reflected in the two-fold increase in average time, which measures the total time spent by each processor in the simulation loop excluding the time waiting for barriers.

The barrier time, which measures the sum of the maximum time for each superstep, reflects the load-imbalance among processors across supersteps. Ideally, if the workload is well-balanced across all supersteps, the average time should be close to the barrier time. While the difference between the average time and barrier time for BSP-TW1 is small, the barrier time for BSP-TW2 is 12% higher than the average time, reflecting significant load-imbalance across supersteps.

The sudden explosion of rollback events using BSP-TW2 is attributed to the over-optimism in executing events of other LPs while some LPs in the same processor are waiting for anti-message acknowledgement. For BSP-TW1, the total number of external anti-messages sent is only 14% of the total number of ordinary external messages sent. For BSP-TW2, since each anti-message must be acknowledged, we need to double the number of external anti-messages to take into account anti-message acknowledgements. The total number of external anti-messages and anti-message acknowledgements sent amount up to 110% of the total number of actual ordinary external messages sent in BSP-TW2.

Our next attempt is to reduce the number of rollbacks by adding a throttling mechanism to BSP-TW2. For every LP that is suspended, the remaining limit to the number of events that can be processed in the current superstep is reduced by 25%. This reduction is maintained across supersteps as long as the LPs remain suspended. The net effect is that a processor with a large number of LPs suspended will be penalised from executing more events. We refer to this variant of the BSP-TW protocol as BSP-TW3.

As shown in the example in Table 3.4, the number of rollbacks and remote anti-messages for BSP-TW3 drop significantly. In fact, both the number of events rolled back and the number of remote anti-messages sent are also lower than that of BSP-TW1. This agrees with the small difference in the average time for BSP-TW1 and BSP-TW3. However, the addition of the throttling mechanism resulted in even higher load-imbalance between processors across supersteps. This is reflected in the long barrier time for BSP-TW3, which is 71% higher than the average time.

In oder to isolate the cost of suspending an LP from the cost of acknowledging anti-messages, we implemented a version of BSP-TW1, hereafter referred to as BSP-TW4, with the anti-message acknowledgement mechanism but without LP suspension. BSP-TW4 performs the same anti-message acknowledgement task as BSP-TW3. However, BSP-TW4 does not suspend LP for sending out anti-messages, and thus has the same simulation statistics as BSP-TW1. The difference in performance between BSP-TW1 and BSP-TW4 hence reveals the cost of acknowledging anti-messages.

Figure 3.6 shows the speedup graphs for BSP-TW1, BSP-TW2, BSP-TW3 and BSP-TW4 on 4, 9 and 16 processors. Table 3.5 shows the load-imbalance in terms of percentage difference between barrier time and average time. Table 3.6 shows the percentage of anti-messages over the total number of committed events in the system. The statistics for BSP-TW4 is the same as BSP-TW3 and is thus not shown in the table.

A few observations can be made from the graphs. Firstly, the small difference in performance between BSP-TW1 and BSP-TW4 shows that the cost of anti-message acknowledgement is indeed very small. However, BSP-TW4 shows noticeable performance degradation with increasing number of processors. This can be attributed to the corresponding increase in the percentage of anti-messages with the number of processors as shown in Table 3.6. Table 3.6 also shows that the model with high connectivity has a higher proportion of anti-messages than the model with low connectivity.

As shown in Figure 3.6, the percentage of anti-messages using BSP-TW3 is much lower than that of BSP-TW2. Although there is a substantial performance improvement

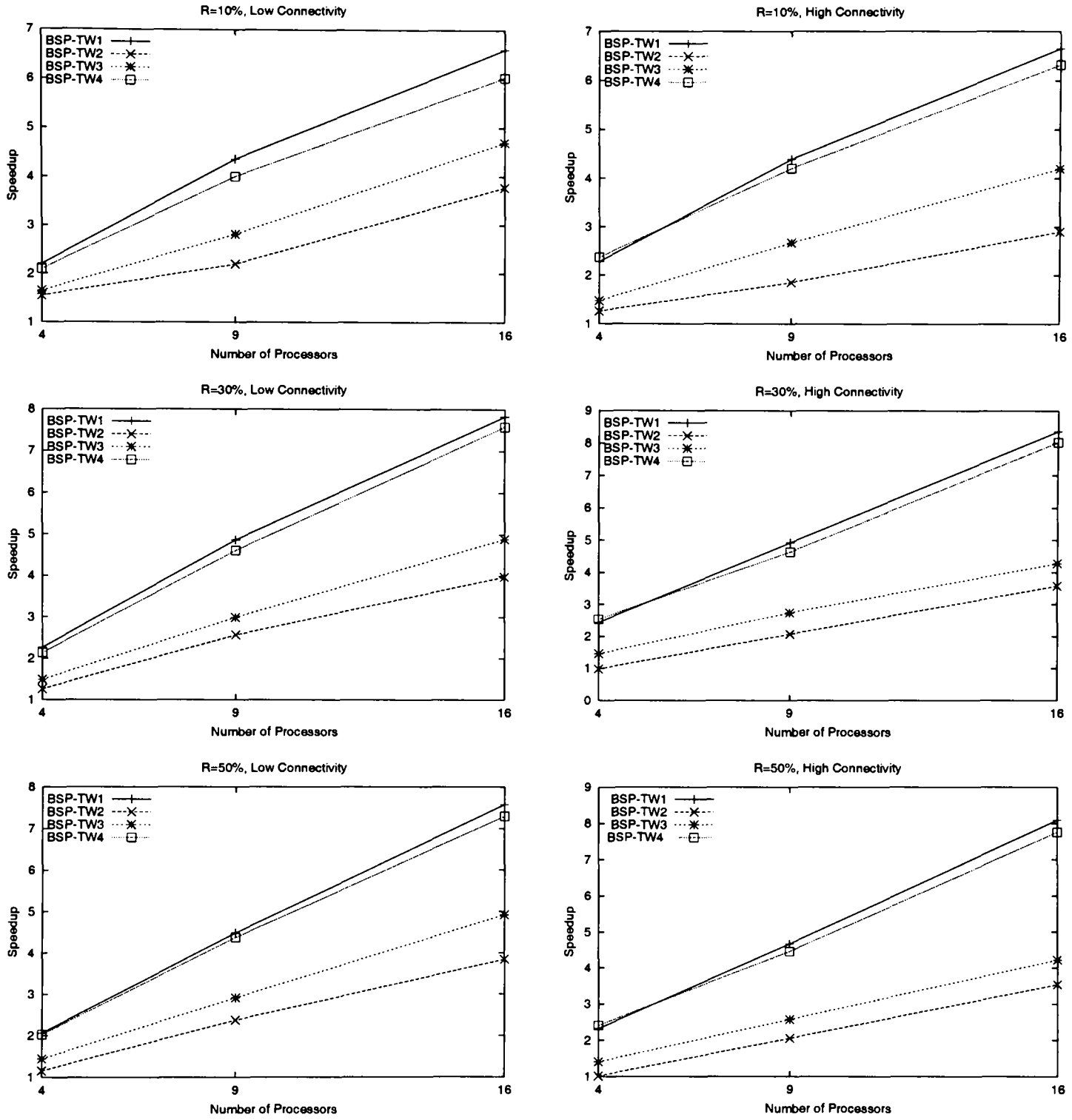


Figure 3.6: Speedup Graphs for Distributed Mutual Exclusion Model using BSP-TW1, BSP-TW2, BSP-TW3 and BSP-TW4 Protocols.

using BSP-TW3 over BSP-TW2, the performance of BSP-TW3 is still significantly worse than that of BSP-TW1. Moreover, Table 3.5 shows that BSP-TW3 has even worse load-imbalance than BSP-TW2. For most cases using 9 and 16 processors, the barrier takes twice as long as the average time.

	Low Connectivity			High Connectivity		
	4	9	16	4	9	16
R=10%						
BSP-TW1	8.9	24.4	44.7	7.5	21.9	39.0
BSP-TW2	20.6	51.3	75.2	12.9	48.7	83.9
BSP-TW3	36.7	92.7	103.2	64.8	110.8	145.4
BSP-TW4	7.5	22.3	44.6	7.8	19.8	39.8
R=30%						
BSP-TW1	6.4	14.7	24.9	4.7	17.5	19.4
BSP-TW2	16.6	30.6	45.6	13.6	30.5	39.5
BSP-TW3	50.2	84.0	102.7	78.1	118.4	149.7
BSP-TW4	6.3	13.8	22.6	5.0	16.9	19.7
R=50%						
BSP-TW1	8.4	16.8	23.6	4.9	17.6	20.6
BSP-TW2	18.0	31.8	42.2	12.3	29.9	38.5
BSP-TW3	53.2	81.8	99.2	72.0	115.8	140.8
BSP-TW4	8.2	16.3	23.2	5.3	17.4	19.1

Table 3.5: Percentage Difference between Barrier Time and Average Time using BSP-TW1, BSP-TW2, BSP-TW3 and BSP-TW4 Protocols.

	Low Connectivity			High Connectivity		
	4	9	16	4	9	16
R=10%						
BSP-TW1	0.7	10.5	16.3	8.0	17.6	23.1
BSP-TW2	31.3	74.4	62.1	91.3	132.4	120.8
BSP-TW3	1.3	8.4	10.9	2.7	7.9	15.2
R=30%						
BSP-TW1	2.1	5.2	9.1	4.7	10.7	14.0
BSP-TW2	70.6	81.8	86.0	133.4	129.1	125.5
BSP-TW3	1.4	3.9	6.9	2.3	5.6	10.6
R=50%						
BSP-TW1	2.4	5.8	8.7	4.0	9.2	12.2
BSP-TW2	74.2	82.9	86.5	119.4	121.4	115.9
BSP-TW3	1.6	4.0	6.9	2.4	6.3	11.0

Table 3.6: Percentage of Anti-Messages over Committed Events using BSP-TW1, BSP-TW2 and BSP-TW3 Protocols.

3.6 Extended Barrier with Implicit Anti-Message Acknowledgement

Analysis of BSP-TW2 and BSP-TW3 suggests that load-imbalance due to LP suspension is the root cause of the performance difficulties, and the suspension is due to the large

P	l (μs)	g (μs/word)
4	22.7	0.08
9	54.1	0.15
16	121.1	0.23

Table 3.7: BSP Parameters for 86-nodes 195Mhz (101.03 Mflops) Origin 2000 using Different Number of Processors P.

Description of BSP-TW Algorithms	
BSP-TW5	Original BSP-TW with Extended Barrier
BSP-TW6	BSP-TW5 with suspension of LP that sends external messages
BSP-TW7	BSP-TW6 with 25% throttling
BSP-TW8	BSP-TW6 with 10% throttling

Table 3.8: Description of BSP-TW Algorithms: BSP-TW5, BSP-TW6, BSP-TW7 and BSP-TW8.

number of supersteps needed to push anti-message acknowledgements around in the standard BSP framework. Table 3.7 shows that the cost of executing a barrier synchronization (**bsp_sync()**) on the SGI Origin is much smaller than the observed cost of load-imbalance. This observation led us to consider a mechanism that would accelerate the anti-message acknowledgement processing, at the cost of additional barriers. Table 3.8 gives the descriptions of the next four variants of BSP-TW that will be described in this section.

The algorithm in Figure 3.7 uses a series of supersteps to roll back an entire chain of causally incorrect events whenever external anti-messages are sent from a processor.

We compare this algorithm with the original BSP-TW algorithm outlined in Figure 2.1 in Chapter 2. Part A of the algorithm remains the same. Part B is expanded into three parts. In part B1, all normal messages and anti-messages are received and any rollbacks are processed accordingly. In part B2, suspend-messages will be broadcast to all other processors if any anti-message is sent in part B1. The `bsp_sync()` that follows, together with the receiving of events in part B3, allows the suspend-messages and anti-messages sent in parts B1 and B2 to arrive at the respective destinations.

Part C of the algorithm remains the same. In part D, the number of events that a processor can execute in the current superstep is set to 0 if one of the following conditions are satisfied: 1) suspend-message is sent in part B1; 2) suspend-message is received in part B3; 3) anti-message is sent in part B3; 4) anti-message is sent while processing one of the n_e events.

```
bsp_begin()
[A] Initialization;
while GVT < SimEndTime
    [B1] Receive external events and process rollback
    [B2] Broadcast suspend-message if any anti-message is sent in part B
    bsp_sync();
    [B3] Receive external events and process rollback
    [C] Compute new GVT, perform fossil collection and compute new event limit n_e
        every k supersteps
    [D] Execute n_e events. n_e is set to zero if :
        [1] suspend-message is sent in part B1;
        [2] suspend-message is received in part B3;
        [3] anti-message is sent in part B3;
        [4] anti-message is sent while processing an events.
    bsp_sync();
endwhile
bsp_end();
```

Figure 3.7: Algorithm for BSP-TW with Extended Barrier.

The net effect of the above scheme is that the processors will engage in a series of normal event processing supersteps. Any external anti-message sent will be followed by a series of supersteps that consist of receiving events and rollbacks only, without any event execution.

We call this series of supersteps "extended barriers". The extended barrier works like a special superstep whereby all anti-messages sent while the extended barrier is in effect will be acknowledged. The suspend-messages keep all processors suspended while the anti-messages undo the entire rollback chain.

A point to note is that the superstep counter used for keeping track of GVT computation and the computation of event limit is only incremented at the end of the extended barrier, but not after each superstep within the extended barrier. We consider an extended barrier a single "superstep", regardless of the number of supersteps taken to complete it.

It is also worthwhile to note that anti-message acknowledgement is not needed in this algorithm since no normal event processing can proceed while the rollback is going on. We will refer to this variant of extension to BSP-TW as BSP-TW5.

3.7 Normal Message Acknowledgement

We have so far concerned ourselves only with anti-message acknowledgements. Nicol and Liu also showed that stale state can be eliminated if every ordinary message is acknowledged before an LP sends another, in the same fashion as anti-messages are acknowledged. The fundamental idea is the same—wait until any rollbacks caused by the message have been fully propagated before continuing to process the LP. At face value it would seem that the cost of implementing acknowledgements would be high, as far more ordinary messages are sent than anti-messages. However, once again the structure of SimBSP and our solution to process rollback trees quickly can be leveraged to provide an efficient solution to the stale state problem.

We propose an extension to BSP-TW5, called BSP-TW6, which deals with the threat of stale state. For normal messages sent between LPs on the same processor, no acknowledgements are needed as any rollbacks caused by the messages are immediately performed. If an LP sends messages to LPs in other processors, the LP is immediately suspended for the current superstep.

Note that in this case, only the LP that sends out external messages is suspended, coresident LPs are unaffected. Also, the LP is suspended for only one superstep and any events destined for the suspended LP in the current superstep will be kept in a temporary buffer and will be merged back into the event-list at the end of the superstep.

It is somewhat remarkable that a sending LP needs only be suspended for one superstep. The reason is that acknowledgements for the external messages are implicitly received at the start of the next superstep. Any rollback caused by the external messages at the receiving LPs would trigger an extended barrier, at the end of which all the rollback-chains initiated by the external messages are removed. Suspension of the LP for a step just ensures that any rollback tree triggered by its message is completely processed before the LP resumes. This is equivalent to receiving explicit acknowledgements. We denote this extension as BSP-TW6.

Tables 3.9 and 3.10 show that BSP-TW6 generally has twice the number of rollbacks and anti-messages compared to BSP-TW5. We saw this same phenomenon when comparing BSP-TW1 and BSP-TW2— the suspension of LPs, coupled with the SimBSP instance on executing exactly n_e events in a superstep, forces unprofitable speculative execution. To correct this we experimented with the same throttling mechanism as mentioned in section 3.5 on BSP-TW6. Each time an LP is suspended for sending out external messages,

	Low Connectivity		High Connectivity	
	BSP-TW5	BSP-TW6	BSP-TW5	BSP-TW6
R=10%				
4 Processors	13704	35397	348256	389862
9 Processors	163531	197949	641588	972061
16 Processors	339111	405452	1073115	1580840
R=30%				
4 Processors	60591	127091	437738	721267
9 Processors	140716	270793	928435	1673417
16 Processors	256731	443110	1409948	2603786
R=50%				
4 Processors	51155	98431	290935	594300
9 Processors	133335	251563	681462	1312950
16 Processors	215710	327548	917059	1696651

Table 3.9: Comparison between BSP-TW5 and BSP-TW6 in terms of the Number of Anti-Messages.

	Low Connectivity		High Connectivity	
	BSP-TW5	BSP-TW6	BSP-TW5	BSP-TW6
R=10%				
4 Processors	21669	53374	566460	681951
9 Processors	220665	269700	1082110	1652469
16 Processors	437982	545274	1803351	2733183
R=30%				
4 Processors	89081	180633	729065	1163745
9 Processors	203045	382296	1493024	2645103
16 Processors	366679	621089	2284885	4144120
R=50%				
4 Processors	71739	133516	460798	899032
9 Processors	179900	333402	1061214	1985119
16 Processors	289204	438400	1422109	2580613

Table 3.10: Comparison between BSP-TW5 and BSP-TW6 in terms of the Number of Rollbacks.

the event limit per superstep, n_e, is reduced by a fixed percentage. The net effect is that the more LPs that are suspended on a processor in a superstep, the less event the processor gets to execute in that superstep.

We experimented with two different throttling limits, 25% and 10%. The variant of BSP-TW6 with 25% throttling will be referred to as BSP-TW7 and the one with 10% throttling will be referred to as BSP-TW8.

	BSP-TW5	BSP-TW6	BSP-TW7	BSP-TW8
No. of supersteps	1435 (2816)	3280 (6405)	6560 (10179)	3813 (6694)
No. of events rolled back	460798	899032	96247	93115
No. of remote messages	447877	474252	413456	413443
No. of remote anti-messages	45934	72309	11513	11500
Average time (sec.)	214.11	232.266	215.508	216.399
Barrier time (sec.)	274.492	502.083	296.056	278.552

Table 3.11: Performance Comparison between BSP-TW5, BSP-TW6, BSP-TW7 and BSP-TW8 on 4 Processors for the High Connectivity Model with R=50%.

Table 3.11 shows a comparison on the simulation statistics using BSP-TW5 and BSP-TW6 for the high connectivity simulation model with R=50%. The numbers in parentheses show the actual number of supersteps taken if we expand and count each individual superstep within each extended barrier. Figure 3.8 shows the performance of BSP-TW5, BSP-TW6, BSP-TW7 and BSP-TW8 with respect to BSP-TW1.

Comparing Table 3.11 and the statistics for BSP-TW1 in Table 3.4, we note that the average times required to complete the simulation using BSP-TW5, BSP-TW6, BSP-TW7 and BSP-TW8 are closed to the average time for BSP-TW1. The mechanisms we propose do not induce a great deal of computational overhead. However, BSP-TW6 shows long barrier time which is twice the average time, indicating load-imbalance using this approach. This agrees with our observation on BSP-TW2 that suspension of some LPs while allowing other LPs in the same processor to proceed ahead will result in over-optimism, hence leading to load-imbalance.

The significant reduction in barrier time with BSP-TW7 and BSP-TW8 also confirms our observation on BSP-TW3 that the over-optimism can be controlled using some form of throttling. However, we also note that the performance is sensitive to the throttling level used. This is evident from Figure 3.8 which shows BSP-TW7 performing significantly worse than BSP-TW8. The results suggest further performance improvement might be achieved by fine-tuning the throttling level used.

The key question to ask after carrying out these refinements is what is the performance cost of ensuring that a SimBSP simulation is free from rollback inconsistency and stale state. Figure 3.9 shows the speedup curves for BSP-TW1, BSP-TW5, and BSP-TW8 on 4, 9 and 16 processors for the models with R=50%. Comparison between BSP-TW5 and BSP-TW8 shows that once we have freedom from rollback inconsistencies, the cost of getting freedom from stale state is not excessive. Comparison between BSP-TW1 and BSP-TW5 shows that the cost of freedom from rollback inconsistency is an increase of about 20%

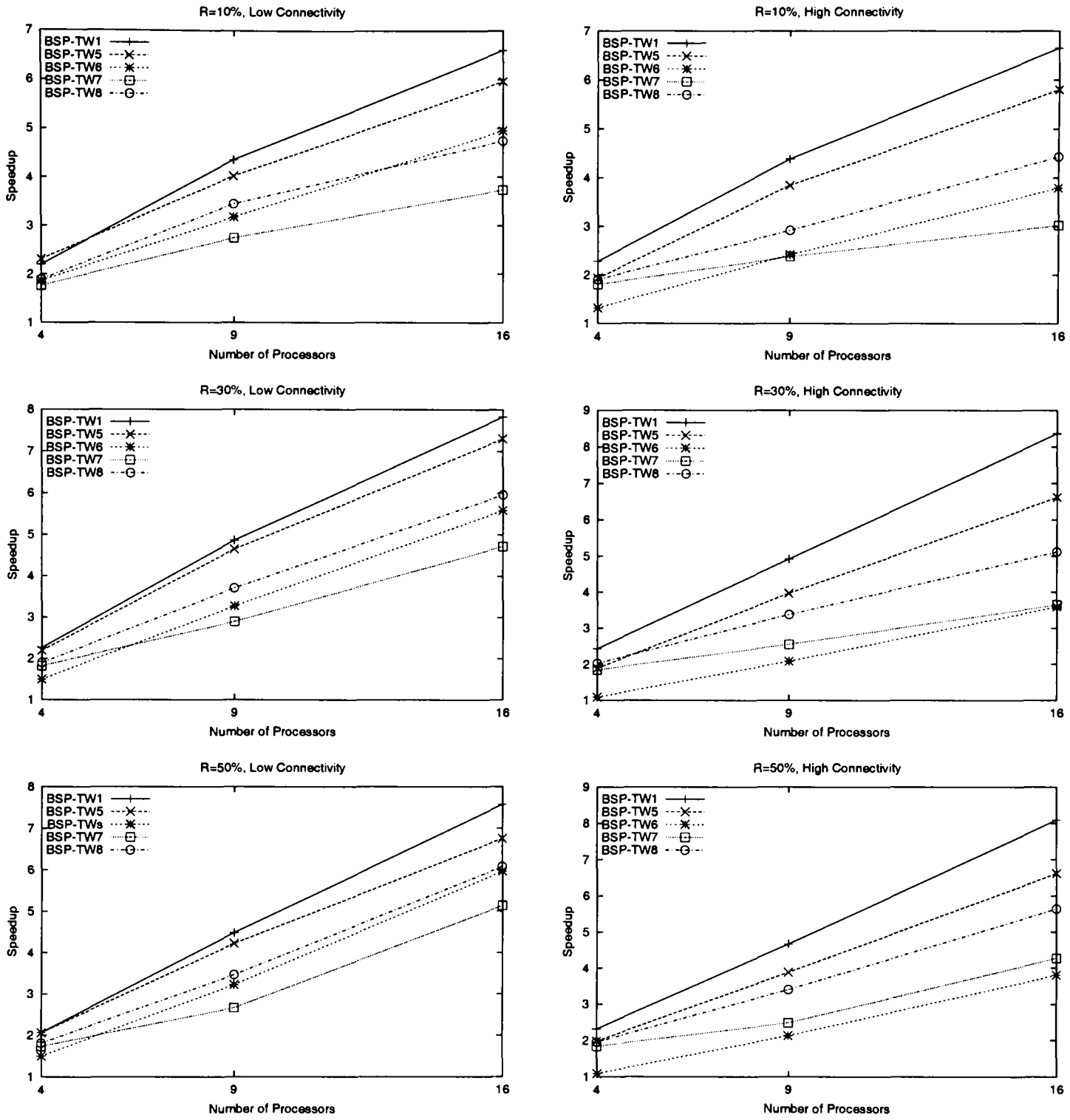


Figure 3.8: Speedup Graphs for Distributed Mutual Exclusion Model using BSP-TW1, BSP-TW5, BSP-TW6, BSP-TW7 and BSP-TW8.

in execution time, which seems quite an acceptable price to pay. Furthermore, that cost does not manifest itself as more computational work so much as it manifests itself as load-imbalance. Better techniques for dynamically managing load might conceivably reduce this cost further.

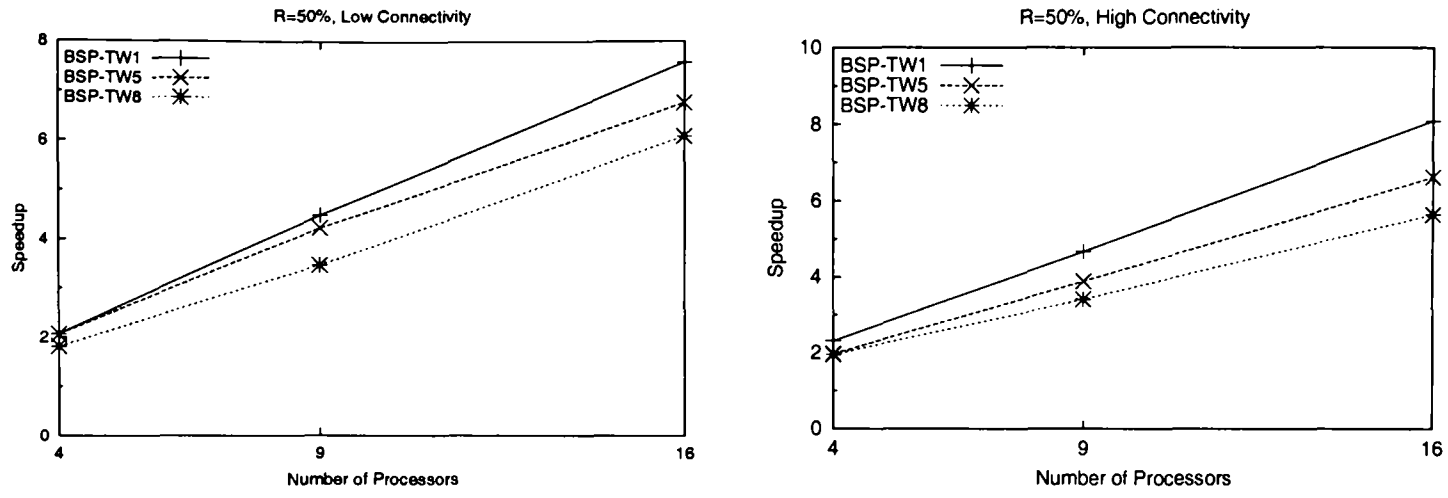


Figure 3.9: Speedup Graphs using Different BSP-TW Protocols.

3.8 Proof for BSP-TW with Extended Barrier

We now present the proof that BSP-TW with extended barrier and LP suspension does indeed eliminate rollback inconsistency and stale state.

Lemma 1 *Using extended barrier, all anti-messages are acknowledged.*

Proof Using extended barrier, internal anti-messages sent out by an LP, say LP_i, do not require explicit acknowledgement. As all internal anti-messages are processed immediately in BSP-TW, all rollbacks on local LPs are carried out and any further anti-messages are sent before the processor is allowed to execute a new event. These anti-messages are implicitly acknowledged if no external anti-messages are sent out while the rollback is taking place.

If LP_i, or any LP involved in the internal rollback, sends out external messages, the processor will be suspended immediately (through setting the event limit per superstep to zero). The anti-messages are received at the beginning of the next superstep in part B1 of Figure 3.7. If these anti-messages cause any further rollbacks and generate another wave of external anti-messages, then suspend-messages will be sent to all processors. Any suspend-messages will be received in part B3, forcing all processors to be suspended for one superstep. The process repeats itself for the number of supersteps required to undo the entire rollback chain. The anti-messages are considered implicitly "acknowledged" at the end of this extended barrier when no anti-messages are sent in part B1.

Lemma 2 *Using extended barrier and suspension of LP that sends out external messages, normal messages are also acknowledged.*

Proof For normal messages that are sent internally, the acknowledgement is implicitly received in the sense that if the message does not trigger any rollback, program execution control is returned to the processor immediately to execute another event. If the messages do cause rollback in other local LPs, the rollback will be carried out immediately and program execution control will only be returned to the processor after the rollback is completed. Any anti-message that is sent to external LPs immediately triggers an extended barrier.

For normal messages that are sent externally by an LP, the LP is suspended immediately for the rest of the superstep. These messages will arrive at the respective destination LPs in part B1 of Figure 3.7 in the next superstep. Any rollbacks on the destination LP that are caused by receiving these events would have taken place at the end of part B1. Any anti-messages that are sent out due to these rollbacks trigger the start of an extended barrier.

Thus, for both internal and external messages sent by an LP, the LP (or the processor) will be suspended while any rollbacks that are initiated by these messages take place. The external messages will be considered implicitly "acknowledged" when the rollbacks, if any, are completed.

Thus, it follows from lemmas 1 and 2 and the proof given in [76] that all rollback inconsistencies and stale states are eliminated using BSP-TW with extended barrier and LP suspension.

3.9 Related Work

In this section, we describe two related work in the context of this study. In [18], a fault tolerance rollback-based optimistic discrete event simulation protocol is proposed to allow crash recovery of a distributed simulation. This scheme uses dependency vectors tagged onto normal messages to detect causality violation among LPs and thus eliminates the use of anti-messages. Although the scheme is proven to be free from rollback inconsistency, there is no provision in the scheme for the elimination of stale state.

The BSP-TW8 protocol proposed in section 3.6 closely resembles the "wolf calls" protocol described in [66]. Both schemes use the broadcast of a special control message to stop the spread of incorrect computations. While the BSP-TW8 protocol sends this control message to all the processors, the "wolf calls" protocol determines the set of LPs this con-

trol message should be sent to using a "spheres of influence" parameter. This technique is not suitable for BSP-TW8 due to the synchronous nature of the computation. Suspending only some processors in any superstep will only lead to more speculative executions and worsen the load-imbalance among the processors.

Although the "wolf calls" protocol seems to be capable of eliminating rollback inconsistency, there is also no mechanism in the protocol to prevent the occurrence of stale state.

3.10 Summary

The potential risk hazards of an optimistic protocol are clearly illustrated in the distributed mutual exclusion model presented in this chapter. Some scenarios whereby the mutual exclusion property of the distributed mutual exclusion protocol is violated are presented in the context of BSP-TW protocol. We verified that by adding anti-message acknowledgement into an optimistic protocol such as BSP-TW, rollback inconsistency errors can be eliminated.

Different algorithms to improve the performance of BSP-TW with anti-message acknowledgement are considered. We showed that BSP-TW with extended barrier can achieve the effect of anti-message acknowledgement with comparable performance to the original BSP-TW algorithm. We further extended the algorithm to allow stale state to be eliminated from the simulation.

Experimental results from these algorithms are presented and analyzed. Our study shows that reasonable performance can still be achieved for the algorithms to eliminate rollback inconsistency. However, the further performance degradation that resulted if we try also to eliminate stale state may well deserve some careful considerations.

We have also observed that the performance of BSP-TW is sensitive to the amount of throttling applied on the event limit in each superstep. In the next chapter, we examine how adaptive tuning on the event limit of BSP-TW protocol can be used to improve load-balance and reduce rollback.

Chapter 4

Adaptive Tuning of BSP-TW

4.1 Overview

While the BSP-TW algorithm provides an automatic means of throttling the number of events being simulated per superstep based on statistics from fossil collected events, the main objective of the algorithm is to complete a simulation run using the least number of supersteps possible. The large proportion of rollbacks is recognized as a potential problem in trying to complete a simulation using the least number of supersteps.

In this chapter, the impact of excessive rollback to the performance of the original BSP-TW algorithm is illustrated. A new adaptive BSP-TW algorithm that automatically determines the number of events to be executed per superstep is proposed. The algorithm seeks to keep the computation load-imbalance and rollback overhead to a minimum, at the expense of incurring higher synchronization cost. Experimental results using the original BSP-TW algorithm and the new adaptive BSP-TW algorithm on several benchmark simulation models are presented and compared.

The rest of the chapter is organized as follows. In section 4.2, the BSP cost function for the original BSP-TW algorithm is presented. In section 4.3, we describe the adaptive BSP-TW algorithm.

Section 4.4 presents experimental results comparing the adaptive BSP-TW with the original BSP-TW on several benchmark simulation models. Section 4.5 describes some related work and section 4.6 summarizes the work carried out in this chapter.

4.2 Cost Model for BSP-TW

The BSP cost model for the BSP-TW algorithm can be expressed as:

$$\text{cost}(S) = \sum_{i=1}^{n_s} (w(i) + gh(i) + l) \tag{4.1}$$

$$= W + g * \sum_{i=1}^{n_s} h(i) + n_s l \tag{4.2}$$

where n_s is the total number of supersteps; $w(i)$ is the computation cost for superstep i; $W = \sum_{i=1}^{n_s} w(i)$ is the total computation cost of the algorithm; and $h(i)$ is the maximum number of messages sent or received respectively by any processor in superstep i. The architecture dependent parameters g and L represent the communication and synchronization costs respectively.

The cost of the algorithm consists of three parts, the computation cost W, the communication cost $g * \sum_{i=1}^{n_s} h(i)$ and the synchronization cost $n_s l$. The computation cost W, which is affected by factors such as load-imbalance and the number of events rolled back, often dominates the cost for the BSP-TW algorithm. This is especially true for a shared-memory environment with low values of l and g [88].

If W represents a perfectly balanced workload among the P processors which does not include any rolled-back or re-simulated events, we can define a more realistic workload, W', that takes into account load-imbalance and re-simulated events. W' is defined as follows:

$$W' = \frac{W}{\alpha\beta}; \tag{4.3}$$

α and β are defined as:

$$\alpha = \frac{N_s}{PN_{max}} \text{ and } \beta = \frac{N_c}{N_s} \tag{4.4}$$

where N_s and N_c are the total number of events simulated and the total number of events

committed respectively at the end of a simulation run. P is the number of processors and N_{max} is the cumulative sum of the maximum number of events that are simulated in any processor per superstep. Both N_s and N_{max} include events that are re-simulated due to rollbacks.

Intuitively, α measures the load-balance among processors. A well-balanced workload between processors across all supersteps will yield a value of α close to 1.0. Given the total number of events rolled back, N_r=N_s-N_c, 1-β measures the proportion of rollback incurred by the algorithm. A β value close to 1.0 implies very little rollback. The product $\alpha\beta$ measures the combined effect of load-imbalance and rollback on the simulation.

Although the computation cost equation ignores the cost of state-saving and rollbacks, it shows that if the computation cost significantly dominates the cost equation, i.e.

$$\frac{W}{\alpha\beta} \gg g * \sum_{i=1}^{n_s} h(i) + n_s l, \tag{4.5}$$

then performance gain might be possible by using more supersteps to achieve better load-balance and minimize the number of events rolled back.

Experimental results show that trying to complete a simulation run using the minimum number of supersteps often leads to high load-imbalance as well as a large number of events being rolled back. The original BSP-TW algorithm attempts to curb the excessive rollback of events by arbitrarily reducing the computed event limit, n_e, by a fixed constant factor k. A constant factor of 0.75 is used for the evaluation of optimistic protocols in [67]. However, our studies show that the optimal value of k is different for different simulation models and can only be determined when the particular simulation model is executed.

Figure 4.1 shows the execution times, T_e, α, β and the product $\alpha\beta$ for two different simulation models (see section 4.4) using different values of k ranging from 0.1 to 1.0. While the PHold model seems to perform well over a range of k values from 0.4 to 0.9, the manufacturing model yields the best performance when $k = 0.4$ is used. The manufacturing model executed using $k = 0.4$ needs 4233 supersteps to complete the simulation. Compare this to the run with $k = 1.0$ which only uses 2499 supersteps, the additional 1734 supersteps translate to only $39ms$ on a SGI Original 2000 with a barrier synchronization cost l=22.7μs. The extra 1734 supersteps significantly improve $\alpha\beta$ and lead to a 13% or 48s reduction in the execution time of the simulation.

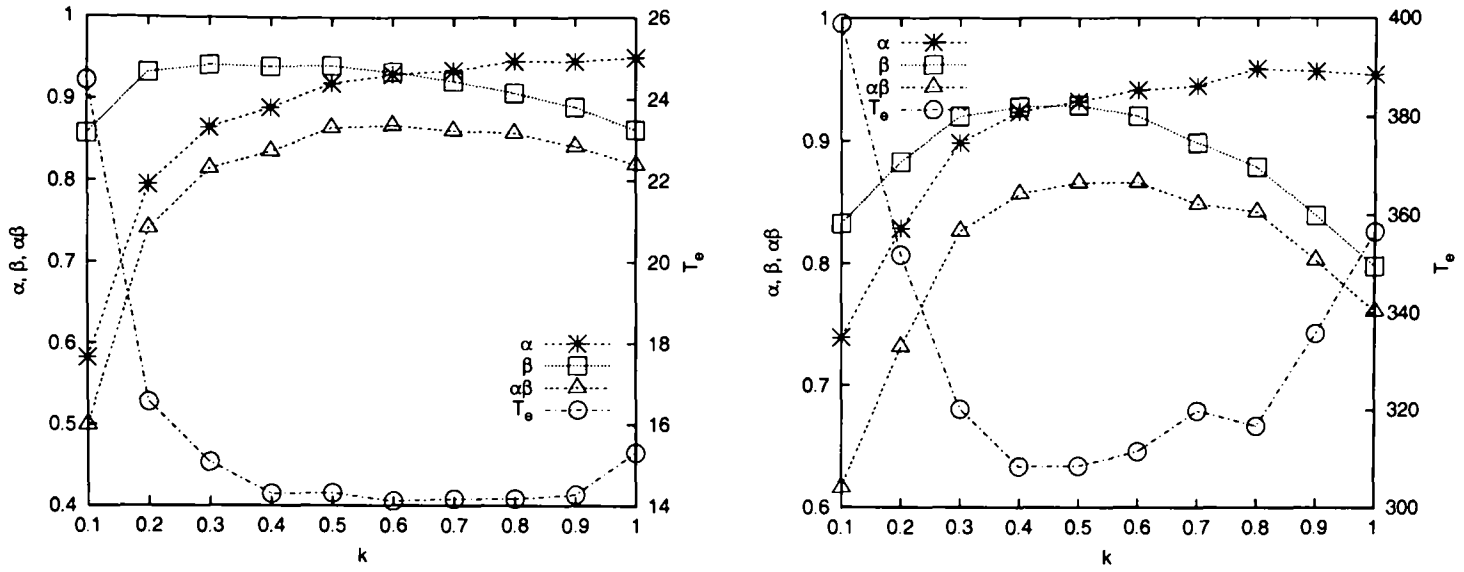


Figure 4.1: Performance of (a) PHold Model and (b) Manufacturing Model with different values of k.

4.3 Adaptive BSP-TW

In this section, an adaptive BSP-TW scheme that seeks to minimize load-imbalance between processors across supersteps as well as reduce the number of events rolled back is proposed.

The scheme proposed is heuristic in nature. If the total number of pending events for a processor on any superstep is n_p, the scheme seeks to adaptively optimize the value of γ ($\frac{1}{n_p} \leq \gamma \leq 1.0$) such that executing γn_p events for each of the n_g supersteps between successive GVT updates maximizes a cost function $\phi(\gamma)$ defined as follows:

$$\phi(\gamma) = \frac{\Delta\text{GVT}\hat{\alpha}}{1 - \hat{\beta}} \tag{4.6}$$

where ΔGVT is the increase in GVT between successive GVT updates. $\hat{\alpha}$ and $\hat{\beta}$ measure the corresponding values of α and β for the n_g supersteps between successive GVT updates.

The objective of $\phi(\gamma)$ is to advance GVT as fast as possible while at the same time keeping the number of events rolled back to a minimum as well as balance the load of the processors across supersteps.

The quadratic fit method [64] is used to optimize $\phi(\gamma)$. The basic idea is to find three values of $\gamma : \gamma_i < \gamma_j < \gamma_k$ with $\phi(\gamma_i) \leq \phi(\gamma_j) \geq \phi(\gamma_k)$. Then the point which maximizes

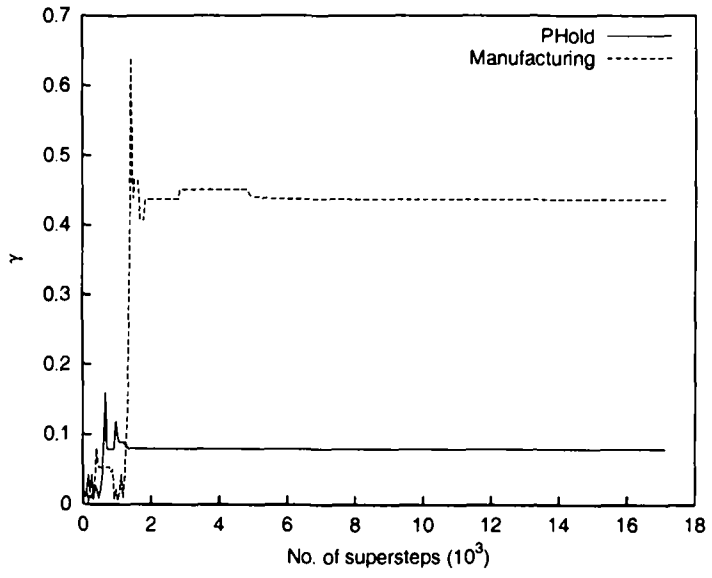


Figure 4.2: Optimizing γ for the PHold and Manufacturing Models.

a fitted quadratic can be computed and used as the next value of γ. The routine used to find an initial pair of γ_i and γ_k which "bracket" a maximum is similar to the routine found in [80]. Each $\phi(\gamma)$ is measured by executing the simulation for one GVT update interval using the event limit γn_p for the n_g supersteps within the interval.

Figure 4.2 shows some sample values of γ obtained by the adaptive BSP-TW for the two models shown in Figure 4.1. The adaptive scheme begins with an initial value of γ=0.01 and rapidly converges to a stable state for the rest of the simulation.

4.4 Experiments

In this section, experimental results for using different simulation models to compare the performance of the adaptive BSP-TW against the original BSP-TW are presented. The following abbreviations for the different versions of BSP-TW are used:

BSP-TW$_o$: Original BSP-TW using k=0.75;
BSP-TW$_a$: BSP-TW using adaptive tuning.

The experiments that follow were conducted on a 86-node 195MHz Origin 2000 system. The elapsed time reported is the average of five runs. For all the experiments, a spin-loop of 25μs is used to model fine event granularity and a GVT update interval of 50 supersteps is used. The simulation run length for both the symmetric and asymmetric PHold experiments is 10^3 time unit. A simulation run length of 10^4 time unit is used for both the

manufacturing model and the arbitrary flow model.

4.4.1 PHold Model

The PHold model [30] is widely used to benchmark parallel simulation protocols. The PHold model consists of P processors with D_p LPs per processor and D initial events per LP. The event population per processor N is hence given by $N = DD_p$. Events are uniformly distributed among the LPs at the beginning of simulation. Each LP will receive an event, increment its time-stamp, and send the event to another LP. The time-stamp increment follows an exponential distribution with mean 1.0 and the LP to send the event to is chosen from a uniformly distributed random variable in the range (0 ... n-1).

In [67], the PHold experiments are conducted on eight processors and a fixed event population of N=128 events per processor. Experiments are conducted for different values of D. The results presented in Figure 4.1a are for the runs with D=128.

Figure 4.3 shows the results of using BSP-TW$_o$ and BSP-TW$_a$ on the PHold model. Although BSP-TW$_o$ performs marginally better in terms of the α values, BSP-TW$_a$ is able to obtain substantially higher value of β than BSP-TW$_o$. The combined effect is shown in the graph for $\alpha\beta$ whereby the performance of BSP-TW$_a$ is consistently better than BSP-TW$_o$, confirming the trend of the elapse time performance. The trade-off for achieving a higher value of $\alpha\beta$ is depicted in the two to three times more supersteps required by BSP-TW$_a$ to complete these simulation runs.

The set D=128 is interesting because the values of α and β for both algorithms are quite close but BSP-TW$_a$ completed the simulation using much fewer supersteps. The improvement in elapse time using BSP-TW$_a$ is very small since the reduction of 5304 supersteps translates to an improvement of only 120ms.

In the second set of experiments, we introduce asymmetry into the PHold workload by including source processes into the PHold model. Each of the source processes periodically generates an event and sends it randomly to one of the PHold processes with a time-stamp increment, and then sends another event to itself. The PHold nodes in the asymmetric model behave as the nodes in the symmetric PHold model except that they never send events to the source processes. To prevent an explosion in the number of events in the system, events received by the PHold processes are not forwarded to another process if these events originate from a source process.

The experiment setup is identical to that of the symmetric PHold model. A total of eight

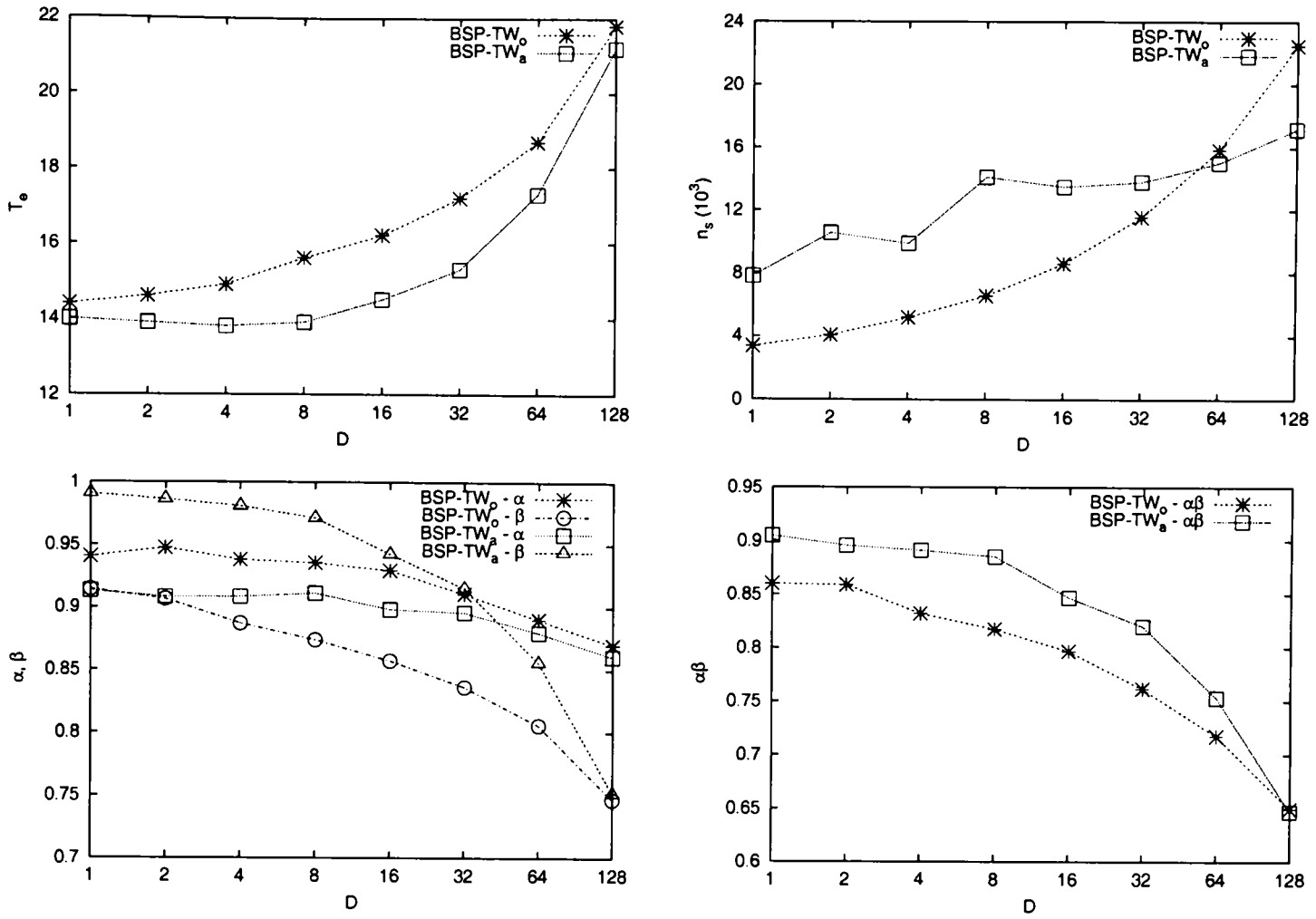


Figure 4.3: Comparison of BSP-TW$_o$ and BSP-TW$_a$ on PHold Model.

processors are used. One of the processors is designated to hold all the source processes, while the other seven processors hold normal PHold nodes. The processor that holds all the source processes is never rolled back.

Figure 4.4 shows the results of using BSP-TW$_o$ and BSP-TW$_a$ on the asymmetric PHold model. The graph for α shows that BSP-TW$_o$ suffers from severe load-imbalance. Comparatively, BSP-TW$_a$ is able to maintain a higher load-balance for all the runs. In terms of β, BSP-TW$_a$ minimizes the number of events rolled back for all values of D except for the set D=128. In this case, BSP-TW$_a$ also completes the simulation using relatively fewer supersteps compared to BSP-TW$_o$. The graph for $\alpha\beta$ also shows that BSP-TW$_a$ performs consistently better than BSP-TW$_o$.

4.4.2 Manufacturing Simulation

In this section, the impact of reducing rollback and load-imbalance using the BSP-TW$_a$ protocol is further examined using the manufacturing simulation model described in [53]. The manufacturing model consists of different entities of a production line with assembly

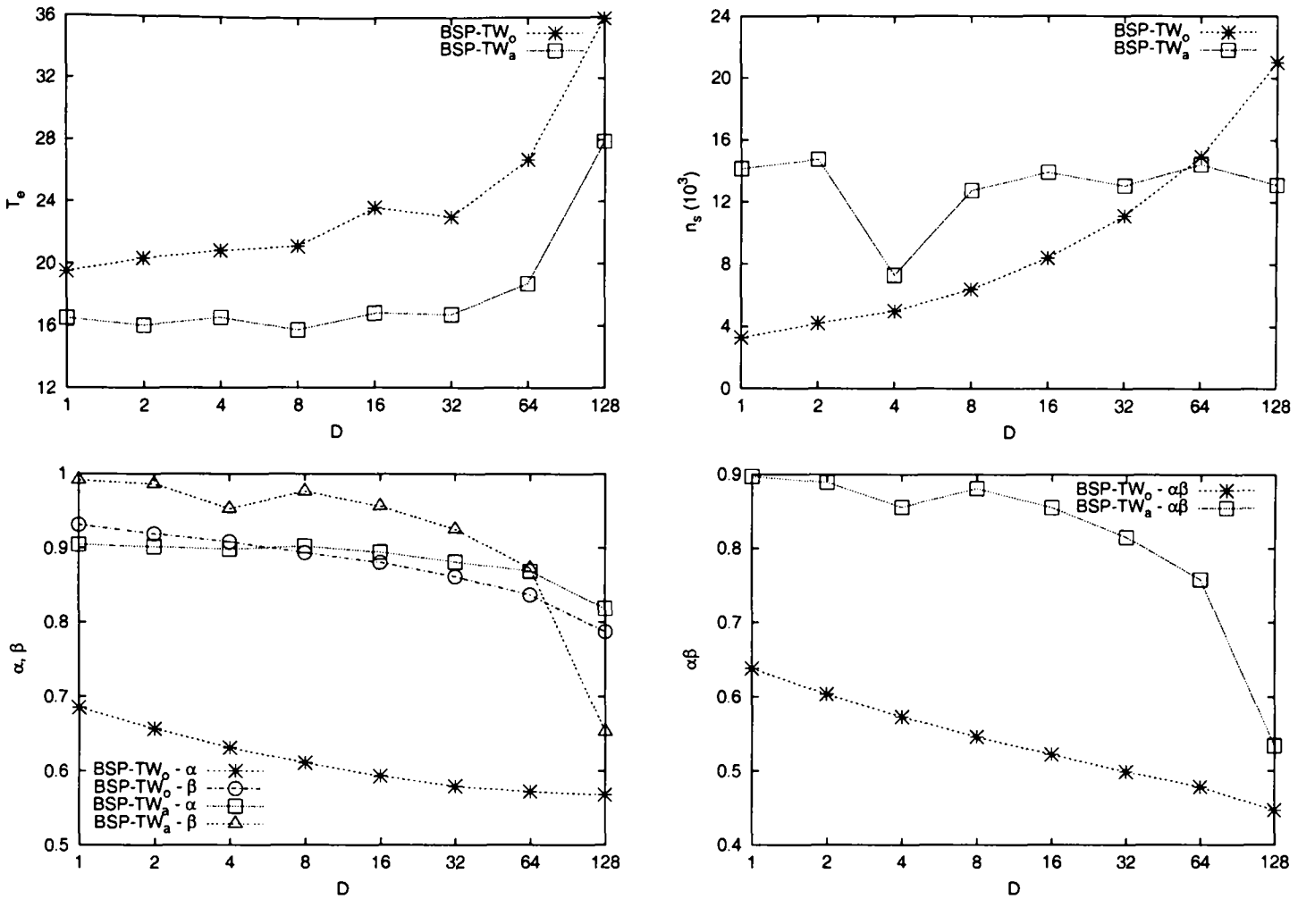


Figure 4.4: Comparison of BSP-TW$_o$ and BSP-TW$_a$ on Asymmetric PHold Model.

and test facility.

Figure 4.5 shows the layout of a production line and an assembly and test facility. Each production line comprises multiple stages and each stage consists of a processing/control station pair. At the end of each stage, some products will be returned to the beginning of the stage for rework. At the end of a production line, some products will also be sent to the beginning of the production line for rework. The products to be sent for rework are determined by the corresponding rework probability on each of the `fork` node.

In the assembly line, products from multiple production lines are uniformly routed to one of the many `join` nodes, each feeding into an assembly station. The purpose of the `join` node is to ensure that at least one product from each production line is available before the assembly station starts assembling them. The assembled products are then collected and routed to one of the testing stations for final testing.

The configuration of the manufacturing model consists of a total of seven production lines. Each production line consists of 100 production stages. The assembly and test facility consists of 100 assembly stations and 100 testing stations. There are a total of

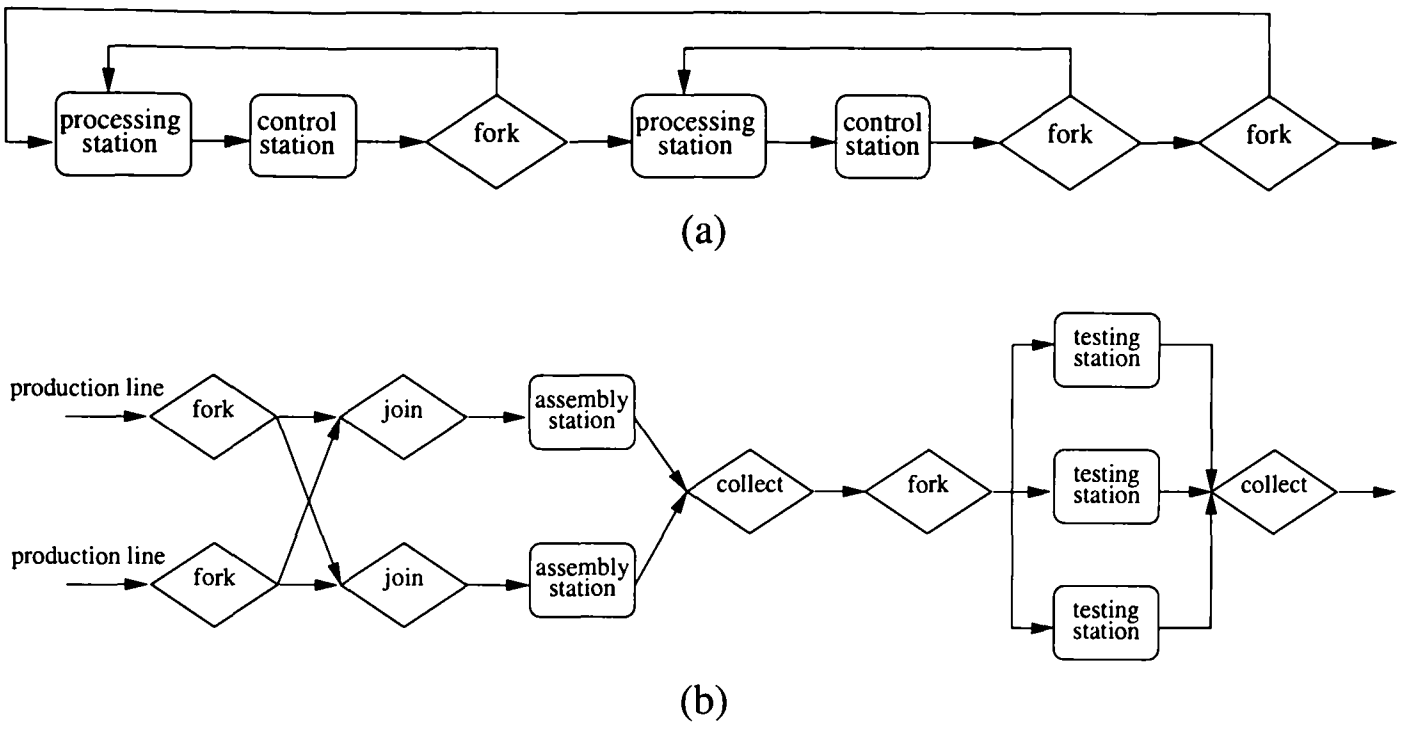


Figure 4.5: Layout of (a) a Production Line and (b) an Assembly and Testing Facility.

2417 simulation objects in this model.

For this set of experiments, all inter-arrival times and service times use fixed values. A block partition of block size 25 is used to partition simulation objects onto the processors.

Table 4.1 shows the performance of BSP-TW$_o$ and BSP-TW$_a$ on the manufacturing model in terms of elapse time T_e, speedup S_p, number of supersteps n_s, α, β and $\alpha\beta$. The experiment is repeated using 4, 8 and 16 processors. The experimental results shown in Figure 4.1b are for the same model executed using four processors.

In this set of experiments, although the required number of supersteps using BSP-TW$_a$ is about four to eight times more than BSP-TW$_o$, there are very few rolled-back events and BSP-TW$_a$ is able to maintain load-balance between processors across supersteps. In fact, as shown in Figure 4.1b, the maximum values of β achieved by using different values of k with BSP-TW$_o$ is 0.93, while BSP-TW$_a$ is able to achieve a β value close to 1.0. By using more supersteps, BSP-TW$_a$ is able to yield a 20% to 36% higher speedup compared to BSP-TW$_o$.

4.4.3 Arbitrary Flow Network Model

Unlike the asymmetric PHold model, where the number of unprocessed messages is always constant, the arbitrary flow network model [21] is a model with asymmetric workload where the number of pending events can grow without bound and there is a possibility

		BSP-TW$_o$	BSP-TW$_a$
4 CPU	T_e/S_p	315.11/1.97	242.04/2.36
	n_s	3009	25092
	α	0.94	0.97
	β	0.89	1.00 (0.9995)
	$\alpha\beta$	0.84	0.97
8 CPU	T_e/S_p	171.73/3.62	124.87/4.87
	n_s	3075	10761
	α	0.87	0.94
	β	0.86	1.00 (0.9992)
	$\alpha\beta$	0.75	0.94
16 CPU	T_e/S_p	95.30/6.55	71.69/8.93
	n_s	3321	13719
	α	0.80	0.84
	β	0.83	0.99
	$\alpha\beta$	0.68	0.83

Table 4.1: Comparison of BSP-TW$_o$ and BSP-TW$_a$ on Manufacturing Model.

of very long rollback. This model was also used in [65] to evaluate the performance of probabilistic synchronization scheme in conjunction with Time Warp.

The experiment setup for this model is the same as the one described in [21], with one source, one sink and eight application nodes. The source node generates events and sends them to the application nodes; the sink node receives events from the application nodes.

The application nodes communicate by sending time-stamped messages of two types: *propagating* and *non-propagating*. A propagating message triggers one or more additional messages to be sent by the application node processing it. Non-propagating messages do not result in additional communication and are used to model data-transfer. If several new messages are sent as a result of processing a propagating message, only one of them will be propagating.

The eight application nodes are divided into six *fast* nodes and two *slow* nodes. The slow nodes have time-stamp increment that is exponentially distributed with a mean of 1; whereas the fast nodes have time-stamp increment that is exponentially distributed with a mean of 100. A fast or slow node communicates with its own group with a probability of 0.2 and the other group with a probability of 0.6.

A total of 10 processors are used for this experiment. The source, sink and application nodes are each allocated onto separate processors. Two different event generation rates for the source node using exponential distribution with mean 50 and 100 are experimented.

Table 4.2 shows the comparison of BSP-TW$_o$ and BSP-TW$_a$ using the adaptive flow

Event rate		BSP-TW$_o$	BSP-TW$_a$
50	T_e	230.79	170.46
	n_s	34680	25449
	α	0.40	0.38
	β	0.77	0.74
	$\alpha\beta$	0.31	0.28
100	T_e	99.14	82.51
	n_s	25959	18564
	α	0.41	0.49
	β	0.71	0.67
	$\alpha\beta$	0.29	0.33

Table 4.2: Comparison of BSP-TW$_o$ and BSP-TW$_a$ on Arbitrary Flow Network Model using different event rates for the source node.

network model. As indicated by the α values for both BSP-TW$_o$ and BSP-TW$_a$, the workload for the simulation model is severely un-balanced.

Although BSP-TW$_o$ achieves higher α and β for the run with event rate of 50, BSP-TW$_a$ completed the simulation using much fewer supersteps than BSP-TW$_o$. For both experiments, BSP-TW$_a$ outperforms BSP-TW$_o$ in terms of simulation elapse time. Timing measurements also reveal that BSP-TW$_a$ has much balanced supersteps in terms of elapse time in both set of experiments. The reason for this is that the difference in granularity for the different types of events is not taken into account by α. Although the PHold models and the manufacturing model all have events of similar granularity, the same is not true for the arbitrary flow network model. In this model, a propagating event has an average event granularity of $70\mu s$ while a non-propagating event has an average event granularity of $30\mu s$. The calculation of α gives equal weightage to both types of events.

4.5 Related Work

The quadratic fit method had also been used in [42] to optimize the performance of an adaptive protocol. This protocol uses a concept called "real-time blocking window" (RTBW) on each input channel of an LP to select events for execution. The size of this window is adaptively adjusted to maximize the rate of increase in simulation time for each channel.

A survey of adaptive techniques for parallel simulation protocols can be found in [20]. The issue of event granularity has also been examined in [81], where a scheduling algorithm that reduces the average granularity of rolled-back events is described.

4.6 Summary

An adaptive scheme for BSP-TW that attempts to minimize both load-imbalance as well as the proportion of events rolled back has been presented. The scheme is shown to yield better performance on the different simulation models experimented. The experimental results obtained for the arbitrary flow network model suggest that the BSP-TW$_a$ can be improved further to take event granularity into consideration when computing α.

Past study [38] had shown that minor variation in load in TW results in drastic changes in performance. As it is, the proposed adaptive scheme only tries to minimize load-imbalance statically. It is believed that further performance improvement can be achieved using dynamic load-balancing techniques on simulation model such as the arbitrary flow network model that is inherently un-balanced in load. This issue will be studied in the next chapter.

Chapter 5

Dynamic Load-Balancing of BSP-TW

5.1 Overview

In this chapter, we develop a dynamic load-balancing algorithm for the BSP-TW protocol. Using the BSP cost equation, we show that in order to achieve maximum performance from such a simulation, the dynamic load-balancing algorithm needs to take into account factors such as computation workload, communication workload as well as lookahead between processors.

The experiments described in this chapter are carried out without interruption from external workload. In the next chapter, we enhance the dynamic load-balancing algorithm to handle situations in which external workload is present.

The sections in this chapter are arranged as follows. Section 5.2 describes the motivation for carrying out dynamic load-balancing in BSP-TW. This is followed by a brief survey of related work carried out by other researchers on dynamic load-balancing for parallel simulation protocol.

In section 5.3, a theoretical model for dynamic load-balancing in BSP-TW is developed. Based on this model, we develop the first extension for BSP-TW in section 5.4 to handle the balancing of computation workload. The experiments described in section 5.4.1 examine the different parameters that affect the performance of the dynamic load-balancing algorithm.

In section 5.5, we extend the dynamic load-balancing algorithm to handle the balancing of communication workload. We further enhance the dynamic load-balancing algorithm in

section 5.6 to enable the optimization of lookaheads. The advantage of this optimization is illustrated using a manufacturing model with serious performance issues due to the presence of zero lookahead.

We conclude this chapter in section 5.7 with experiments on the arbitrary flow network model using the dynamic load-balancing algorithm that has been developed.

5.2 Background

There are two main reasons for the use of dynamic load-balancing and process migration in parallel simulation system. The first reason is to allow consistent performance to be achieved from arbitrary simulation model with inherently irregular workload. The second reason concerns the fact that computing resources may not be fully dedicated and the presence of external workload can severely affect the execution of the simulation model.

In this section, we look at several factors that motivate the study of dynamic load-balancing and process migration in the field of parallel simulation.

5.2.1 Static Partitioning

Static partitioning is a method commonly used for assigning work to different processing elements. The basic premise is that the workload of the individual component of the simulation model can be quantified statically and this workload distribution remains relatively unchanged as the simulation progresses.

There are a few drawbacks in this approach. First, it is generally hard to quantify the workload of the individual components of any given simulation model. Most of the time, in-depth domain knowledge is required to arrive at a reasonable estimate of the initial workload distribution of a simulation model.

In some of the studies, the simulation model is run sequentially for a short period of time so that the workload for different components of the system can be measured. The measured workload is then used as an initial estimate to partition the simulation model.

The definition of workload is different in different contexts. Some studies use workload that are based solely on local computation and ignore any effects on the communication aspect; while other studies take into account workload due to both computation and communication.

5.2.2 Time Varying Internal Workload

Even if the initial workload can be quantified accurately, the state of the system is bound to evolve and change over time. This dynamic and constant changing behaviour of the simulation system is precisely the reason why simulation models are studied in the first place.

The dynamic behaviour of the simulation system means that an initial balanced workload may become grossly unbalanced over time. To automatically adjust to this dynamically changing workload, the simulation system must be equipped with dynamic load-balancing facilities that are able to re-balance the workload whenever the needs arise.

5.2.3 Time Varying External Workload

Most experiments with parallel simulation protocols are conducted on dedicated parallel systems. In most cases, extra measures are taken to ensure that the system is free from intervention from unwanted external processes or workload.

The emergence of distributed computing clusters, however, requires a radically new treatment of parallel programming. The computing resources in the distributed cluster are shared among many users and in most cases cannot be used in a dedicated fashion. A parallel simulation job on multiple processing stations in such a cluster will need to anticipate unscheduled interruption due to external workload from other users.

A (statically) well-balanced workload could be distributed onto the available computing resources at the start of a simulation run with all the computing resources free of any external workload. However, as time elapsed external load (such as users logging in to run other jobs) sets in. The initial balanced partition may gradually become grossly unbalanced. The simulation system must have the dynamic load-balancing facilities that allow simulation workload to be migrated to less loaded machines.

The same holds true when more computing resources become available over time. The simulation system should be able to make use of additional resources as they become available, as well as be able to discard the use of certain resources as they become un-economical to use or unavailable.

5.2.4 Processor Utilization under Optimistic Protocol

Unlike most other problems in the areas of load-balancing and process migration, parallel simulation has a peculiar characteristic that processor utilization is not always an accurate measure of the actual workload of different parts of the simulation model. This problem is especially true for optimistic protocols. Optimistic protocols that allow erroneous processing to take place and correcting them subsequently could run into situations in which a processor is heavily utilized but is actually spending most of the processing cycles working on events that are later rolled back.

Various metrics have been proposed to assess the "actual" workload of a processor in optimistic simulation. Reiher and Jefferson proposed an "effective utilization" metric based on the fraction of the time spent by the processor on work that are eventually committed [82]. They use this metric to migrate processes from processors with high effective utilization to those with low effective utilization. Glazer and Tropper described a metric based on time slices [39]. A time slice is a metric proportional to the ratio of the amount of computation time required by a process over the advance of its simulation time. They presented speedup improvement ranging from 12% to 49% on three different simulation models running on a simulation multiprocessor environment.

Burdorf and Marti presented a strategy based on local simulation (virtual) times [7]. By periodically computing the mean and standard deviation of all the LPs' local simulation times in the system, objects from processors that are far behind (in simulation time) are moved to processors that are further ahead (in simulation time). The aim is to "slow down" the fast processor in order to reduce the amount of rollback. Burdorf and Marti observed that this scheme leads to better load-balance in the presence of external workload.

A metric based on "virtual time progress" is proposed by Schlagenhalf et al. [86] to balance the load of a VLSI circuit application on a distributed Time Warp simulation in the presence of external workload. "Virtual time process" reflects how fast a simulation process continues in virtual time. Using this metric, the authors reported a reduction of 24% in simulation time on a logic circuit model.

In a study reported in [10], Carothers and Fujimoto presented an approach for background execution of Time Warp. The scheme allows a Time Warp system to execute in background and consume unused CPU cycles across a collection of heterogeneous machines. The metric used is "Processor Advance Time" (PAT), which reflects the amount of real time needed to advance the virtual time of a logical process by one unit. A personal communication service network model is used in this study. The experimental results

showed an improvement of up to 45% in the presence of external workload.

Another metric based on workload of clusters was proposed by Avril and Tropper in [1]. In this work, the workload of a processor is the sum of the workload of all the clusters in the processor. The workload of a cluster is simply the number of events processed by the cluster since the last load-balance. The algorithm iteratively matches the most heavily and lightly loaded processors and transfer clusters between them to balance the load. Experimental results on two benchmark digital circuit models show 40-100% improvement in throughput using this technique.

A recent paper by Choe and Tropper [16] discusses a metric based on space-time product. The space-time product of a processor at time t is defined as the product of the local simulation time of the processor and the total memory allocated for events and states. Using this metric, the algorithm attempts to control the difference between the space-time product of the processors involved in the simulation. The authors note that this strategy is able to balance the memory used by the processors as well as to keep processors close to one another in virtual time so as to reduce the possibility of rollbacks. The dynamic load-balancing algorithm in the Time Warp system is used together with a flow control scheme to regulate the sending of messages between processors. Experiments carried out on a shuffle ring network as well as a personal communication system model show that the dynamic load-balancing algorithm together with the flow control scheme is able to provide 13-23% performance improvement.

A recent comparison of some of the metrics used for dynamic load-balancing of optimistic simulation can be found in [23]. The survey concludes that "the effectiveness of a dynamic load-balancing algorithm depends strongly on the nature of the model which is being simulated".

5.3 Dynamic Load-Balancing Cost Model

In order to develop efficient DLB algorithms for BSP-TW, a theoretical model that will allow the analysis of different approaches to dynamic load-balancing for BSP-TW needs to be constructed.

From equation (4.1) in section 4.2, the BSP cost for a BSP-TW algorithm S is

$$cost(S) = \sum_{i=0}^{n_s} (w(i) + gh(i) + l). \tag{5.1}$$

Although the cost model is relatively simple, we can immediately see that the performance of the BSP-TW algorithm depends on three terms: 1) computation balance; 2) communication balance; and 3) n_s, the total number of supersteps.

It should be noted that these three factors are not independent, but closely related to one another. Trying to improve one of them may not necessary lead to overall improved performance as the other two factors may be adversely affected. A good dynamic load-balancing algorithm needs to take into account all three terms and minimize them in order to achieve satisfactory result when performing load migration.

In order to study dynamic load-balancing in BSP-TW, we first consider a superstep n_0 in which GVT computation has just been completed. The load-balancing algorithm will now decide whether or not to perform dynamic load-balancing. This point of choosing if dynamic load-balancing should be carried out is termed as the *migration point*. A migration point will occur every λn_g supersteps ($\lambda \geq 1$) where n_g is the number of supersteps between each GVT computation.

If a decision has been made to perform dynamic load-balancing in superstep n_0, its effects on the simulation system will be assessed at the next migration point in superstep $n_0 + \lambda n_g$.

Let the value of GVT in superstep i be $\nu(i)$. Let $\delta\nu(i)$ represents the increase of GVT from superstep i to superstep $i + 1$ and $T(n_0)$ be the real time at the start of superstep n_0.

We need to consider both the case where load-balancing is not performed as well as the case where load-balancing is performed. The subscript n shall be used to represent the components in the BSP cost equation under *normal* operation *without* load-balancing and the subscript b to represent the components *with* load-balancing.

We will first consider the case whereby the decision has been made *not* to perform load-balancing at superstep n_0. We consider the state of the system at superstep $n_0 + \lambda n_g$. The value of GVT at superstep $n_0 + \lambda n_g$ is

$$\nu_n(n_0 + \lambda n_g) = \nu(n_0) + \sum_{i=n_0}^{n_0+\lambda n_g-1} \delta\nu_n(i). \tag{5.2}$$

The time taken to execute the λn_g supersteps is given by

$$C_n = \sum_{i=n_0}^{n_0+\lambda n_g-1} \Big(w_n(i) + gh_n(i) + l \Big). \tag{5.3}$$

The real time at the start of superstep $n_0 + \lambda n_g$ is given by

$$T_n(n_0 + \lambda n_g) = T(n_0) + C_n. \tag{5.4}$$

Let $\nu' = \frac{\delta\nu}{\delta t}$ represent the rate of advance of GVT in real time (GVT rate). The GVT rate at superstep $n_0 + \lambda n_g$ is given by

$$\nu'_n = \frac{\nu_n(n_0 + \lambda n_g)}{T_n(n_0 + \lambda n_g)}. \tag{5.5}$$

Now consider the case in which a decision is made to perform load-balancing in superstep n_0. The state of the system is given by

$$\nu_b(n_0 + \lambda n_g) = \nu(n_0) + \sum_{i=n_0}^{n_0+\lambda n_g-1} \delta\nu_b(i). \tag{5.6}$$

The time taken to execute the λn_g supersteps is given by

$$C_b = \sum_{i=n_0}^{n_0+\lambda n_g-1} \Big(w_b(i) + gh_b(i) + l \Big) + C_l, \tag{5.7}$$

where C_l is the cost of performing the load migration.

The real time at the start of superstep $n_0 + \lambda n_g$ is given by

$$T_b(n_0 + \lambda n_g) = T(n_0) + C_b. \tag{5.8}$$

The GVT rate is given by

$$\nu_b' = \frac{\nu_b(n_0 + \lambda n_g)}{T_b(n_0 + \lambda n_g)}. \tag{5.9}$$

In order for the dynamic load-balancing decision to be effective, the necessary condition is

$$\nu_b' \geq \nu_n'. \tag{5.10}$$

This condition can be achieved by several means. To maximize the value of ν_b', we can try to maximize the value of $\nu_b(n_0 + \lambda n_g)$, or we can try to minimize the value of $T_b(n_0 + \lambda n_g)$. From equations (5.7) and (5.8), $T_b(n_0 + \lambda n_g)$ can be minimized by balancing computation workload and communication workload. From equation (5.6), $\nu_b(n_0 + \lambda n_g)$ can be maximized by maximizing the GVT increment between supersteps. This can be achieved by improving the lookaheads between processors.

5.3.1 Steps to Migrate Simulation Objects

The process of migrating simulation objects from one processor to another processor involves several steps. Consider an object o_i being migrated from processor P_m to P_n. Table 5.1 outline the different steps for migrating an object from one processor to another.

The migration algorithm first rolls back the simulation object o_i to GVT. This allows o_i to be migrated to P_n without the need to migrate any past state information. A similar approach was used in [10] where the entire simulation computation is rolled back to GVT. In our algorithm, only the objects that are to be migrated are rolled back to GVT. This prevents simulation objects not involved in the migration from being unnecessarily rolled

migrate(int o_i, int P_m, int P_n)
1. Rollback object o_i to GVT
2. Save the state of o_i and send the state of o_i to processor P_n
3. Broadcast new processor mapping of o_i to other processors
4. For all events e_j in P_m's event-list destined for object o_i, forward e_j to P_n
5. For all events external e_j subsequently received that are destined for object o_i, forward e_j to P_n

Table 5.1: Steps for Migrating Objects between Processors.

back. Also, migration can only happen immediately after a GVT update. This has the effect of reducing the number of rolled-back events due to migration.

In the second step of the migration algorithm, processor P_m saves the state of o_i and sends the state of the o_i to P_n. The new processor mapping of o_i is broadcast to other processors in step 3. In step 4, any events in the P_m's event-list that are destined for o_i are forwarded to P_n.

The superstep structure of BSP-TW requires that the update of the new mapping of o_i can only reach the other processors at the start of the next superstep. As a result, some events destined for o_i may still be sent by other processors and arrive at P_m in the next superstep. Processor P_m will need to forward these events accordingly to P_n based on the new processor mapping of o_i.

When processor P_n receives the state of object o_i in the next superstep, it restores the state of object o_i to complete the migration process.

5.4 Balancing Computation Workload

We now take a preliminary look at a simple approach to dynamic load-balancing. In this section, we are only interested in balancing the computation workload of the simulation model. Several different strategies will be examined in the subsequent sections.

5.4.1 Algorithm for Balancing Computation Workload

Most load-balancing algorithms will be triggered whenever load-imbalance is detected. For our system, we define a computation load-imbalance metric, WB, to represent the load-

imbalance in computation workload among the processors. Suppose a migration point is set at superstep s. At the end of superstep $s-1$, each processor P_i broadcasts its total computation workload $P_i.wl$ to all the other processors. WB is then defined as follows:

$$WB = \frac{\max(P_i.wl) - \text{mean}(P_i.wl)}{\text{mean}(P_i.wl)}. \tag{5.11}$$

We define a load-imbalance threshold parameter ϵ such that computation load-imbalance is deemed to occur when $WB > \epsilon$. A new value for GVT rate, ν'_{new}, is also computed at each migration point. This value will be compared against the corresponding GVT rate, ν'_{old}, computed in the previous migration decision point. The process of migrating simulation objects will inevitably cause fluctuation in GVT rate and thus the system should only attempt to perform another migration when the GVT rate has stabilized. In our system, this happens when

$$(1.0 - \phi)\nu'_{old} \leq \nu'_{new} \leq (1.0 + \phi)\nu'_{old}. \tag{5.12}$$

where ν'_{old} and ν'_{new} are the GVT rates for the previous two GVT computation intervals. The stability threshold parameter ϕ prevents the system from migrating any more simulation objects until the system has stabilized.

We can now describe the dynamic load-balance algorithm for BSP-TW. We refer to this BSP-TW algorithm that balances computation workload as BSP-TW DLB$_c$. The BSP-TW DLB$_c$ algorithm attempts to balance computation workload by migrating simulation objects from heavily loaded processors to other processors that are less heavily loaded. The pseudo code in Figure 5.1 shows how the computation load-balancing algorithm is triggered based on the parameters λ and ϕ.

Figure 5.2 shows the algorithm for determining the amount of computation workload to move between processors. The procedure `balance_computation()` is iteratively executed by each processor. At each step, the processors with maximum computation workload, P_{max}, and the processor with minimum computation workload, P_{min}, are selected. The amount of computation workload, x, to be moved from P_{max} to P_{min} in order to equalize the workload between the two processors is computed.

Note that it is not always possible to move a fixed amount of workload away from a

```
bsp_begin();
[A] Initialization;
ν'_old := 0;
while GVT < SimEndTime do
    [B] Receive external events and process rollback;
    [C] Compute new GVT, perform fossil collection and
        compute new event limit n_e every n_g supersteps;
    [D] After each λ GVT computation:
        compute new GVT rate ν'_new
        if (1.0 − φ)ν'_old ≤ ν'_new ≤ (1.0 + φ)ν'_old then
          compute WB;
          if WB > ε then balance_computation() endif
        endif;
        ν'_old := ν'_new;
    [E] Execute n_e events;
    bsp_sync();
endwhile
bsp_end();
```

Figure 5.1: BSP-TW DLB_c Algorithm for Computation Load-balancing.

processor. A simple example is when a processor has only one simulation object and half of the workload is to be moved out of the processor. The only possible solution in this case is to move all of the workload out of the processor or move none at all. In the algorithm, the field $P_i.wlpo$ gives the average computation workload per simulation object on processor P_i. No migration will occur if the workload to be moved is smaller than $P_i.wlpo$.

Processor P_{max} then proceeds to move the required x amount of computation workload to processor P_{min}. Each processor then updates $P_m.wl$ and $P_n.wl$ accordingly. The iteration continues until the computation imbalance WB falls below ϵ or when the required workload to migrate is smaller than $P_{max}.wlpo$.

The variable *flag* is used to determine if any migration has taken place in the procedure. This will be used in section 5.5 for the purpose of balancing communication workload.

Figure 5.3 shows the algorithm for selecting objects for migration. The selection of objects to migrate works as follows. The computation workload of each object o_i between two migration points is given by the field $o_i.wl$. The list of objects is first sorted in descending order based on the wl field. This allows fewer simulation objects to be selected for migration. The procedure terminates when the required amount of workload is moved to the destination processor.

```
balance_computation()
    flag := false;
    while WB > ε do
        let P_max be the processor with the maximum computation workload;
        let P_min be the processor with the minimum computation workload;
        x := (P_max.wl + P_min.wl) / 2;
        if x ≥ P_max.wlpo then
            flag := true;
            computation_migrate(x, P_max, P_min);
        else
            break;
        endif
        P_max.wl := P_max.wl - x;
        P_min.wl := P_min.wl + x;
        compute WB;
    endwhile
    return flag;
```

Figure 5.2: Algorithm for Determining Amount of Computation Workload to Migrate.

```
computation_migrate(double load2Move, int srcProc, int destProc)
    if bsp_pid() = srcProc then
        sort object by wl in descending order;
        foreach local object o_i do
            if load2Move > 0 then
                load2Move := load2Move - o_i.wl;
                migrate(o_i, srcProc, destProc);
            endif
        endfor
    endif
```

Figure 5.3: Algorithm for Selecting Objects to Migrate for Computation Balancing.

5.4.2 Experiment with Computation Load-Imbalance in PHold Model

Having presented the BSP-TW DLB_c algorithm for balancing computation workload, we carried out experiments to determine the effectiveness of this algorithm on a BSP-TW system with computation load-imbalance.

The model used here is the symmetric PHold model described in section 4.4.1. A total of 1000 nodes are used in the model with one event per node. Eight 360MHz UltraSparc

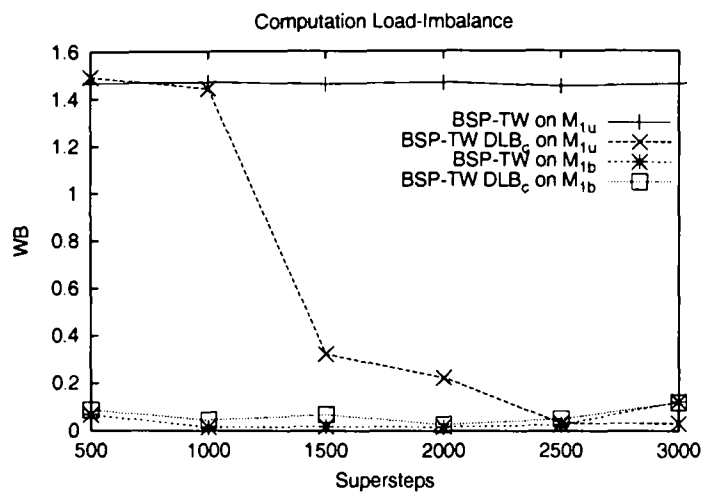


Figure 5.4: Example of Computation Load-Imbalance in PHold Model.

workstations connected by a 100Mbits local area network are used. We perform the experiments using two different kinds of partitions. The first partition method allocates equal number of nodes on each processor using round-robin allocation policy. We refer to this model as M_{1b}. The second partition method also uses the round-robin allocation policy but deliberately allocates three times more simulation objects on one of the processors in order to model computation load-imbalance. We refer to this model as M_{1u}. Note that both models are identical and yield the same simulation results.

Figure 5.4 shows the computation load-imbalance for the two models. The models are executed using the original BSP-TW as well as the BSP-TW DLB$_c$ algorithm. The DLB parameters used in this experiment are: ϕ=0.1, ϵ=0.2, λ=10. The GVT computation interval, n_g, for this experiment and all subsequent experiments in this thesis are fixed at 50 supersteps.

Both BSP-TW and BSP-TW DLB$_c$ algorithms managed to execute the model M_{1b} with very low computation load-imbalance (0.0-0.2). This is expected as the workload in the model is uniformly distributed among the processors and each node in the simulation model has about the same amount of workload. The model M_{1u} poses a serious problem to the original BSP-TW algorithm since one of the processors has three times more workload than the others. Since the BSP-TW algorithm has no provision to balance the workload, the load-imbalance for the simulation run using BSP-TW remains at a high level of 1.4-1.5 for the whole duration of the simulation.

However, the BSP-TW DLB$_c$ algorithm is able to handle the load-imbalance situation in model M_{1u} by recognizing the load-imbalance at superstep 1000. Simulation objects are migrated from the processor with the heaviest workload to other less loaded processors. The effect of load migration can be seen in superstep 1500. The load-balance of the system

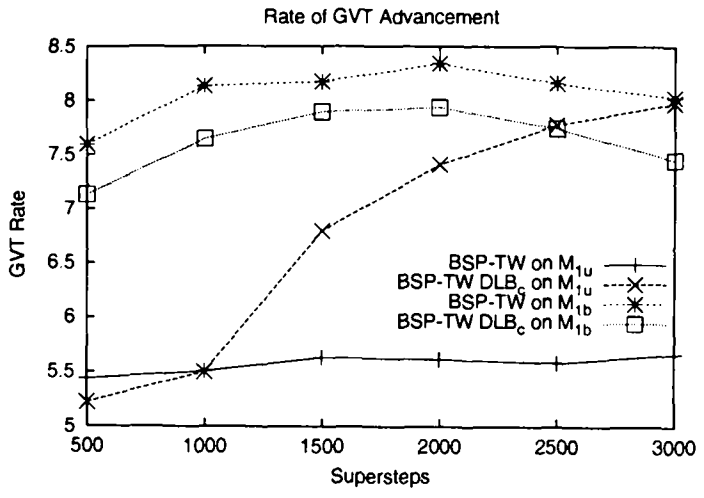


Figure 5.5: Rate of GVT Advancement in PHold Model.

Model	M_{1b}	M_{1u}
BSP-TW	125.2	186.4
BSP-TW DLB_c	131.5	146.2

Table 5.2: Execution Times (sec.) for M_{1b} and M_{1u} using BSP-TW and BSP-TW DLB_c.

drops from 1.4 in superstep 1000 to 0.3 in superstep 1500. The load-balance of the system improves further in the subsequent supersteps and reached a level similar to that for model M_{1b} in superstep 2500.

The improvement in load-balance of the system is also shown in the improvement in the rate of GVT advancement. Figure 5.5 shows the rate of GVT advancement at different migration points for both M_{1b} and M_{1u} executed using BSP-TW and BSP-TW DLB_c. Both BSP-TW and BSP-TW DLB_c are able to maintain high GVT rate for the model M_{1b}. For model M_{1u}, the GVT rate using BSP-TW remains at a low level between 5.0-5.5 for the whole simulation duration. As for the BSP-TW DLB_c algorithm, after the first migration at superstep 1000, the GVT rate improved to a level of 7.0 in superstep 1500. The GVT rate further improves to the same level as that of model M_{1b} in the subsequent supersteps.

Table 5.2 shows the execution times for these two models using both BSP-TW and BSP-TW DLB_c algorithms. The execution times in this experiment and the rest of the experiments in this thesis are obtained from the average of ten runs.

The small difference in the execution times between BSP-TW and BSP-TW DLB_c on the balanced model M_{1b} shows that the overhead in executing the DLB algorithm is small. While the execution time for model M_{1u} using BSP-TW is lengthened due to high computation load-imbalance, the BSP-TW DLB_c algorithm is able to reduce the load-imbalance significantly and improve the execution time of the simulation.

5.4.3 Effect of Varying ϵ

The parameter ϵ is used to determine the computation load-imbalance level at which to trigger the computation load-balancing. To see the effect of varying ϵ has on the system, we repeated the experiment using different values of ϵ. Figure 5.6 shows the load-imbalance and the corresponding GVT rate for model M_{1u} executed using BSP-TW DLB$_c$ with different values of ϵ. For this experiment, the parameters are fixed at ϕ=0.1 and λ=10.

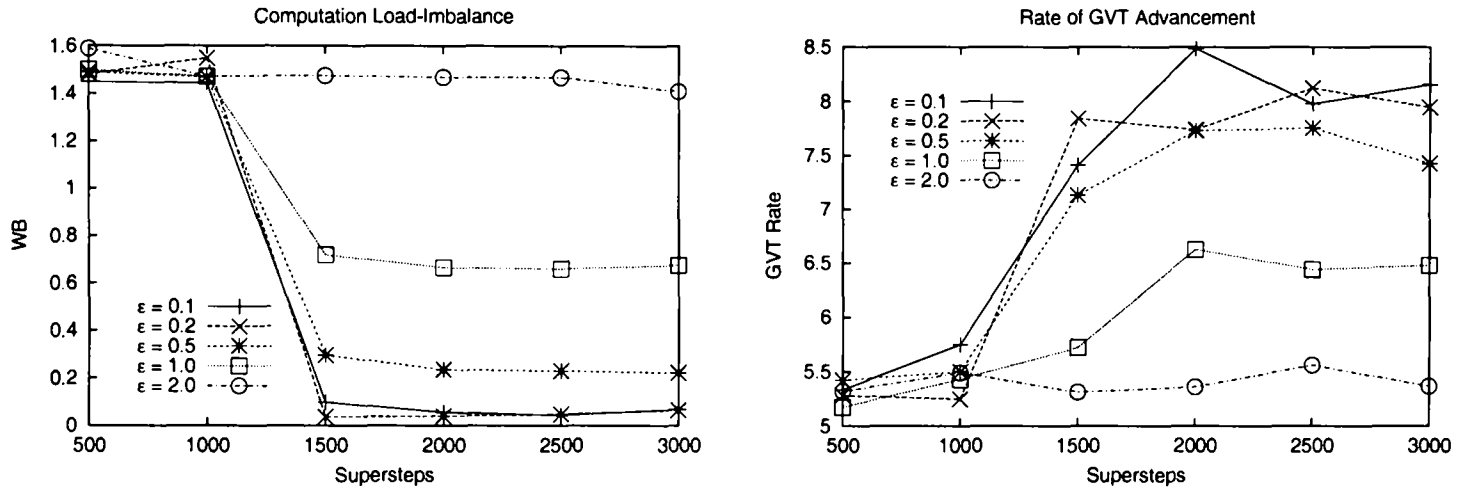


Figure 5.6: Computation Load-Imbalance and Rate of GVT Advancement for M_{1u} with Different Values of ϵ.

With ϵ=2.0, the DLB algorithm is not activated at all since the load-imbalance of the system stays well below 2.0. The runs with ϵ=1.0 did improve the load-imbalance slightly. However, load-migration stops once the load of the system goes below 1.0. The performance improvement gained moving from ϵ=0.2 to ϵ=0.1 is very small. Table 5.3 shows the execution times of the model using different values of ϵ. The performance obtained for ϵ=0.1, 0.2 and 0.5 are quite similar.

ϵ	0.1	0.2	0.5	1.0	2.0
Execution time	144.5	150.0	151.2	170.7	185.0

Table 5.3: Execution Times (sec.) for M_{1u} using Different Values of ϵ.

5.4.4 Effect of Varying ϕ

The stability threshold value ϕ plays an important role in stabilizing the system between each migration point. In this set of experiments, the values of ϕ are varied. The other parameters are fixed at $\epsilon = 0.2$ and $\lambda = 10$. Figure 5.7 shows the load-imbalance and

the corresponding GVT rate of the unbalanced model M_{1u} executed using BSP-TW DLB$_c$ with different values of ϕ.

Setting the threshold at 0.01 means that the GVT rate of the previous computation interval must be within $\pm 1\%$ of the GVT rate of the computation interval before it. This narrow requirement results in the migration being delayed until superstep 2000. For values of ϕ greater or equal to 0.05, the performance of the model are almost identical. This is due to the fact that once migration is performed at superstep 1000, the load-imbalance of the system drops well below the threshold value imposed by ϵ=0.2, and no further migration is necessary. Table 5.4 shows the execution times of the model M_{1u} using different values of ϕ.

The execution times of the model show that the worst performance is obtained using ϕ=0.01. The runs using $\phi \geq 0.05$ produce similar results.

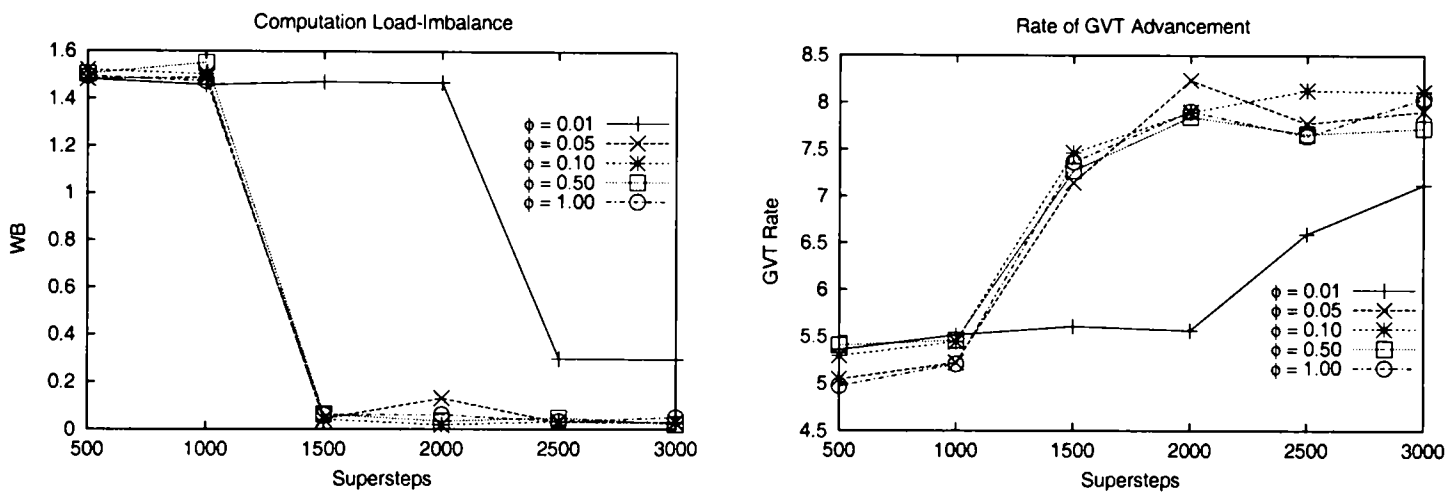


Figure 5.7: Computation Load-imbalance and Rate of GVT Advancement for M_{1u} with Different Values of ϕ.

ϕ	0.01	0.05	0.1	0.5	1.0
Execution time	163.8	144.4	144.8	146.9	148.0

Table 5.4: Execution Times (sec.) for M_{1u} using Different Values of ϕ.

5.4.5 Effect of Varying λ

We now look at the effect of varying λ, the number of GVT computations between each migration point. A small value of λ may not allow the system to have enough time to stabilize between each migration point. A large value of λ gives more time for the system

to settle down after each migration. However, the system may not be responsive enough to react to rapid change in system load.

Figure 5.8 shows the load-imbalance and the corresponding GVT rate of the model M_{1u} executed using BSP-TW DLB_c with different values of λ. For this experiment, the other parameters are fixed at ϵ=0.2 and ϕ=0.1.

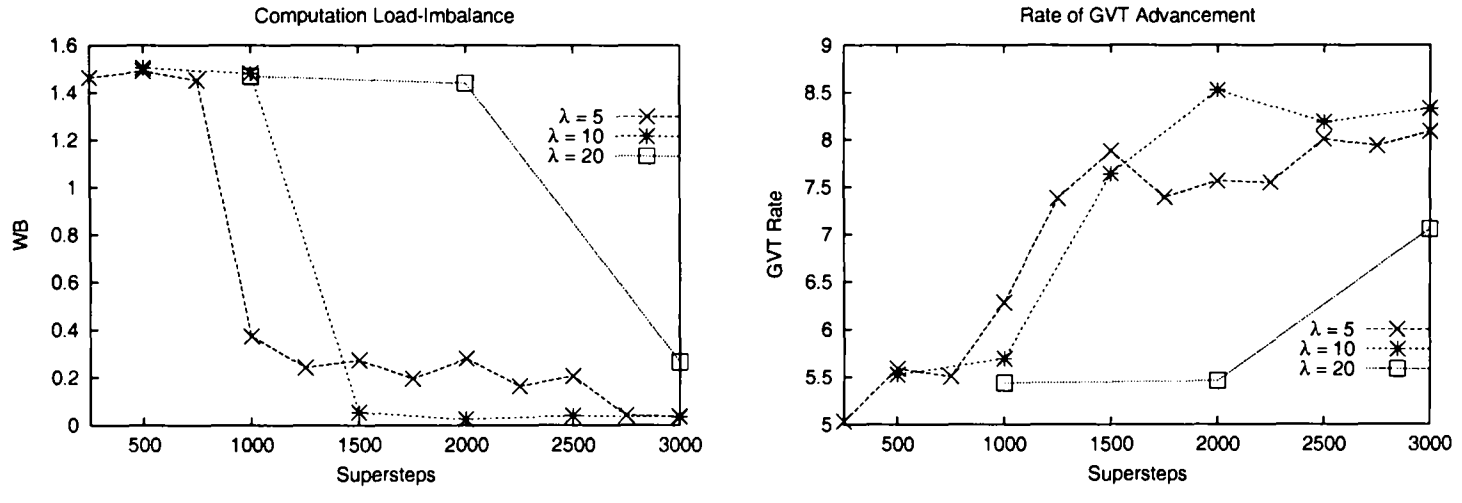


Figure 5.8: Computation Load-imbalance and Rate of GVT Advancement for M_{1u} with Different Values of λ.

With λ=20 and the GVT computation interval n_g set at 50 supersteps, migration decision only takes place every 1000 supersteps. The system is slow to react to the imbalance in computation workload. As a result, the computation workload remains unbalanced until superstep 2000 where the first actual migration occurs.

On the other hand, the runs with λ=10 made the first migration at superstep 1000 and is able to maintain a low computation load-balance for the rest of the simulation. Although the run with λ=5 made the first migration as early as superstep 750, the resulting load-balance settled at a level slightly above 0.2. A further migration occurring at superstep 1500 did not improve the load-balance significantly, but caused a slight drop in GVT rate instead.

Table 5.5 shows the execution times of the model executed using different values of λ. The execution time performance shows that despite the drop in GVT rate at the later part of the execution, the run with λ=5 has the best performance due to the its early migration compared to the runs with λ=10 and 20.

λ	5	10	20
Execution time	136.3	142.2	159.3

Table 5.5: Execution Times (sec.) for M_{1u} using Different Values of λ.

5.5 Balancing Communication WorkLoad

In the previous sections, we examined the PHold model that suffers only from computation imbalance. In this section, we turn our attention to a model that exhibits both computation as well as communication imbalance.

We again start with the PHold model with 1000 nodes. Two types of events are used in this experiment: *normal* event and *comm* event. A *normal* event is used in normal PHold operation, i.e. a node receiving a *normal* event increases the time-stamp of the event and forwards it to another node. The *comm* event is used to model events with high communication requirements and is only generated by one of the designated nodes. On receiving a *normal* event, these designated nodes forward the *normal* event as well as generate a *comm* event to another node. A *comm* event is discarded when received by a node. In this experiment, a *comm* event carries a large payload of an integer array of size 4000. The time-stamp increment for both *normal* and *comm* events are exponentially distributed with mean 1.0.

Two different approaches are used to partition the nodes onto processors. The first approach distributes the normal nodes and the nodes with high communication requirements uniformly among the processors. This model is referred to as M_{2b} and has balanced computation as well as communication workload. The second approach assigns all the nodes with high communication requirements to a certain processor and distributes the normal nodes among the other processors. This model is referred to as M_{2b}. Model M_{2b} has both load-imbalance in computation as well as communication workload.

Figure 5.9 shows the computation imbalance, WB, and the communication imbalance, CB, for the models M_{2b} and M_{2u} executed using BSP-TW and BSP-TW DLB$_c$. The communication imbalance, CB, will be defined in equation (5.13) in the next section. Figure 5.10 shows the corresponding GVT rates for the two models.

Although the BSP-TW DLB$_c$ algorithm is able to reduce computation load-imbalance significantly, it is not capable of handling communication imbalance. The communication imbalance stays at a relatively high level of 1.75 despite the computation workload being well-balanced at a level of 0.2. As shown in Figure 5.10, the GVT advance rate also suffers

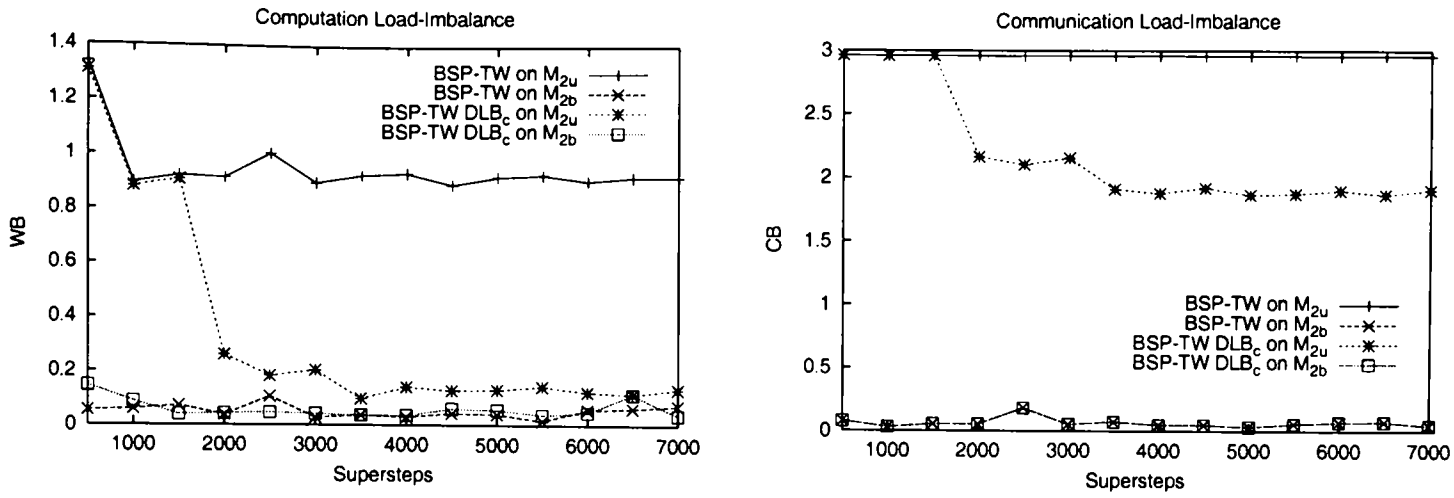


Figure 5.9: Computation and Communication Load-imbalance for M_{2b} and M_{2u} using BSP-TW and BSP-TW DLB$_c$.

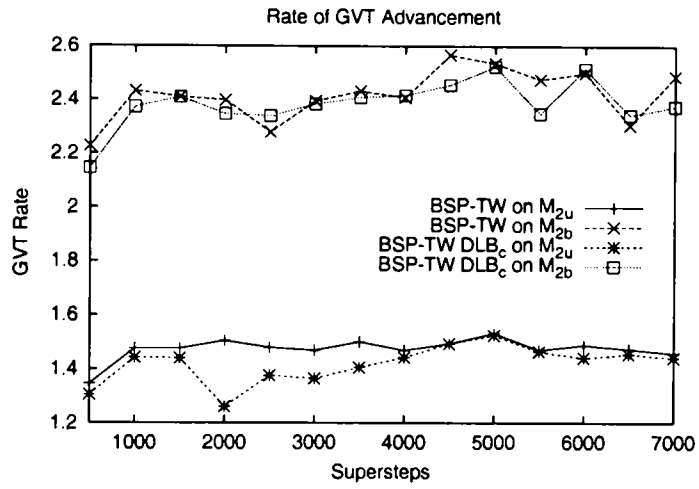


Figure 5.10: Rate of GVT Advancement for M_{2b} and M_{2u}.

due to the presence of high communication load-imbalance. Clearly for the load-balancing algorithm to be effective, the effect of communication imbalance needs to be taken into account in order to maximize performance.

5.5.1 Algorithm for Balancing Communication Workload

We now present a modified dynamic load-balancing algorithm for BSP-TW that allows the balancing of both computation as well as communication workload. We refer to this modified algorithm as BSP-TW DLB$_{cc}$.

Figure 5.11 shows the BSP-TW DLB$_{cc}$ algorithm with computation and communication load-balancing. The BSP-TW DLB$_{cc}$ algorithm is essentially the same as the load-balancing algorithm presented in Figure 5.1 with only computation load-balancing. Besides checking for computation load-imbalance WB, the algorithm now also checks for

```
bsp_begin();
[A] Initialization;
v'_old := 0;
while GVT < SimEndTime do
    [B] Receive external events and process rollback;
    [C] Compute new GVT, perform fossil collection and
        compute new event limit n_e every n_g supersteps;
    [D] After each λ GVT computation:
        compute new GVT rate ν'_new
        if (1.0 − φ)ν'_old ≤ ν'_new ≤ (1.0 + φ)ν'_old then
         flag := false;
         compute WB;
         compute CB;
         if WB > ε then flag := balance_computation(); endif
         if CB > ε and flag = false then balance_communication(); endif
        endif
        ν'_old = ν'_new;
    [E] Execute n_e events;
    bsp_sync();
endwhile
bsp_end();
```

Figure 5.11: BSP-TW DLB_{cc} Algorithm for Computation and Communication Load-balancing.

communication imbalance CB. In order to execute this algorithm, additional information needs to be shared between processors before each migration point. Suppose a migration point takes place in superstep s. At superstep $s-1$, besides sending the computation workload $P_i.wl$, each processor also sends another data $P_i.cl$. $P_i.cl$ is the communication workload of the processor since the last migration point. The communication imbalance CB is defined as:

$$CB = \frac{\max(P_i.cl) - \text{mean}(P_i.cl)}{\text{mean}(P_i.cl)}. \tag{5.13}$$

As with the case for detecting computation imbalance, the threshold value ϵ is used to detect communication imbalance. The BSP-TW DLB_{cc} algorithm first tries to balance computation workload. If no migration of object takes place in the computation balancing step as indicated by the variable *flag*, then the algorithm will try to balance communication workload.

In order to preserve the balanced computation workload achieved by the computation load-balancing step, the communication load-balancing algorithm cannot simply perform one-sided load transfer. Instead, load exchange must be performed between pairs of processors in order to reduce communication load-imbalance while preserving the computation load-balance. The amount of workload to exchange between processors in order to balance communication workload can be computed as follows.

Let P_{max} be the processor having the maximum communication workload and P_{min} be the processor having the minimum communication workload. Suppose the amount of computation workload to exchange in order to balance communication workload is x. The value of x can be computed using the following equation:

$$x = \frac{(P_{max}.cl - P_{min}.cl)(P_{max}.wl * P_{min}.wl)}{2((P_{max}.cl * P_{min}.wl) - (P_{min}.cl * P_{max}.wl))}. \quad (5.14)$$

Figure 5.12 shows the algorithm for balancing communication workload. After determining the amount of computation workload to be exchanged, the algorithm estimates the number of objects to be moved from each processor. Load exchange only takes place if at least one simulation object is migrated from each processor. The selection of simulation objects to exchange for communication load-balancing also uses the `computation_migrate()` procedure in Figure 5.3.

The algorithm terminates when CB falls below ϵ or when the estimated number of objects to be moved from either one of the processors is zero. As with the computation balancing algorithm, the variable *flag* is returned from the procedure to indicate if any migration occurs in the communication load-balancing algorithm. This variable *flag* will also be used by the lookahead optimization module in section 5.6.2.

Using the BSP-TW DLB_{cc} algorithm, the experiment is repeated on models M_{2b} and M_{2u}. Figure 5.13 shows the computation and communication imbalance for the simulation runs. The GVT rates for the runs are shown in Figure 5.14. For these runs, the parameters are fixed at ϕ=0.1, ϵ=0.2 and λ=10.

We see that BSP-TW DLB_{cc} does indeed successfully reduce both the computation and communication load-imbalance for the M_{2u} significantly. The improvement is reflected in the increase in GVT rate after superstep 3500. By superstep 6000, the communication load-imbalance has dropped below 0.2. At this point, the GVT rate for the unbalanced model M_{2u} is comparable to that of the balanced model M_{2b}.

balance_communication()
flag := false;
while $CB > \epsilon$ do
let P_{max} be the processor with the maximum communication workload;
let P_{min} be the processor with the minimum communication workload;
$x := \frac{(P_{max}.cl - P_{min}.cl) * (P_{max}.wl * P_{min}.wl)}{2 * ((P_{max}.cl * P_{min}.wl) - (P_{min}.cl * P_{max}.wl))}$;
if $x \geq P_{max}.wlpo$ **and** $x \geq P_{min}.wlpo$ **then**
computation_migrate(x, P_{max}, P_{min});
computation_migrate(x, P_{min}, P_{max});
$y := \frac{P_{max}.cl + P_{min}.cl}{2}$;
$P_{max}.cl := y$;
$P_{min}.cl := y$;
compute CB;
flag := true;
else
break;
endif
endwhile
return *flag*;

Figure 5.12: Algorithm for Determining Amount of Communication Workload to Migrate.

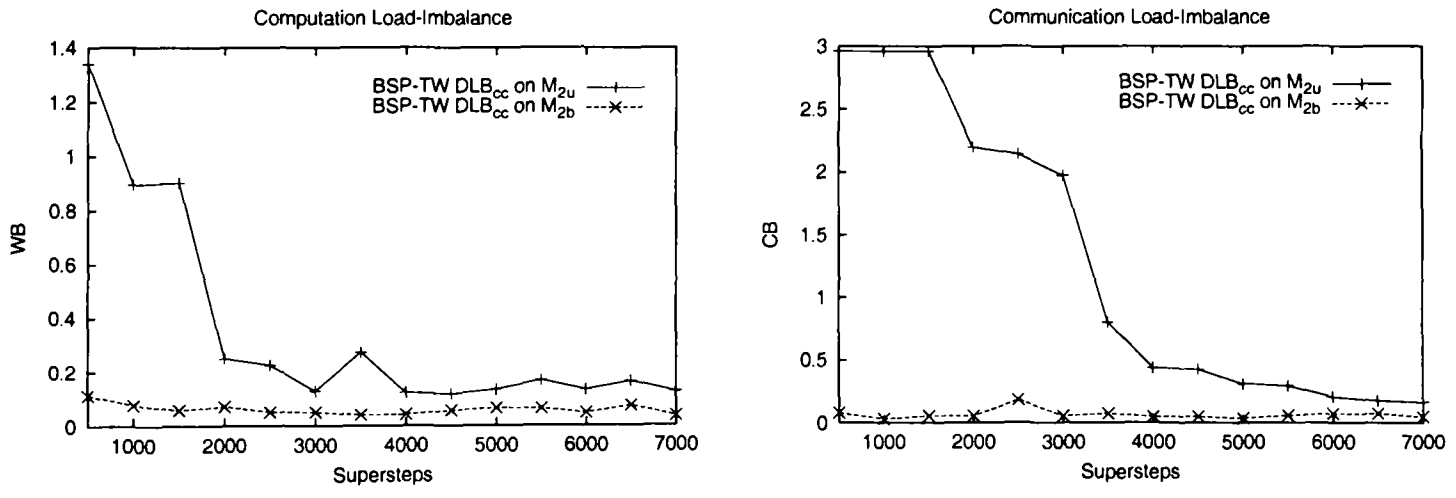


Figure 5.13: Computation and Communication Load-imbalance for M_{2b} and M_{2u} using BSP-TW DLB_{cc}.

Table 5.6 shows the execution time performance for the BSP-TW, BSP-TW DLB_c and BSP-TW DLB_{cc} algorithms on both the M_{2b} and M_{2u} models. We show results for BSP-TW DLB_c and BSP-TW DLB_{cc} using λ=10, 5 and 2. The small difference between BSP-TW and BSP-TW DLB_c on model M_{2b} shows that the overhead involved in performing just computation load-balancing alone is very small. The BSP-TW_{cc} algorithm has slightly

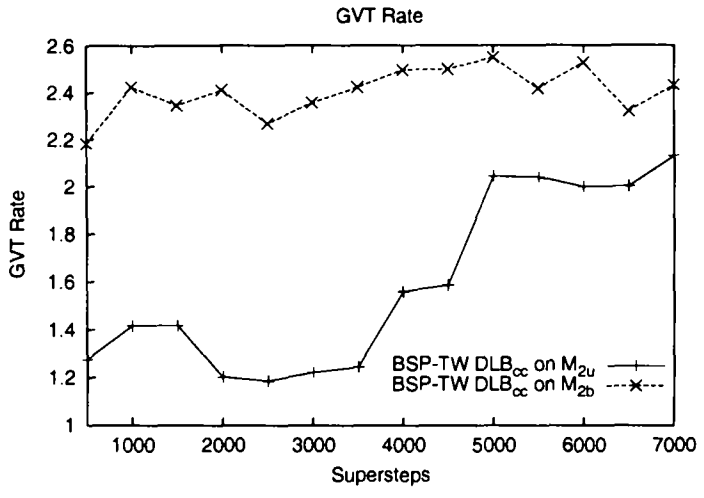


Figure 5.14: Rate of GVT Advancement for M_{2b} and M_{2u}.

	M_{2b}	M_{2u}
BSP-TW	725.2	1346.1
BSP-TW DLB$_c$ λ=10	714.4	1008.5
BSP-TW DLB$_c$ λ=5	719.1	951.0
BSP-TW DLB$_c$ λ=2	750.7	909.6
BSP-TW DLB$_{cc}$ λ=10	714.6	945.7
BSP-TW DLB$_{cc}$ λ=5	725.1	849.6
BSP-TW DLB$_{cc}$ λ=2	736.3	813.1

Table 5.6: Execution Times (sec.) for M_{2b} and M_{2u} using BSP-TW, BSP-TW DLB$_c$ and BSP-TW DLB$_{cc}$ with Different Values of λ.

higher overhead than the BSP-TW$_c$ algorithm. This overhead increases as λ is decreased.

For model M_{2u}, the BSP-TW DLB$_c$ algorithm is able to achieve slight improvement through computation load-balancing. The facilities for communication load-balancing allows the BSP-TW DLB$_{cc}$ algorithm to achieve further improvement in execution times. We also see that decreasing λ has a positive impact on the performance of both BSP-TW DLB$_c$ and BSP-TW DLB$_{cc}$. The runs using BSP-TW DLB$_{cc}$ with λ=2 improves the execution time of M_{2u} by 40% compared with the runs using the original BSP-TW.

5.6 Optimizing Lookaheads

The BSP-TW DLB$_{cc}$ algorithm developed so far has the ability to balance computation and communication workload. However, as mentioned at the start of this chapter, for a DLB algorithm to work well in a PDES protocol, we need to consider a third factor, that is the lookaheads of communication channels between processors. Under BSP-TW, the looka-

heads between processors directly affect the advancement of GVT between supersteps.

From the DLB equation (5.10) in section 5.3, we see that improving computation load-balance and communication load-balance has the effect of reducing the execution time per superstep, δt_b. However, the equation can only hold true if the condition $\delta \nu_b \geq \delta \nu_n$ also holds. If δt_b improves but results in a decrease in $\delta \nu_b$, the improvement in δt_b must be substantial enough to outweigh the drop in $\delta \nu_b$.

Another possibility is that the simulation model is already well-balanced in terms of computation workload and communication workload. To improve performance further, we can improve the lookaheads between processors to produce an increase in $\delta \nu_b$. The DLB equation (5.10) still holds provided any optimizations of lookaheads do not adversely affect δt_b.

5.6.1 Effects of Lookaheads on Manufacturing Model

In this section, we re-examine the manufacturing simulation model described in section 4.4.2. This model poses a serious problem to most conservative simulation protocols due to the presence of zero-lookahead links on the `fork` nodes and the `merge` nodes. A common solution is to partition these zero-lookahead links within a processor so that links between processors have strictly non-zero lookaheads.

While zero-lookahead is less of a problem for optimistic simulation protocol such as BSP-TW, some form of partitioning to eliminate some of the zero-lookahead links can bring about substantial performance gain. To demonstrate this, we carried out experiments using BSP-TW on the manufacturing model with different partitioning methods.

The first partitioning method assigns simulation objects in a round-robin fashion to the processors. This simulation model is referred to as M_{3u}. The second partitioning method assigns consecutive block of 25 simulation objects onto the same processors (the same partitioning was used in section 4.4.2). We refer to this simulation model as M_{3b}. Model M_{3u} is expected to perform worse than M_{3b} since the round-robin assignment on M_{3u} will result in many zero-lookahead links between processors.

Figure 5.15 shows the computation and communication load-imbalance for models M_{3b} and M_{3u} executed using the BSP-TW algorithm. Both models exhibit very low computation and communication load-imbalance. Table 5.7 shows the execution times and percentage of rolled-back events for the manufacturing models. The importance of having good lookahead configuration is evident from the four-fold increase in execution time as well as the

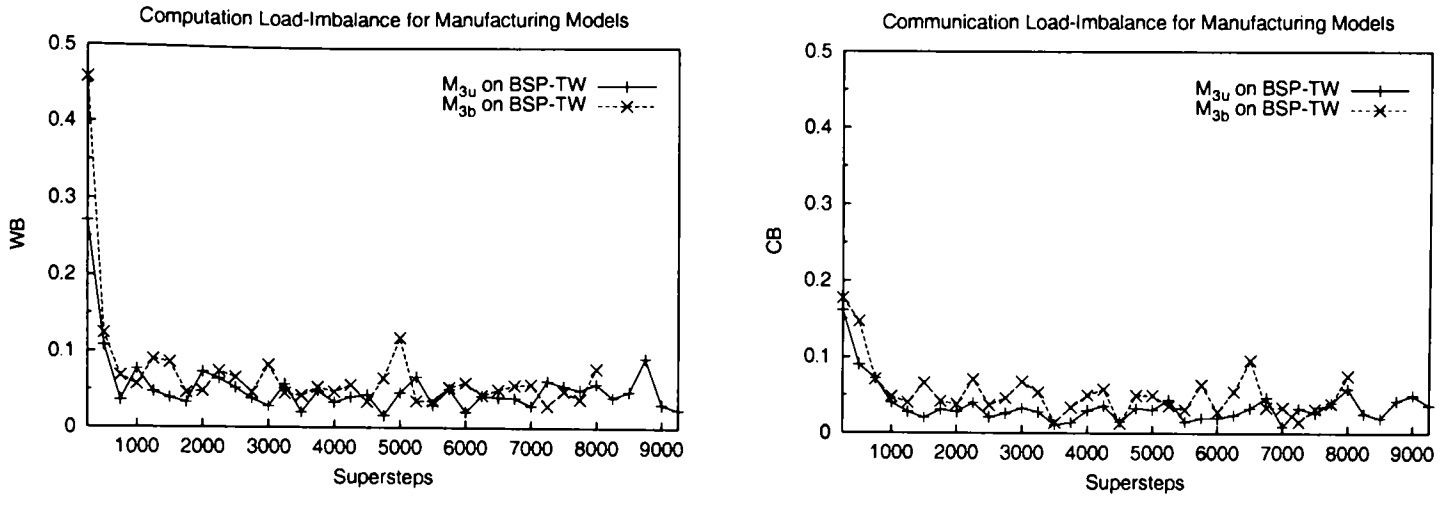


Figure 5.15: Computation and Communication Load-imbalance for M_{3u} and M_{3b}.

Model	M_{3b}	M_{3u}
Execution time	394.4	1730.0
Rollback %	2.1	26.8

Table 5.7: Execution Times (sec.) and Percentage of Events Rolled-back for Models M_{3b} and M_{3u} using Original BSP-TW.

high proportion of rolled-back events for model M_{3u} compared to M_{3b} when BSP-TW is used.

5.6.2 Improving Lookaheads on Manufacturing Model

It is clear that the load-balancing algorithm cannot look at balancing computation and communication workload alone, but must also take into consideration the task of improving lookaheads between processors. In this section, we extend the BSP-TW DLB_{cc} algorithm presented in section 5.5.1 to handle optimization of lookaheads.

The algorithm for the optimization of lookaheads is shown in Figure 5.16. The algorithm will be explained using the example in Figure 5.17. The example consists of three processors and six simulation objects. The lines represent the communication links and the number on each line represents the corresponding lookahead on that link. The terms used in procedure `optimize_lookahead()` are shown in Table 5.8.

Consider the lookahead configuration of simulation object S4. To determine the minimum internal lookaheads S4.la_{ii} and S4.la_{io}, all the internal links into S4 from other objects in processor P1 and out of S4 to other objects in P1 are considered. In this case, S4.la_{ii}=3.0, S4.ola_{ii}=S3, S4.la_{io}=0.5 and S4.ola_{io}=S3.

```
optimize_lookahead()
    mypid = bsp_pid();
    sort objects by number of rolled-back events in descending order;
    load2Move := 0;
    foreach local object o_i
        if o_i.la_ei <= o_i.la_eo then
            if o_i.la_ei < min(o_i.la_ii, o_i.la_io) then
                load2Move := load2Move + o_i.wl;
                if load2Move > η * P_pid.wl then
                    continue;
                endif
                destpid = processor(o_i.ola_ei);
                migrate(o_i, mypid, destpid);
        endif
    endfor
```

Figure 5.16: Algorithm for Optimizing Lookaheads. The function `processor(k)` returns the processor ID of which the object k is mapped to.

$o_i.wl$	:	computation workload of object i
$o_i.la_{ii}$	:	minimum internal input lookahead for object i
$o_i.la_{io}$	:	minimum internal output lookahead for object i
$o_i.la_{ei}$	:	minimum external input lookahead for object i
$o_i.la_{eo}$	:	minimum external output lookahead for object i
$o_i.ola_{ii}$	:	object id of minimum internal input lookahead for object i
$o_i.ola_{io}$	:	object id of minimum internal output lookahead for object i
$o_i.ola_{ei}$	:	object id of minimum input lookahead for object i
$o_i.ola_{eo}$	:	object id of minimum output lookahead for object i

Table 5.8: Terms used in Procedure `optimize_lookahead()`.

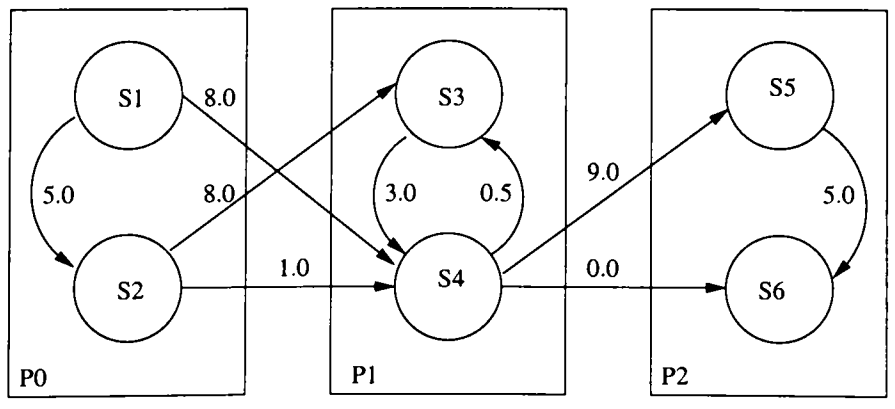


Figure 5.17: Configuration of Simulation Objects Before Optimization of Lookaheads.

To determine the minimum external lookaheads for S4, the algorithm considers all the incoming links from other objects outside processor P1 and all the outgoing links to other objects outside processor P1. In this case, S4.la_{ei}=1.0, S4.ola_{ei}=S2, S4.la_{eo}=0.0 and

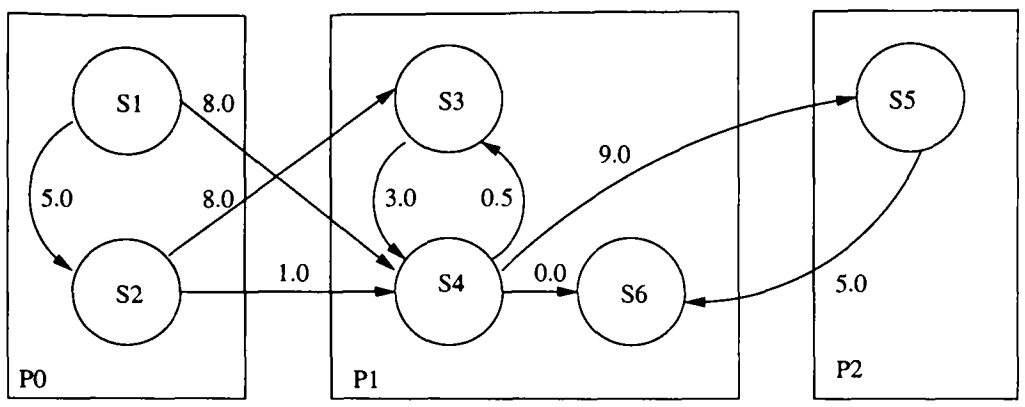


Figure 5.18: Configuration of Simulation Objects After Optimization of Lookaheads.

S4.ola_{eo}=S6.

The aim of the algorithm in Figure 5.16 is to group simulation objects with small lookaheads within the same processor. A simulation object o_i is migrated only when the external input lookahead $o_i.la_{ei}$ is smaller than both the external output lookahead $o_i.la_{eo}$ and the internal lookaheads $o_i.la_{ii}$ and $o_i.la_{io}$. In order to preserve the computation and communication load-balance achieved, a migration threshold parameter η is used to restrict the amount of workload that can be moved at each migration point. The algorithm also gives preference to those simulation objects with higher number of events rolled back over those with lower number of events rolled back. This is achieved through the sort operation at the start of the procedure.

Figure 5.18 shows the configuration of simulation objects after one round of lookahead optimization. Simulation object S6 has been moved from processor P2 to P1. This causes the zero-lookahead link from S4 to S6 to be hidden within processor P1. The new lookaheads for S4 are : $S4.la_{ei}$=1.0, $S4.la_{eo}$=9.0, $S4.la_{ii}$=3.0 and $S4.la_{io}$=0.0. Although simulation object S2 in processor P0 imposed a small lookahead of 1.0 to S4, the condition $S4.la_{ei} < S4.la_{io}$ is not satisfied. This prevented S4 from being moved to processor P0.

Figure 5.19 shows the modified BSP-TW algorithm that balances computation and communication workload, as well as optimizes lookaheads between processors. This algorithm will be referred to as BSP-TW DLB_{ccl}. The optimization of lookahead is only invoked if no computation balancing or communication balancing is taking place. The optimization would not be effective immediately after any computation or communication balancing as the movement of simulation objects in the balancing of computation and communication workload destroys the lookahead configuration of the system.

Unlike most conservative simulation engines where lookaheads between processors are explicitly specified, the BSP-TW DLB_{ccl} algorithm dynamically keeps track of lookaheads for both internal and external links. Also, lookahead values are only stored for those links

```
bsp_begin();
[A] Initialization;
ν'_old := 0;
while GVT < SimEndTime do
    [B] Receive external events and process rollback;
    [C] Compute new GVT, perform fossil collection and
        compute new event limit n_e every n_g supersteps;
    [D] After each λ GVT computation:
        compute new GVT rate ν'_new;
        if (1.0 − φ)ν'_old ≤ ν'_new ≤ (1.0 + φ)ν'_old then
         flag := false;
         compute WB;
         compute CB;
         [D1] if WB > ε then flag := balance_computation(); endif
         [D2] if CB > ε and flag = false then flag := balance_communication(); endif
         [D3] if flag = false then optimize_lookahead() endif
        endif
        ν'_old = ν'_new;
    [E] Execute n_e events;
    bsp_sync();
endwhile
bsp_end();
```

Figure 5.19: BSP-TW DLB_{ccl} Algorithm for Balancing Computation and Communication Workload and Optimizing Lookaheads.

that attained minimum average lookahead. This eliminates the need to maintain large tables of lookaheads between each simulation object. Figure 5.20 outlines the pseudo-code for updating the lookahead values for each simulation object whenever events are sent or received by a processor.

We repeated the experiments on models M_{3b} and M_{3u} using the new BSP-TW DLB_{ccl} algorithm. For this set of experiments, we used different values of η and set the other parameters as : ϕ=0.1, ϵ=0.2 and λ=5.

Table 5.9 shows the execution times for models M_{3b} and M_{3u} using BSP-TW and BSP-TW DLB_{ccl} algorithms with different values of η. Table 5.10 shows the corresponding percentage of events rolled-back. Figure 5.21 shows the GVT rate for both models.

If η is set at a very low value such as 0.05, only a small number of links with small lookaheads are merged together resulting in little performance improvement. Note that the execution time in this case (1157.5 sec) is still 33% better than that achieved by the original BSP-TW protocol.

For Initialization

foreach local object o_i **do**
$o_i.la_{ii} := \infty$; $o_i.ola_{ii} := -1$;
$o_i.la_{io} := \infty$; $o_i.ola_{io} := -1$;
$o_i.la_{ei} := \infty$; $o_i.ola_{ei} := -1$;
$o_i.la_{eo} := \infty$; $o_i.ola_{eo} := -1$;
endfor

For External Output LookAhead (the code for External Input LookAhead is symmetrical)

foreach external event e sent by o_i to o_j **do**
$la := e.rt$ - $e.st$;
if $o_i.ola_{eo} = o_j$ **then**
$o_i.la_{eo} := (o_i.la_{eo} + la)/2$;
else if $la < o_i.la_{eo}$ **then**
$o_i.la_{eo} := la$; $o_i.ola_{eo} := o_j$;
endif
endfor

For Internal LookAhead

foreach internal event e sent by o_i to o_j **do**
$la := e.rt$ - $e.st$;
if $o_i.ola_{io} = o_j$ **then**
$o_i.la_{io} := (o_i.la_{io} + la)/2$;
else if $la < o_i.la_{io}$ **then**
$o_i.la_{io} := la$; $o_i.ola_{io} := o_j$;
endif

if $o_j.ola_{ii} = o_i$ **then**
$o_j.la_{ii} := (o_j.la_{ii} + la)/2$;
else if $la < o_j.la_{ii}$ **then**
$o_j.la_{ii} := la$; $o_j.ola_{ii} := o_i$;
endif
endfor

Figure 5.20: Algorithm for Updating Lookahead. $e.st$ and $e.rt$ are the Send Time and Receive Time of Event e.

The performance of the simulation improves with increasing values of η. The effect of lookahead optimization is evident in the improvement in the percentage of rolled-back events as η is increased. In this experiment, setting η to the maximum value of 1.0 gives the best performance. However, it is not always desirable to set η to 1.0 since this allows links that are not critical to the performance of the simulation to be considered for optimization. The side effect of this is that a large number of simulation objects will be migrated resulting in degradation in load-balance. Figure 5.21 and Table 5.10 show that both the GVT rate as well as the rollback percentage for η=0.5 and 1.0 are quite similar. Setting η to 0.5 will thus

Model	M_{3b}	M_{3u}
BSP-TW	394.4	1730.0
BSP-TW DLB$_{ccl}$ η=0.05	383.1	1157.5
BSP-TW DLB$_{ccl}$ η=0.10	383.5	939.8
BSP-TW DLB$_{ccl}$ η=0.20	387.0	766.4
BSP-TW DLB$_{ccl}$ η=0.50	386.4	642.8
BSP-TW DLB$_{ccl}$ η=1.00	389.0	584.5

Table 5.9: Execution Times (sec.) for M_{3b} and M_{3u} using BSP-TW and BSP-TW DLB$_{ccl}$ with Different Values of η.

Model	M_{3b}	M_{3u}
BSP-TW	2.1	26.8
BSP-TW DLB$_{ccl}$ η=0.05	2.0	17.0
BSP-TW DLB$_{ccl}$ η=0.10	2.0	12.3
BSP-TW DLB$_{ccl}$ η=0.20	2.0	8.4
BSP-TW DLB$_{ccl}$ η=0.50	2.0	5.4
BSP-TW DLB$_{ccl}$ η=1.00	2.0	4.4

Table 5.10: Percentage of Events Rollback for M_{3b} and M_{3u} using BSP-TW and BSP-TW DLB$_{ccl}$ with Different Values of η.

allow majority of the links critical to the performance of the simulation to be considered for optimization while at the same time restricting the amount of workload that can be migrated.

5.7 Experiment with Arbitrary Flow Network Model

In this section, we examine the performance of the BSP-TW DLB$_{ccl}$ protocol using the Arbitrary Flow Network model described in section 4.4.3. For this set of experiments, the model consists of one source node, one sink node and 998 application nodes (only 8 application nodes are used in section 4.4.3). 30% of the application nodes are *slow* nodes while the rest are *fast* nodes.

The experiments are carried out with different probabilities (0.5, 0.6, 0.7 and 0.75) for a fast or slow node to communicate with the other group. When an application node is not communicating with another application node, it communicates with the sink node with probabilities ρ=0.3, 0.2, 0.1 and 0.05 respectively.

A total of 8 processors are used for this set of experiments. The simulation is run for

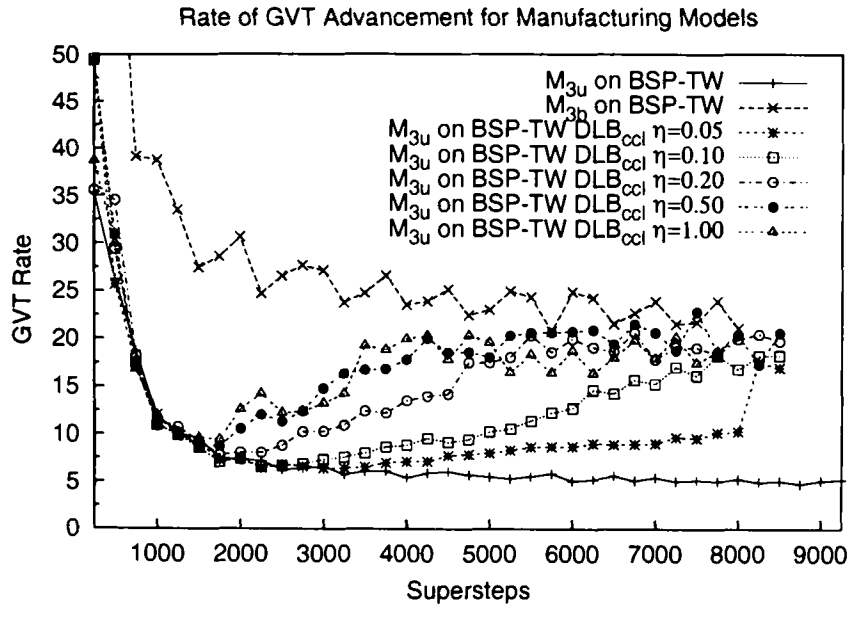


Figure 5.21: Rate of GVT Advancement for M_{3u} and M_{3b}.

ρ	0.05	0.1	0.2	0.3
BSP-TW	659.9	713.4	877.8	1587.1
BSP-TW DLB$_c$	657.7	606.7	532.4	919.3
BSP-TW DLB$_{cc}$	657.5	631.9	542.0	899.0
BSP-TW DLB$_{ccl}$	715.2	694.0	541.4	938.4

Table 5.11: Execution Times (sec.) for Arbitrary Flow Network Model using Different BSP-TW Algorithms.

20000 time unit. The source, sink and application nodes are partitioned onto the processors in a round-robin fashion. The event generation rates for the source node follows an exponential distribution with mean 1.0. We refer to this model as model M_{4u}. The parameters for this set of experiments are fixed at λ=5, ϕ=0.1, ϵ=0.2 and η=0.5.

Figure 5.22 shows the computation and communication load-imbalance of the arbitrary flow network model with different values of ρ executed using the original BSP-TW algorithm. The behaviour of the arbitrary flow network model changes as ρ is increased. Both the computation load-imbalance as well as the communication load-imbalance of the model increase as ρ is increased. The load-imbalance is a result of the processor in which the sink node resides having to receive and process more events.

Table 5.11 shows the execution times for the arbitrary flow network model executed using the different variations of BSP-TW. For ρ=0.2 and 0.3, the model suffers from severe load-imbalance. All three BSP-TW DLB algorithms are able to achieve better performance over BSP-TW by balancing computation and communication workload.

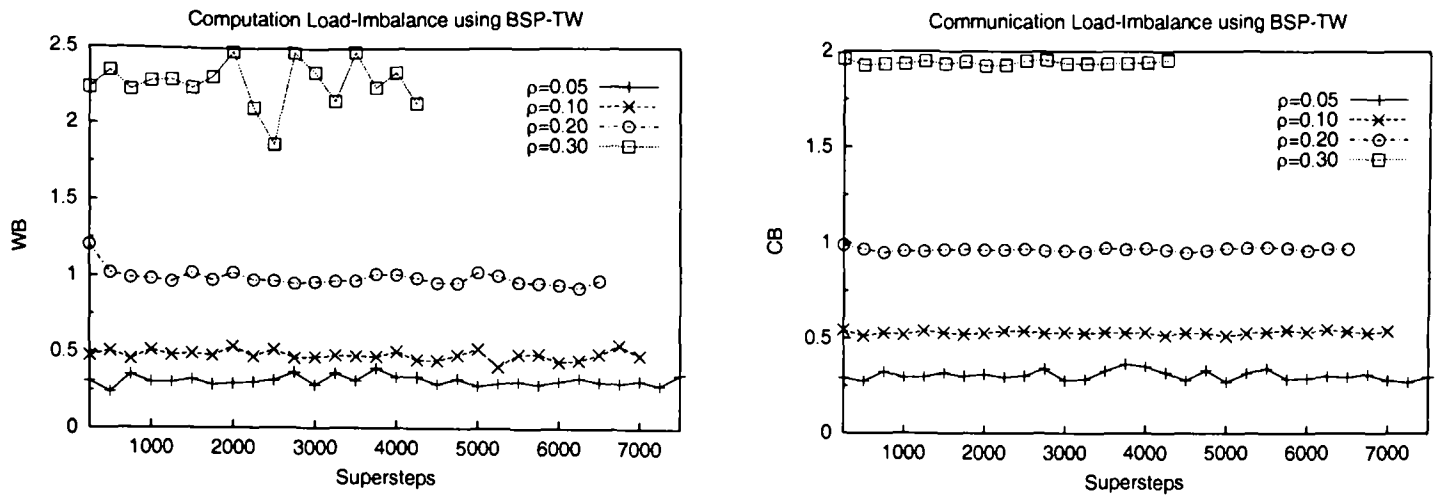


Figure 5.22: Computation and Communication Load-imbalance in the Arbitrary Flow Network Model using the Original BSP-TW.

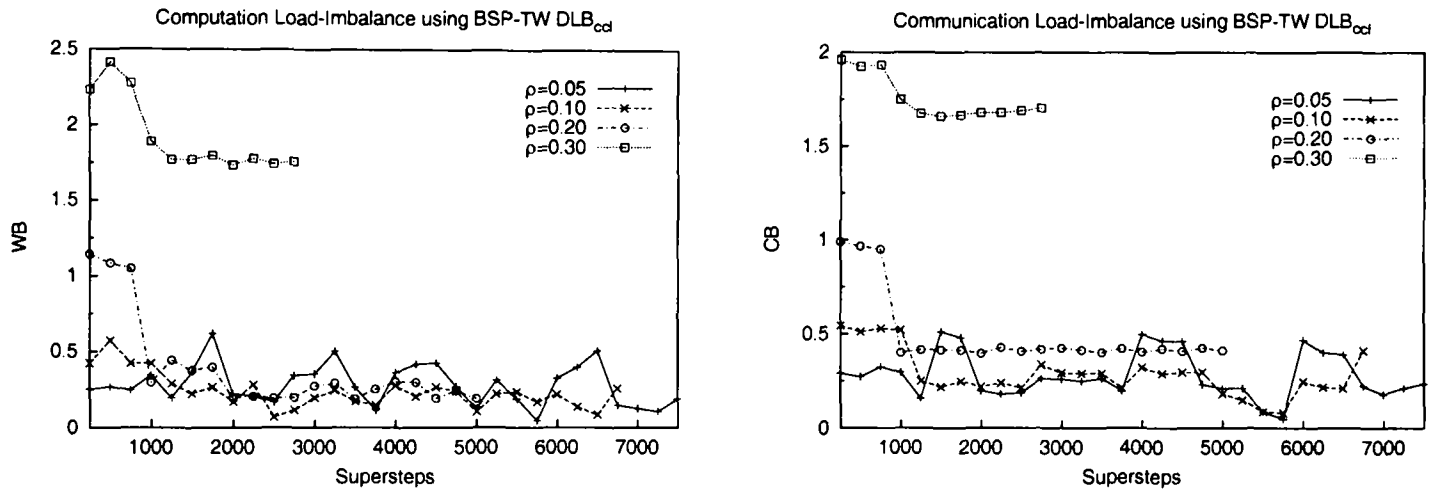


Figure 5.23: Computation and Communication Load-imbalance in the Arbitrary Flow Network Model using BSP-TW DLB$_{ccl}$.

Figure 5.23 shows the computation and communication load-imbalance for the arbitrary flow network model executed using the BSP-TW DLB$_{ccl}$ algorithm. The wide variations in computation and communication load-imbalance for BSP-TW DLB$_{ccl}$ on the model with ρ =0.05 is a result of the algorithm trying to carry out lookahead optimization. This explains the slight performance drop for ρ=0.05 and 0.1 in Table 5.11. The computation and communication load-imbalance of the model for the case ρ=0.3 is reduced considerably through the load-balancing carried out by the algorithms. However, the load-imbalance could not be improved further after all the other nodes are moved out of the processor in which the sink node resides.

5.8 Summary

In this chapter, we highlighted the different components of a simulation model that affect the performance of a simulation run. Our experiments show that the original BSP-TW protocol is not adequate in handling situations in which there are severe computation load-imbalance, communication load-imbalance and poor lookahead configuration.

We carried out successive refinements to the original BSP-TW algorithm to handle these situations. Our experimental results show that the new algorithm BSP-TW DLB_{ccl} is able to balance both computation and communication workload, as well as optimize lookaheads between processors. However, all these experiments are conducted on a dedicated system with no external workload interruption. In the next chapter, we examine the effects of external workload on the BSP-TW computation.

Chapter 6

Managing External Workload using BSP-TW

6.1 Overview

Most of the existing studies on parallel simulation are conducted on dedicated systems and many experiments even go to great length to ensure minimum external interference on the simulation runs. As a result, most existing parallel simulation protocols are designed with the assumption that uninterrupted system resources are available to execute simulation runs.

However, the widespread use of large scale computing cluster prompts us to rethink this approach to designing parallel simulation protocol. Very often computing resources in these clusters are not dedicated and are usually shared among multiple users. The load on each computing node in the clusters can fluctuate widely due to the presence of jobs from other users.

This scenario poses a great challenge for any parallel simulation protocol to run efficiently. While the BSP-TW DLB_{ccl} algorithm described in the previous chapter has facilities to dynamically balance both computation and communication, as well as optimize lookaheads between processors, the algorithm does not take into account the possibility of interruption from external workload. For example, Figure 6.1a shows the computation workload of five processors in a superstep. The shaded boxes show that an external workload is present in processor P0. Although all five processors have the same computation workload (represented by the white boxes, each white box can be considered the com-

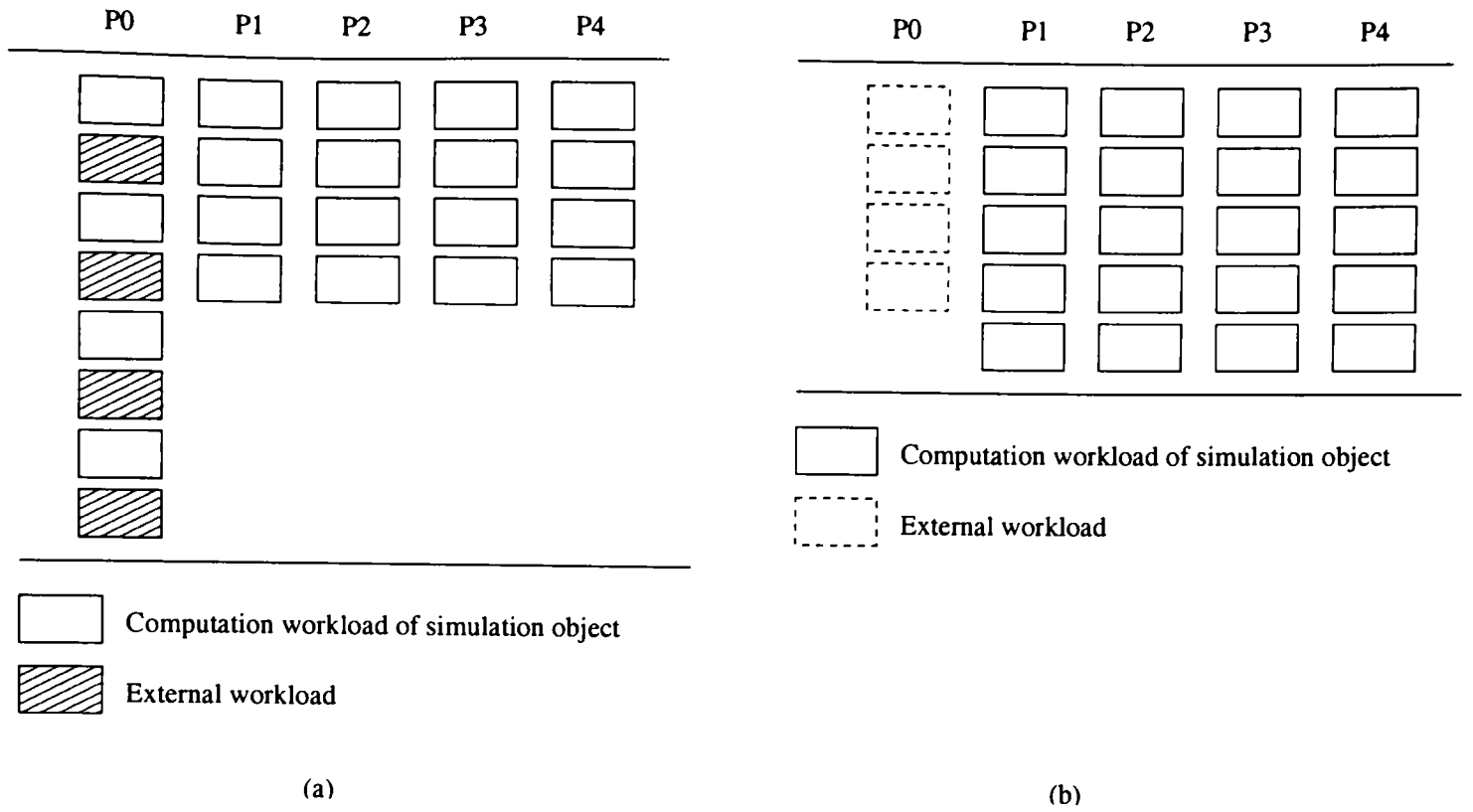


Figure 6.1: An Example of Interruption from External Workload.

putation workload of a simulation object), processor P0 takes twice as long to complete the supersteps since the CPU cycles are shared between the simulation workload and the external workload.

Figure 6.1b shows how BSP-TW DLB_{ccl} algorithm balances the computation workload in the superstep across all processors. All the simulation workload on processor P0 would be distributed to other processors. Note that in this case BSP-TW DLB_{ccl} carries out computation load-balancing based purely on the length of time each processor takes to complete the superstep. It has no knowledge of the presence of external workload on processor P0.

Since processor P0 is now without any computation workload, it can complete each subsequent superstep with minimum delay. However, without any knowledge of the presence of external workload on processor P0, the BSP-TW DLB_{ccl} algorithm will consider P0 to be idle and assume that there is an imbalance in computation workload. It will proceed to migrate simulation objects back into processor P0. The load configuration of the system again returns to that shown in Figure 6.1a. This results in a thrashing situation in which simulation objects are migrated in and out of the processor plagued by external workload.

In order to properly manage interruption from external workload upon a BSP-TW parallel simulation, the BSP-TW protocol needs to take into account the loss of processing cycles on those processors burdened with external workload. In this chapter, we introduce two different solutions to managing external workload under BSP-TW. The first approach

is described in section 6.2 and the proposed solution involves removing processors that are heavily loaded with external workload from the parallel simulation. The second approach, described in section 6.4, solves the problem by considering the available time-slice of the BSP processes on each processor in the parallel simulation.

6.2 Managing External Workload by Evicting Processors

In this section, we propose an extension to the BSP-TW DLB_{ccl} algorithm to allow external workload management. The new algorithm is referred to as BSP-TW DLB_{ccle}. The algorithm works by evicting simulation objects out of those processors loaded with external workload and marking these processors inactive.

6.2.1 BSP-TW DLB_{ccle} Algorithm

Figure 6.2 shows the pseudo-code for the new BSP-TW DLB_{ccle} algorithm. The pseudo code for `balance_extload()` is shown in Figure 6.3. The state variable $P_i.la$ is used to track the average system load of processor P_i. We classify the set of processors with average load greater than the processor load threshold parameter, θ, as heavily loaded. The average load of a processor is obtained by a UNIX system call `getloadavg()`. This system call returns the number of processes in the system run queue averaged over various periods of time. The 1 minute sample returned by the system call is used in the experiments.

At each migration point, the BSP-TW DLB_{ccle} algorithm attempts to evict all the simulation objects out of these heavily loaded processors. The method `migrate_all()` evicts all the simulation objects to other processors with normal workload in a round-robin fashion. A similar approach was used in [10] to handle external workload. The difference is that the simulation objects are first off-loaded to another processor before the loads are re-distributed. This is not suitable for BSP-TW since off-loading all the simulation objects from one processor to another will result in a sudden surge in computation workload on the destination processor in the following superstep.

The status of processor P_i is then set to inactive. As the dynamic load-balancing modules D1 to D3 only consider the set of active processors, simulation objects will not be migrated back to the processors that are still heavily loaded with external workload. When a previously heavily loaded processor's average system load drops below $\frac{\theta}{2}$, the status of the processor is reset to active. This causes the computation and communication load-

```
bsp_begin();
[A] Initialization;
ν'_old := 0;
while GVT < SimEndTime do
    [B] Receive external events and process rollback;
    [C] Compute new GVT, perform fossil collection and
        compute new event limit n_e every n_g supersteps;
    [D] After each λ GVT computation:
        compute new GVT rate ν'_new;
        if (1.0 − φ)ν'_old ≤ ν'_new ≤ (1.0 + φ)ν'_old then
         [D0] flag := balance_extLoad();
         compute WB;
         compute CB;
         [D1] if WB > ε and flag = false then flag := balance_computation(); endif
         [D2] if CB > ε and flag = false then flag := balance_communication(); endif
         [D3] if flag = false then optimize_lookahead(); endif
         endif
        endif
        ν'_old = ν'_new;
    [E] Execute n_e events;
    bsp_sync();
endwhile
bsp_end();
```

Figure 6.2: BSP-TW DLB_{ccle} Algorithm for Balancing Both Internal and External Workload, and Optimizing Lookaheads.

```
balance_extload()
    flag := false;
    mypid := bsp_pid();
    foreach processor P_i s.t. P_i.la > θ do
        flag := true;
        if mypid = i then
            migrate_all();
        endif
    endfor
    return flag;
```

Figure 6.3: Algorithm for Balancing External Workload.

balancing modules to detect the idle processor and allows simulation objects to be moved back to the processor.

6.3 Experiments using BSP-TW DLB$_{ccle}$ Protocol

In order to examine the effect of external workload on the different BSP-TW algorithms that have been proposed and to examine the efficiency of the BSP-TW DLB$_{ccle}$ algorithm, we carried out experiments using the manufacturing models described in section 5.6. The load-balancing parameters for the following sets of experiments are: λ=5, ϕ=0.1, ϵ=0.2, η=0.5 and θ=1.5.

Experiments are carried out using different BSP-TW algorithms on the same manufacturing models M_{3b} and M_{3u} described in section 5.6.1. Besides the two different partitioning strategies used, external workload are also introduced onto different number of processors. Two types of external workload are used: persistent and transient workload. The persistent external workload is introduced from the start of the simulation and lasts through the entire simulation duration. The transient external workload is introduced sometime after the simulation is started and lasts for a fixed duration.

6.3.1 Persistent External WorkLoad

For the first set of experiments, the manufacturing models M_{3b} and M_{3u} are executed using the original BSP-TW, BSP-TW DLB$_{ccl}$, and BSP-TW DLB$_{ccle}$ protocols. External workload are applied throughout the simulation duration on one of the processors. The number of external workload is varied from 0 to 2. Table 6.1 shows the execution times for models M_{3b} and M_{3u} executed using the different BSP-TW protocols under different number of external workload. The execution times for the runs without external workload are shown for comparison.

Table 6.1 shows that the performance of BSP-TW on both models deteriorate as external workload are introduced. While the performance for model M_{3b} is only affected by the presence of external workload, the performance for model M_{3u} is affected both by the presence of external workload as well as the poor lookahead configuration.

Figure 6.4 shows the GVT rates for the different algorithms on both models under different number of external workload for the first 1000 units of execution time. We see that both BSP-TW DLB$_{ccl}$ and BSP-TW DLB$_{ccle}$ greatly improve the GVT rate of the simulation.

For the case with only one external workload, the BSP-TW DLB$_{ccl}$ improves the GVT rate of model M_{3u} to the same level as that of the BSP-TW on model M_{3b}. This shows that

Ext. Load	Protocol	M_{3b}	M_{3u}
0	BSP-TW	394.4	1730.0
	BSP-TW DLB$_{ccl}$	386.4	642.8
	BSP-TW DLB$_{ccle}$	392.0	628.3
1	BSP-TW	646.6	2541.8
	BSP-TW DLB$_{ccl}$	572.0	1082.4
	BSP-TW DLB$_{ccle}$	495.1	818.7
2	BSP-TW	983.8	4102.2
	BSP-TW DLB$_{ccl}$	723.1	1034.0
	BSP-TW DLB$_{ccle}$	527.1	888.1

Table 6.1: Execution Times (sec.) for M_{3b} and M_{3u} using BSP-TW, BSP-TW DLB$_{ccl}$ and BSP-TW DLB$_{ccle}$ with Different Number of External Workload on Processor P0.

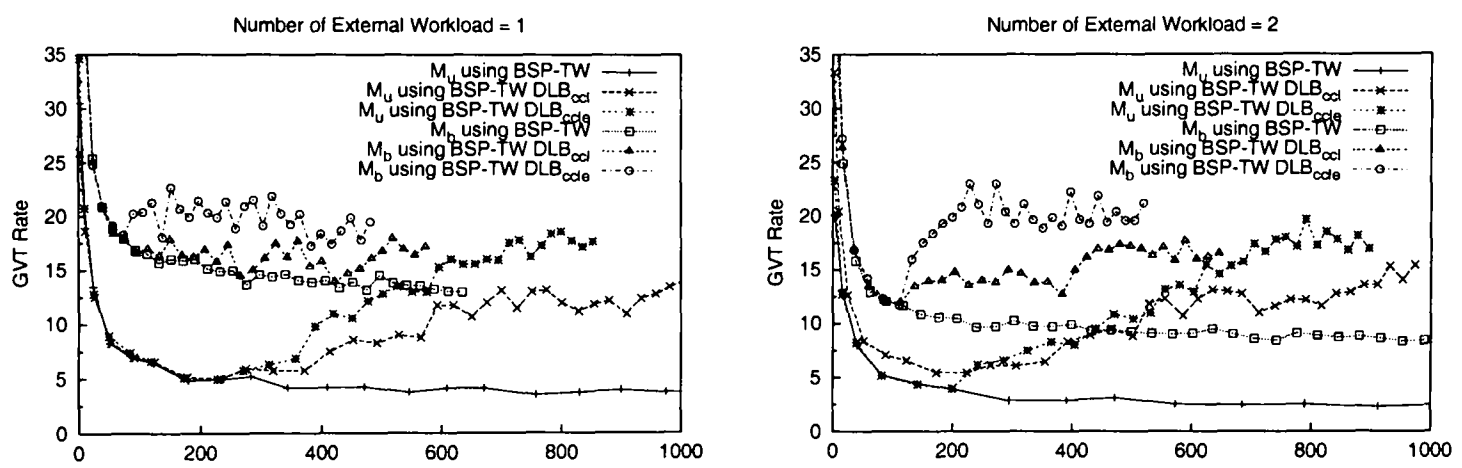


Figure 6.4: GVT Rates for M_{3b} and M_{3u} using BSP-TW, BSP-TW DLB$_{ccl}$ and BSP-TW DLB$_{ccle}$ Algorithms under Different Number of External Workload.

the improvement is attributed mainly to the lookahead optimization. On the other hand, the BSP-TW DLB$_{ccle}$ is able to both optimize lookahead as well as handle the presence of the external workload. This is evident from the much higher GVT rate for BSP-TW DLB$_{ccle}$ on model M_{3u} in Figure 6.4 compared to that for BSP-TW on model M_{3b}.

When the number of external workload is increased to two, the load-balancing capability of BSP-TW DLB$_{ccl}$ becomes apparent. This enables BSP-TW DLB$_{ccl}$ to achieve a much higher GVT rate on model M_{3u} compared to BSP-TW on model M_{3b}.

6.3.2 Transient External Workload

In the second set of experiments, we examine the effect of loading one of the processors in the simulation system with transient external workload to observe the behaviour of the

	Duration = 200 sec.				Duration = 600 sec.			
	No. of Ext. Workload				No. of Ext. Workload			
	1	2	3	4	1	2	3	4
BSP-TW	920.3	964.0	990.9	1014.1	1087.5	1209.3	1262.1	1311.0
BSP-TW DLB$_{ccl}$	929.8	1014.6	1048.5	1108.8	1065.4	1163.0	1215.8	1305.8
BSP-TW DLB$_{ccle}$	1063.9	1099.7	1104.5	1129.5	1067.4	1078.6	1112.8	1165.4

Table 6.2: Execution Times (sec.) for M_{3b} using BSP-TW, BSP-TW DLB$_{ccl}$ and BSP-TW DLB$_{ccle}$ with Different Number of External Transient Workload.

system in the presence of the external workload and after the workload has been removed. This set of experiments is performed only on model M_{3b} and the length of the simulation is extended from 10^4 time unit to 2×10^4 time unit. The external workload are applied 200 seconds after the simulation begins. We experimented with transient external workload that last for 200 seconds and 600 seconds respectively. The number of external workload on each processor is also varied from 1 to 4.

Table 6.2 shows the execution times for the model using BSP-TW, BSP-TW DLB$_{ccl}$ and BSP-TW DLB$_{ccle}$ algorithms. The performance of BSP-TW DLB$_{ccle}$ degrades compared to BSP-TW and BSP-TW DLB$_{ccl}$ for the set of experiments with transient workload of 200 seconds. The effect of external workload management are more evident when the duration of the transient external workload is increased to 600 seconds. The increased duration of the presence of the external workload makes the eviction of simulation objects by BSP-TW DLB$_{ccle}$ worthwhile and allows it to obtain better performance compared to the other two protocols.

Figures 6.5 and 6.6 show the GVT rates for the different algorithms under different number of external workload that last for 200 seconds and 600 seconds respectively. We see that the GVT rate for the BSP-TW algorithm drops proportionally to the number of external workload but returns to its original level once the external workload are removed.

The sharp drop in GVT rate for BSP-TW DLB$_{ccle}$ protocol in both set of graphs indicates the migration point at which simulation objects are moved back to the processor that was previously loaded with the transient workload. The re-population of simulation objects onto a processor requires a new event limit to be computed. This results in high event rollback rate during the first few GVT computation intervals immediately after the migration point. The movement of simulation objects into the processor also destroys the optimized lookahead configuration between processors. Both the high event rollback ratio and the ruined lookahead configuration contribute to the sharp drop in GVT rate.

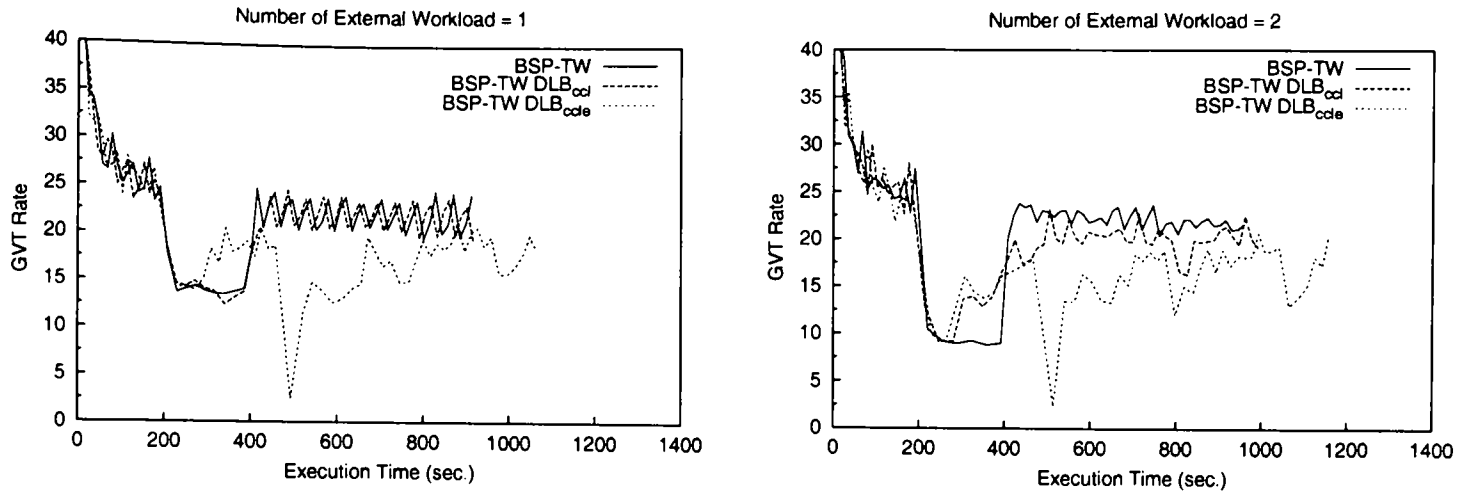


Figure 6.5: GVT Rates for M_{3b} using BSP-TW, BSP-TW DLB$_{ccl}$ and BSP-TW DLB$_{ccle}$ Algorithms under Different Number of External Transient Workload. Duration of Transient Workload = 200 seconds.

The graphs show that the BSP-TW DLB$_{ccle}$ algorithm is able to restore the GVT rate to its previous high level. However, a considerable length of time is required to accomplish this. For the case with a short transient workload of 200 seconds, it means that the benefits obtained by migrating simulation objects out of the loaded processor do not justify the cost of re-population of simulation objects and re-optimization of lookaheads once the external workload is removed.

6.3.3 External Workload on Multiple Processors

In the final set of experiments, we examine the effects of artificially applying external workload on more than one processor. This set of experiments is conducted using only model M_{3b} with two external workload applied on each of the designated processors throughout the duration of the simulation. The experiments are repeated by loading external workload on two processors followed by four processors.

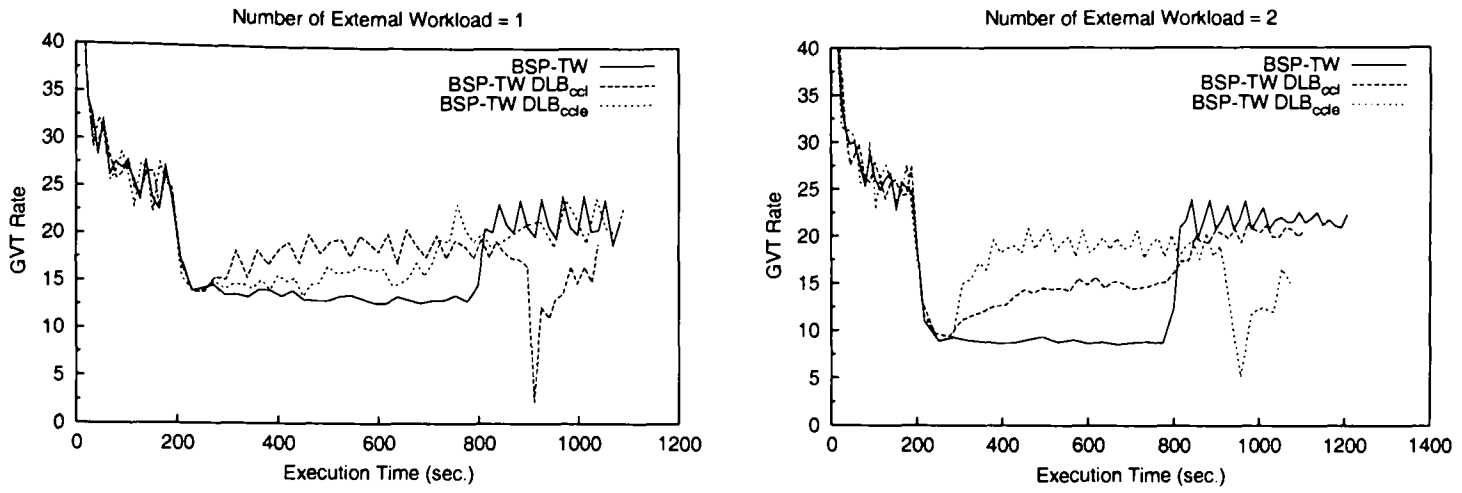


Figure 6.6: GVT Rates for M_{3b} using BSP-TW, BSP-TW DLB$_{ccl}$ and BSP-TW DLB$_{ccle}$ Algorithms under Different Number of External Transient Workload. Duration of Transient Workload = 600 seconds.

Table 6.3 shows the execution times for the different algorithms on model M_{3b} with different number of processors being loaded. The timings for the cases with zero and one processor loaded are included for comparison.

We first note that the performance of the original BSP-TW algorithm is not affected by the number of processors loaded with external workload. This is expected since the BSP cost model predicts that the performance of the BSP-TW protocol is affected by the processor that is the most loaded rather than the number of processors that are loaded.

The performance of BSP-TW DLB$_{ccl}$ deteriorates as more processors are loaded. While the load-balancing modules try to balance computation workload by migrating simulation objects from heavily loaded processor to less loaded ones, the effects of external workload are not taken into account. The result is that simulation objects are migrated to and fro between heavily loaded and less loaded processors.

Figure 6.7 shows the accumulated workload of each processor between migration points for the experiment with four processors loaded with external workload. External workload

	No. of Processor Loaded			
Protocol	0	1	2	4
BSP-TW	394.4	983.8	1002.4	1025.1
BSP-TW DLB_{ccl}	386.4	723.1	1050.3	1357.3
BSP-TW DLB_{ccle}	392.0	527.1	609.6	949.3

Table 6.3: Execution Times (sec.) for M_{3b} using BSP-TW, BSP-TW DLB_{ccl} and BSP-TW DLB_{ccle} with Different Number of Processors Loaded.

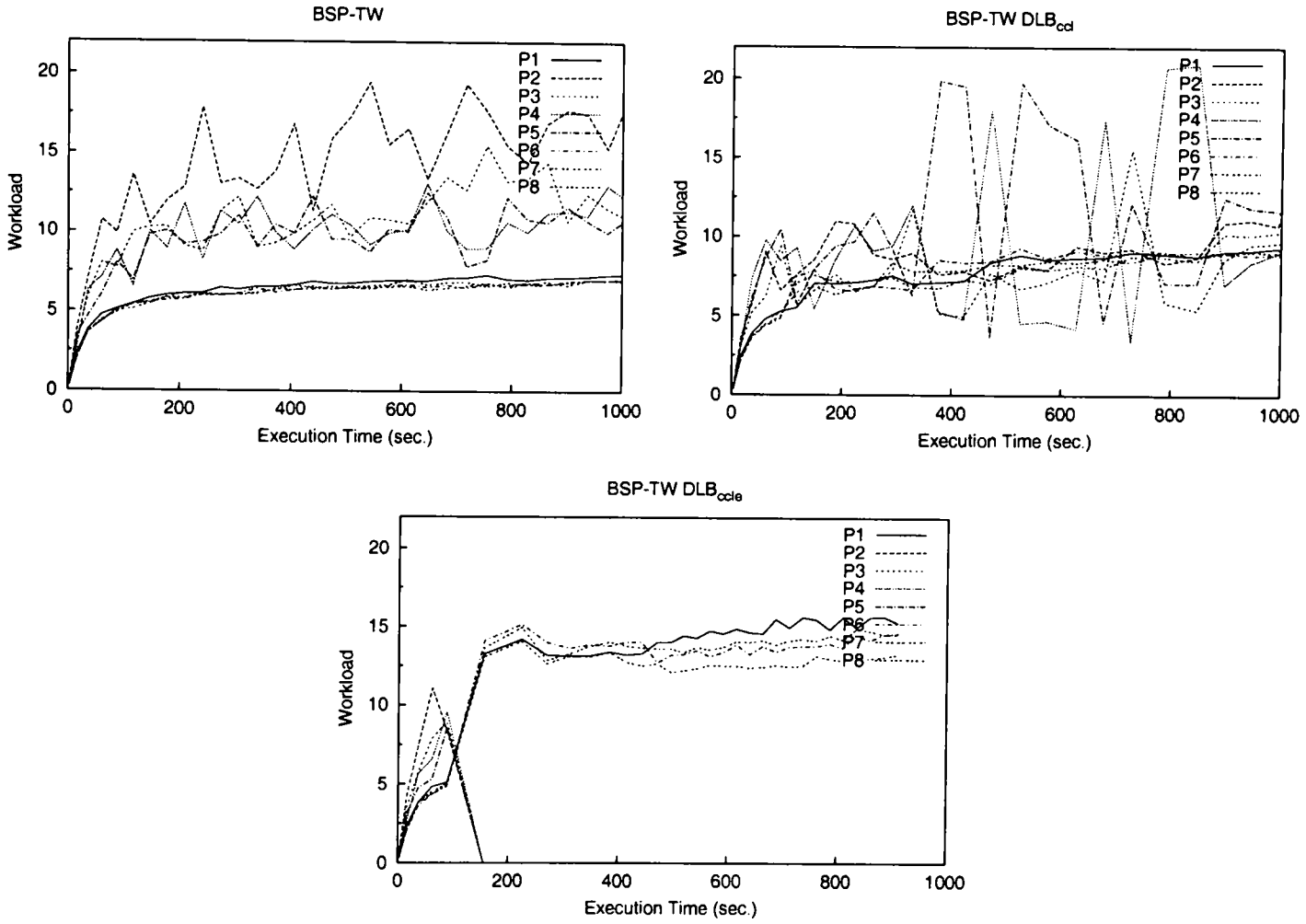


Figure 6.7: Workload of Individual Processor with Persistent Workload on Four Processors.

is applied to processors P2, P3, P4 and P5. The thrashing behaviour of BSP-TW DLB_{ccl} is evident from the workload of processors P4 and P5 from time 300 to time 800.

The BSP-TW DLB_{ccle} does not suffer from the thrashing behaviour observed in the BSP-TW DLB_{ccl} algorithm since all heavily loaded processors are not considered for load-balancing purpose. However, as more processors are excluded for computation, the individual workload of the remaining active processors increases and the performance of BSP-TW DLB_{ccle} approaches that obtained using the original BSP-TW.

6.4 Managing External Workload by Time Slicing

Although the BSP-TW DLB_{ccle} protocol described in the previous section does solve the problem of external workload interruption, it sacrifices the complete use of a processor whenever it is loaded with external workload, regardless of the amount of external workload in the processor. Also, the performance of BSP-TW DLB_{ccle} depends largely on how θ is set. If the value of θ is set too low, many processors may be evicted due to the presence of very small external workload. If the value of θ is set too high, the BSP-TW DLB_{ccle} may not react effectively to the presence of external workload.

In this section, we consider another approach to managing external workload by considering the available time slice for the BSP process on the heavily loaded processors, rather than leaving the processors completely out of the parallel computation.

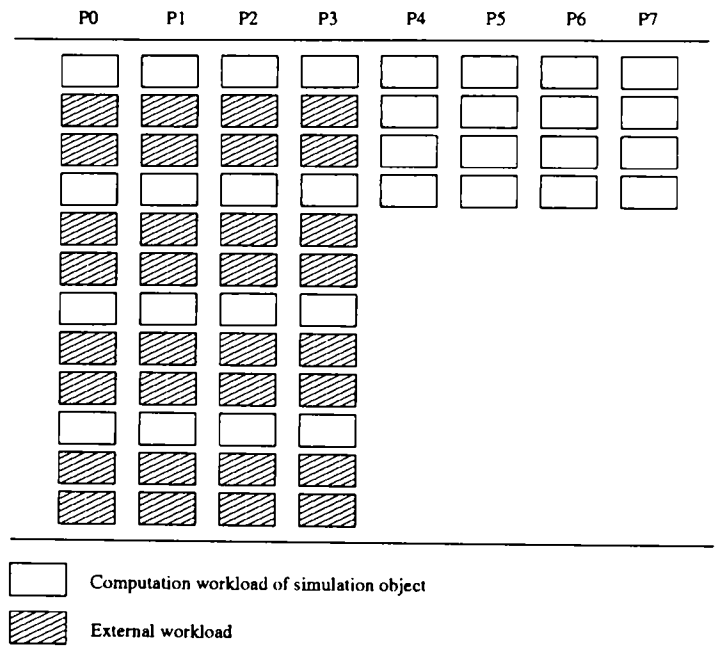
6.4.1 Example of External Workload Management using Time Slicing

We first illustrate our approach using the example shown in Figure 6.8. The figure shows the computation workload of a superstep for eight processors. Processors P0 to P3 are each loaded with two external workloads, indicated by the shaded boxes. The computation workload of simulation objects on all eight processors in the superstep are the same, as shown by the white boxes. Each white box can be considered the computation workload of a simulation object.

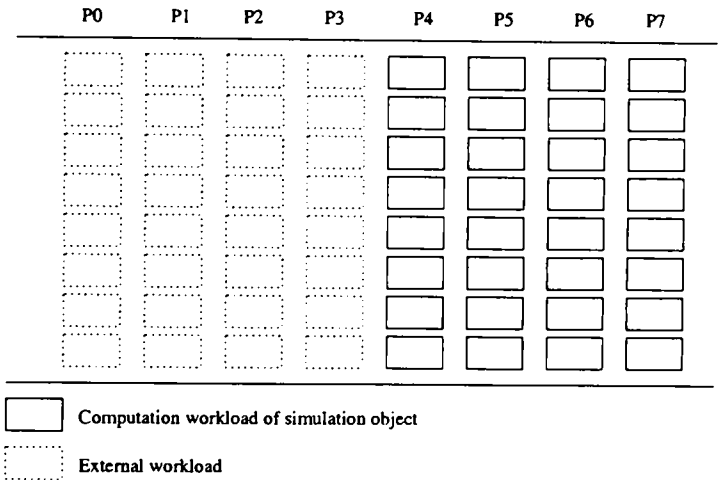
Due to the presence of external workload, the superstep on processors P0 to P3 takes three times the amount of time to process, as compared to those on processors P4 to P7. We can also say that the simulation workload is only given one-third slice of the CPU processing time. If we assume that each box (white or shaded) consumes one unit of CPU processing time, the superstep takes 12 units of CPU processing time.

Figure 6.8b shows workload configuration using the BSP-TW DLB_{ccle} algorithm. All the simulation objects are evicted from the four loaded processors and distributed to processors P4 to P7. The resulting configuration is such that processors P0 to P3 each can complete the superstep with minimum delay while processors P4 to P7 now have twice the amount of workload to process. The CPU processing time for this superstep is reduced to 8 time unit.

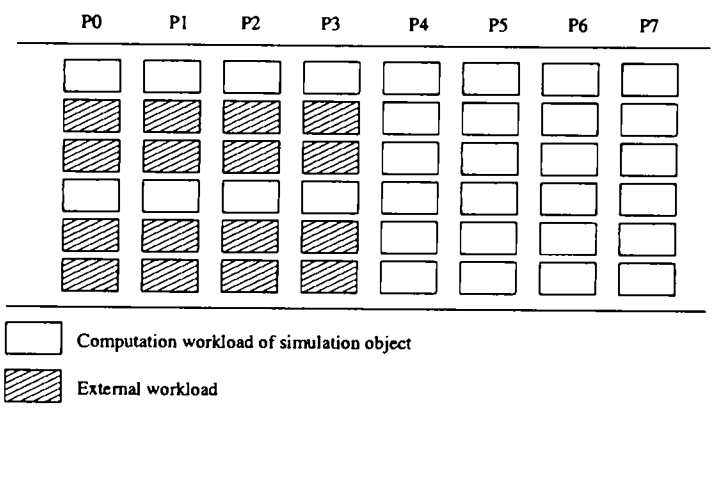
Another approach to managing the workload is to consider the fact that the BSP workload on the heavily loaded processors still have access to one-third slice of the CPU pro-



(a)



(b)



(c)

Figure 6.8: An Example of External Workload Management using Time Slicing.

cessing time. We can migrate parts of the simulation objects out of these processors so that the overall workload for all processors (taking into account the external workload) after the migration is still balanced.

Figure 6.8c shows an example for how this is done. Two simulation objects are migrated out of each processor loaded with external workload. The resulting workload configuration is balanced across all processors. The superstep now requires only 6 units of CPU processing time.

The reason for the improvement over that using BSP-TW DLB_{ccle} is due to the use of the remaining one-third slice of CPU processing time on those heavily loaded processors to process part of the simulation objects' workload. The increased in workload of those processors not affected by external workload is reduced compared to that using BSP-TW

DLB_{ccle}.

6.4.2 BSP-TW DLB_{ccls} Algorithm

We can now describe the BSP-TW DLB_{ccls} algorithm that provides an alternative solution to managing external workload by considering the allocated CPU time slice for the computation workload in each processor in the system. The outline of the BSP-TW DLB_{ccls} algorithm is essentially the same as the BSP-TW DLB_{ccl} algorithm shown in Figure 5.19. Unlike the BSP-TW DLB_{ccle} algorithm which adds another module to the BSP-TW algorithm, the BSP-TW DLB_{ccls} algorithm works together with the computation load-balancing module. Also, it should be noted that the BSP-TW DLB_{ccls} algorithm does not require the use of the processor load threshold parameter θ.

Before we describe the modification to the module for balancing computation workload, we first need to resolve the condition for detecting imbalance in computation workload. For example, we would want to consider the workload configuration in Figure 6.8a as unbalanced while the configuration in Figure 6.8c as well-balanced. As the time taken to complete a superstep in each processor is computed by summing up the time taken for executing each event in the superstep, the BSP-TW DLB_{ccl} algorithm will instead consider the configuration in Figure 6.8a as well-balanced while the configuration in Figure 6.8c as unbalanced. This conclusion is a direct result of not considering the external workload factor and is what leads to the presence of thrashing behaviour observed in section 6.3.3.

To resolve this problem, we can make use of the additional knowledge of the average system load of each processor ($P_i.la$) to work out a better approximation of the workload on each processor. We first scale the computation workload of each processor ($P_i.wl$) by its corresponding average system load as follows:

$$P_i.wl := P_i.wl * P_i.la\ . \tag{6.1}$$

Note that for those processors with average system load less than 1.0, $P_i.la$ will be set to 1.0.

The calculation of the computation imbalance, WB, of the system remains the same using the formula shown in equation (5.11). Using this formula, the load imbalance for the superstep shown in Figure 6.8a will be 0.5 while the superstep in Figure 6.8c will be treated as having perfectly balanced workload.

```
balance_computation()
    flag := false;
    while WB > ε do
        let P_max be the processor with the maximum computation workload;
        let P_min be the processor that yield the minimum average workload when paired with P_max
        x := (P_max.wl * P_max.la − P_min.wl * P_min.la) / (P_max.la + P_min.la)
        if x ≥ P_max.wlpo then
            flag := true;
            computation_migrate(x, P_max, P_min);
        else
            break;
        endif
        P_max.wl := P_max.wl - x;
        P_min.wl := P_max.wl
        compute WB;
    endwhile
    return flag;
```

Figure 6.9: Algorithm for Determining Amount of Computation Workload to Migrate taking into account System Workload.

The BSP-TW DLB_{ccls} algorithm has exactly the same structure as that of BSP-TW DLB_{ccl} shown in Figure 5.19. The difference lies in the `balance_computation()` module, which now needs to take into consideration the system load of those processors involved in the load transfer process.

Figure 6.9 shows the optimized pseudo-code for the `balance_computation()` module. Note that the computation workload used in the module have all been scaled by the average system load of individual processors. The processor P_{min} is not taken to be the one with the lowest computation workload, but rather the processor that will yield the lowest average workload when selected to engage in the load transfer process with the processor P_{max} that has the heaviest computation workload.

Figure 6.10 shows an example to illustrate why the processor that has the lowest computation workload is not chosen to be processor P_{min}. The example shows three processors and their respective computation workload for a superstep. We see that processor P0 is loaded with one external workload and processor P1 is loaded with four external workload. Although processor P2 is free from any external workload, its overall computation workload is still higher than processor P1.

Suppose processor P1 is now chosen to engage in the load transfer process with proces-

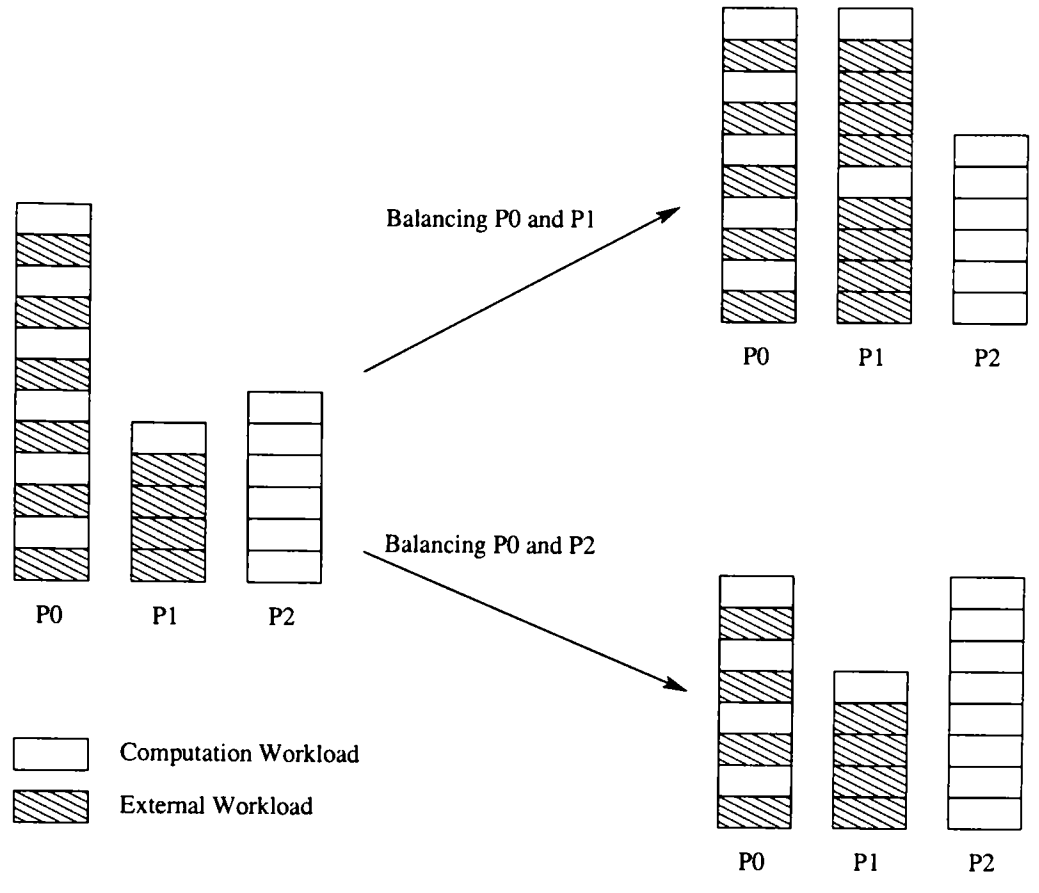


Figure 6.10: An Example to Illustrate the Selection of Processor P_{min}.

sor P0. One unit of computation workload will be migrated from processor P0 to processor P1. This results in a net decrease of two units of computation workload in processor P0 and a corresponding five-unit increase in computation workload in processor P1. The overall improvement in the maximum computation workload is two units.

However, if processor P2 is chosen instead, two units of computation workload will be migrated from processor P0 to processor P1. This results in a net decrease of four units in computation workload in processor P0 and a corresponding two-unit increase in computation workload in processor P2. The overall improvement in the maximum computation workload in this case is four units. Although processor P2 is not the processor having the lowest computation workload, selecting it for the balancing process yields better performance compared to selecting processor P1, which has the lowest computation workload.

The presence of external workload on the individual processor and the scaling of computation workload for each processor requires some modifications to the formula used to compute the amount of workload to be transferred from processor P_{max} to P_{min}. The following formula computes the amount of computation workload, x, that needs to be migrated from processor P_{max} to processor P_{min} so that the resulting workload, y, on both processors after the migration is equal.

$$x = \frac{P_{max}.la(P_{max}.wl - P_{min}.wl)}{P_{max}.la + P_{min}.la} \quad (6.2)$$

$$y = P_{max}.wl - x \quad (6.3)$$

The balancing process continues until the computation load-imbalance drops below ϵ, or when the amount of workload to be transferred is less than the average computation workload per processor.

6.5 Experiments using BSP-TW DLB$_{ccls}$ Protocol

In this section, we describe a set of experiments to compare the performance of BSP-TW DLB$_{ccls}$ algorithm with the original BSP-TW, as well as the BSP-TW DLB$_{ccle}$ algorithm. The experimental configurations are the same as those used in the previous section. A total of eight workstations are used to execute the simulation model M_{3b}. The experiments are carried out by loading different number (1, 2, 4 and 6) of processors with different number (1, 2, 3, 4) of external workload.

Table 6.4 shows the execution times using the three different protocols on model M_{3b}. The column under BSP-TW DLB$^{*}_{ccls}$ is executed using a modified version of BSP-TW DLB$_{ccls}$. This version uses a modified average system load for each processor, which is shown below:

$$P_i.la := (P_i.la)^2. \quad (6.4)$$

The modified system load of individual processor is then applied to the scaling of the computation workload in equation (6.1). This modification has no effect on those processors with no external workload since $P_i.la$ will still be equal to 1.0. For those processors with average system load greater than 1.0, this change has the effect of encouraging the BSP-TW DLB$_{ccls}$ to migrate more simulation objects out of them. Similarly, it also discourages the load-balancing algorithm from migrating simulation objects back into them.

Table 6.4 shows that by not discarding completely those processors with heavy system load, the BSP-TW DLB$_{ccls}$ protocol is able to achieve better performance than the BSP-TW DLB$_{ccle}$ protocol for the case of six processors loaded with external workload. The performance of BSP-TW DLB$_{ccls}$ drops rapidly when the number of external workload on

		BSP-TW			
K	N	DLB_{ccl}	DLB_{ccle}	DLB_{ccls}	DLB^{*}_{ccls}
1	1	602.0	489.4	551.2	520.6
	2	789.8	517.8	559.3	558.0
	3	844.3	566.1	617.9	569.6
	4	993.3	592.2	787.8	606.1
2	1	718.9	596.8	672.9	613.5
	2	1170.5	634.1	705.1	615.1
	3	1318.3	635.9	732.7	651.2
	4	1667.6	693.5	958.8	690.7
4	1	826.2	944.0	768.6	747.8
	2	1502.5	954.7	981.9	749.0
	3	1697.1	987.7	1055.7	767.6
	4	2329.2	1021.6	1275.6	840.5
6	1	918.9	1462.8	934.0	867.8
	2	1704.6	1736.5	1350.0	1038.4
	3	2090.9	1843.2	1449.8	1124.8
	4	2971.3	1870.6	1816.9	1128.7

Table 6.4: Execution Times (sec.) for M_{3b} using BSP-TW DLB_{ccls} with N Number of Processors Loaded with K Number of External Workload.

the six processors is increased from one to four. Also, as the number of heavily loaded processors is reduced, the performance of BSP-TW DLB_{ccls} drops below that of BSP-TW DLB_{ccle}.

This drop in performance in BSP-TW DLB_{ccls} can be attributed to two factors: 1) insufficient simulation objects are migrated out of those heavily loaded processors as the number of external workload on these processors is increased; 2) side effects from the lookahead optimization module.

In order to verify the first hypothesis, we carried out the runs with BSP-TW DLB^{*}_{ccls} to test if better performance can be achieved by encouraging more simulation objects to be migrated out of the heavily loaded processors. In a way, the squaring of the average system load of individual processor in equation (6.4) serves to exaggerate the load situation of those heavily loaded processors such that more simulation objects can be migrated out of them.

Table 6.4 shows that this approach does significantly improve the performance of the BSP-TW DLB_{ccls} algorithm. For the case with four and six processors loaded with external workload, the performance of BSP-TW DLB^{*}_{ccls} drops gradually with increasing external workload. This shows that the dismal performance of BSP-TW DLB_{ccls} is indeed due to

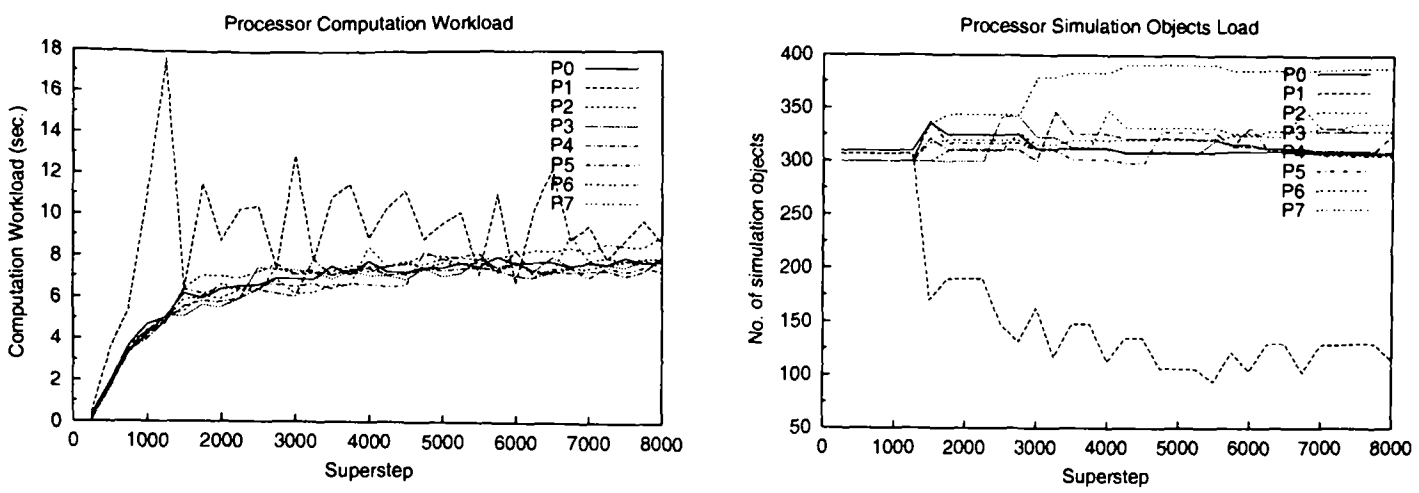


Figure 6.11: Breakdown of Individual Processors Computation Workload and Number of Simulation Objects using BSP-TW DLB^{*}_{ccls}.

insufficient simulation objects being migrated out of those heavily loaded processors.

However, for the runs with only one processor being loaded with external workload, the performance by either BSP-TW DLB_{ccls} or BSP-TW DLB^{*}_{ccls} is still slightly worse than that using BSP-TW DLB_{ccle}. This performance drop can be attributed to the side effect of lookahead optimization.

Figure 6.11 shows a breakdown of the computation workload as well as the number of simulation objects on each processor for a run executed using BSP-TW DLB^{*}_{ccls}. In this run, processor P1 is loaded with one external workload. We see that at superstep 1250, the balancing module is activated and the number of simulation objects on processor P1 drops from 307 to 170. Correspondingly, the computation workload for processor P1 decreases from a high level of 17.5 to 6.4.

However, at superstep 1500, an optimization of lookahead is carried out by the BSP-TW DLB^{*}_{ccls} algorithm. This resulted in 20 simulation objects being migrated back into processor P1. The computation workload of processor P1 is increased to 11.5 in superstep 1750. At this point, the pattern repeats itself with the computation balancing module migrating simulation objects out of processor P1 and the lookahead optimization module migrating simulation objects back into processor P1. This problem has been previously mentioned in section 5.6.2. The main problem here is that the migration threshold η=0.5 allows up to 50% of the simulation objects to be migrated during each round of lookahead optimization and this tends to disrupt the load-balance achieved in the computation balancing process.

A possible solution to resolving this issue might be to use a value of η smaller than 0.5. While this will reduce the amount of simulation workload that can be migrated back into the heavily loaded processors, it will also slow down the lookahead optimization process

on other processors, causing the performance to drop. An effective solution requires the value of η to be set differently for different processors with different load configurations. Further work will need to be carried out to explore this possibility.

6.6 Summary

In this chapter, we explained how the BSP-TW DLB_{ccl} algorithm is inadequate in handling situations in which some processors in the system are interrupted by external workload. Two different solutions are proposed in order for the BSP-TW algorithm to properly handle interruption from external workload.

In our first solution to this problem, we proposed the method of evicting simulation objects from a processor whenever its system load exceeds a pre-defined threshold parameter θ. This technique was implemented in the BSP-TW DLB_{ccle} protocol. Experimental results show that the BSP-TW DLB_{ccle} protocol does give performance improvement over the BSP-TW DLB_{ccl} protocol that does not consider external workload. However, we also note that while this approach works well for cases when only a small subset of processors are loaded with external workload, the performance of BSP-TW DLB_{ccle} deteriorates rapidly as more processors are loaded with external workload and are excluded from computation.

Rather than migrating all the simulation objects out of a heavily loaded processor and discarding it from computation altogether, our second solution to the problem of interruption from external workload considers migrating only parts of the simulation objects out of those heavily loaded processors. The objective is to allow the BSP-TW computation to share a slice of CPU processing time with those external workload, while preserving the overall load-balanced between supersteps. This second solution does not require the use of a threshold parameter θ and is implemented in the BSP-TW DLB_{ccls} protocol.

Experimental results comparing the performance of the BSP-TW DLB_{ccls} and BSP-TW DLB_{ccle} protocols show that BSP-TW DLB_{ccls} protocol is able to achieve better performance over BSP-TW DLB_{ccls} when a high proportion of processors in the BSP-TW computation are burdened with external workload. However, the performance of BSP-TW DLB_{ccls} drops rapidly with increasing system workload on those heavily loaded processors. By amplifying the processor system workload to exaggerate the load-imbalance of the system, we show that the BSP-TW DLB_{ccls} protocol can indeed achieve significant performance improvement over both the BSP-TW DLB_{ccl} and BSP-TW DLB_{ccle} protocols.

Chapter 7

BSP-TW with a Variable Number of Processors

7.1 Overview

From the preceding chapters, we can see that the performance of an optimistic parallel simulation protocol such as the BSP Time Warp depends on different number of factors, both internal and external. In order to maximize the performance of a long running parallel simulation, the simulation protocol needs to adapt to its surrounding environment and make the necessary changes when required. These changes can range from throttling the event limit between supersteps to dynamic load-balancing via migration of simulation objects between processors.

In Chapter 6, we see that using the maximum number of available processors in the presence of external workload can lead to performance degradation. Both the BSP-TW DLB$_{ccle}$ and BSP-TW DLB$ccls$ algorithms are able to improve the performance of the BSP-TW simulation by migrating simulation objects out of those processors that are loaded with external workload.

However, even without the presence of external workload, the inherent synchronization and communication overhead of the underlying parallel computation platform can sometimes lead to situations whereby improved performance can be achieved by using fewer processors than the maximum number of processors available.

In this chapter, we propose another approach for executing BSP-TW simulation by dynamically varying the number of processors during the runtime of a simulation. Based

on a performance cost model, we carry out experiments on a real-world semi-conductor wafer fabrication model using this new algorithm and compare its performance with the BSP-TW DLB_{ccl} algorithm described in Chapter 5.

The rest of this chapter is organized as follows. In section 7.2, we first illustrate how using fewer processors in some situations can lead to improved performance in a BSP-TW parallel simulation. We then formulate a performance cost model that can be used to decide whether to add or remove processors from a simulation run during runtime in section 7.3. Based on the performance model, the new BSP-TW DLB_{accl} algorithm for automatically adding or removing processors during runtime is described in section 7.4. Section 7.5 describes the Sematech wafer fabrication model used in the experiments. In section 7.6, we present the experimental results comparing the performance of the new BSP-TW DLB_{accl} algorithm and the BSP-TW DLB_{ccl} algorithm on a cluster of workstations as well as a shared memory system. Some related work are described in section 7.7, followed by a summary of the chapter in section 7.8.

7.2 Benefits of Using Fewer Processors

The conventional approach in trying to achieve improved performance for a parallel simulation run often involves using the maximum number of available processors for the parallel computation, in addition to other different ways to optimize the performance of the parallel run.

However, there are situations in which using the maximum number of processors available might produce worse performance compared to another run using fewer number of processors. These situations can arise due to difficulty in partitioning the simulation model, high system load due to workload from other users, as well as high system overheads in both communication and synchronization costs.

In this section, we will first illustrate how the situation can arise in which the performance of a BSP-TW execution can benefit from using fewer processors. Figure 7.1 shows a superstep configuration for four processors P0 to P3. The configuration shows that a significant proportion of the computation workload is due to rolled-back events. There is also a substantial amount of communication workload among the four processors. This can arise due to high volume of data being exchanged between the processors, or it can be due to high BSP communication parameter g. We also see that for the four-processor configuration, the barrier synchronization cost for the superstep is considerably high and

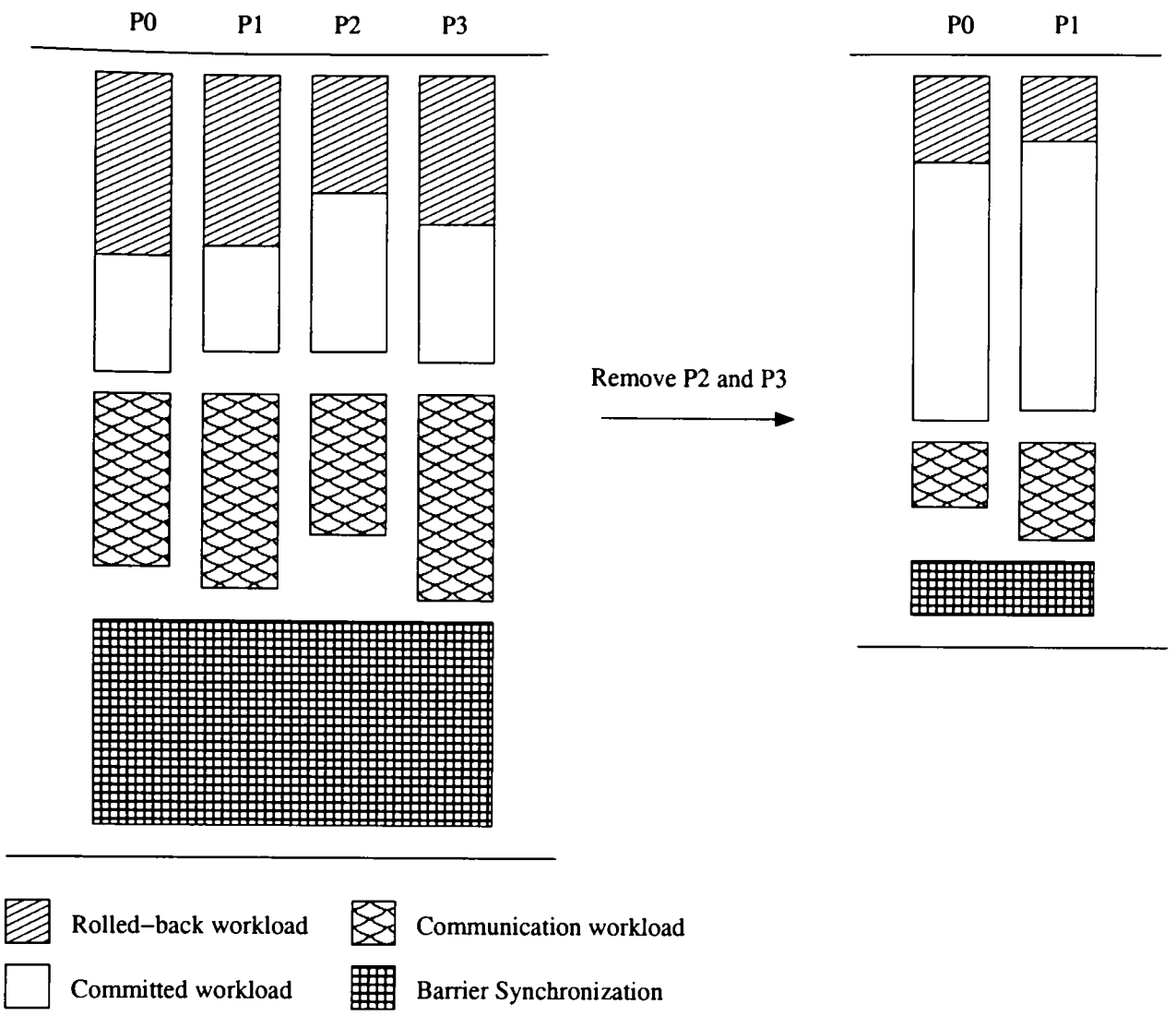


Figure 7.1: An Example to Illustrate Performance Improvement on BSP-TW by Removing Processors.

represents about 30% of the total cost for the superstep.

Suppose processors P2 and P3 are both removed from the parallel computation, and the computation workload on these two processors is distributed onto the other two remaining processors. The total (committed) computation workload for the superstep stays the same and the average (committed) computation workload on processors P0 and P1 is now doubled. However, the use of fewer processors for the BSP-TW computation may result in a significant reduction in the rolled-back workload. Also, by using only two processors for the parallel computation, both the amount of data that is exchanged between processors, as well as the BSP communication parameter g, are reduced. Similarly, we expect the BSP barrier synchronization cost l for a two-processor configuration to be much lower than that for a four-processor configuration. As shown in Figure 7.1, all these reductions in overhead can lead to a reduction in the time required to complete a superstep for the BSP-TW execution.

7.3 Performance Cost Model

In order to decide the number of processors needed for a parallel simulation, the following factors have to be considered:

- computation and communication load-imbalance;
- communication and synchronization overhead;
- event rollback rate;
- event granularity/overhead.

Let the cost of executing an event using the sequential simulation engine be c_e and the total number of events executed in the sequential execution be n_e. The total cost of sequential execution, C_{seq}, is given by

$$C_{seq} = n_e c_e. \tag{7.1}$$

Let P be the number of processors used in the parallel execution of the simulation and $k_e \geq 1$ be the event overhead in the parallel execution. The event overhead accounts for the additional cost of state-saving and fossil collection for each event in the parallel execution. The cost of executing an event in the parallel execution will be $k_e c_e$. Let $k_r \geq 0$ be the event rollback ratio for the parallel execution such that the total number of events executed in the parallel execution is $(1 + k_r)n_e$.

Let k_b $(0 \leq k_b \leq P)$ be the load-imbalance factor for the parallel execution of the simulation such that

$$k_b = \frac{n_{max} - \frac{(1+k_r)n_e}{P}}{\frac{(1+k_r)n_e}{P}}, \tag{7.2}$$

$$n_{max} = \frac{(1+k_b)(1+k_r)n_e}{P} \tag{7.3}$$

where n_{max} is the maximum number of events executed by any of the P processors.

From the BSP cost equation (4.1), the total cost of parallel execution, C_{par}, is given by

$$C_{par} = W_{par} + gH + n_s l \tag{7.4}$$

where g and l are the BSP communication and synchronization parameters. W_{par} is the accumulated total computation cost for every superstep. H is the accumulated total of the maximum bytes of data sent or received by any processor in every superstep.

The cost $gH + n_s l$ constitutes the overhead for communication and barrier synchronization. We can define k_o to be the overhead ratio such that

$$k_o = \frac{gH + n_s l}{W_{par}}. \tag{7.5}$$

The cost of parallel execution can be rewritten as

$$C_{par} = (1 + k_o) W_{par}. \tag{7.6}$$

Using equation (7.3), we can rewrite W_{par} as

$$W_{par} = \frac{(1 + k_b)(1 + k_r) k_e n_e c_e}{P}. \tag{7.7}$$

The total cost for parallel execution can be expressed as

$$C_{par} = \frac{(1 + k_o)(1 + k_b)(1 + k_r) k_e n_e c_e}{P}. \tag{7.8}$$

The achievable speedup, S, is given by

$$S = \frac{n_e c_e}{\frac{(1+k_o)(1+k_b)(1+k_r)k_e n_e c_e}{P}} \tag{7.9}$$

$$= \frac{P}{(1+k_o)(1+k_b)(1+k_r)k_e}. \tag{7.10}$$

7.4 BSP-TW DLB$_{accl}$ Algorithm

Using the DLB cost equation for BSP-TW, we propose another extension to the BSP-TW algorithm that automatically determines the required number of processors for a parallel simulation execution.

This algorithm is an extension to the BSP-TW DLB$_{ccl}$ algorithm described in Chapter 5. The BSP-TW DLB$_{ccl}$ algorithm from Figure 5.19 is enhanced with a module to automatically add or remove processors by comparing the current performance and the expected performance using the cost model developed in section 7.3.

Figure 7.2 shows the new BSP-TW DLB$_{accl}$ algorithm. The difference between BSP-TW$_{accl}$ and BSP-TW DLB$_{ccl}$ is the addition of module D0.

Module D0 is added to provide functionality for adding or removing processors based on predicted performance derived from the cost equation. The pseudo-code for the function `add_remove_cpu()` is shown in Figure 7.3. In order for the algorithm to decide whether to add/remove processor or to maintain the current number of processors used, the following terms for speedup based on equation (7.10) are computed:

- S_{cur} : the achievable speedup for the current migration interval.
- S_{add} : the achievable speedup if processors are added.
- S_{rem} : the achievable speedup if processors are removed.

To obtain k_o for the respective speedup terms, we consider equation (7.5). Since we are considering the achievable speedup for the current migration interval, n_s is set to λn_g for all three cases.

In the experiments, the number of processors is doubled or halved each time a decision is made to add or remove processors from the parallel simulation system. We make the

```
bsp_begin();
[A] Initialization;
ν'_old := 0;
while GVT < SimEndTime do
    [B] Receive external events and process rollback;
    [C] Compute new GVT, perform fossil collection and
        compute new event limit n_e every n_g supersteps;
    [D] After each λ GVT computation:
        compute new GVT rate ν'_new;
        if (1.0 − φ)ν'_old ≤ ν'_new ≤ (1.0 + φ)ν'_old then
         [D0] flag := add_remove_cpu();
         compute WB;
         compute CB;
         [D1] if WB > ε and flag = false then flag := balance_computation(); endif
         [D2] if CB > ε and flag = false then flag := balance_communication(); endif
         [D3] if flag = false then optimize_lookahead(); endif
         endif
        endif
        ν'_old = ν'_new;
    [E] Execute n_e events;
    bsp_sync();
endwhile
bsp_end();
```

Figure 7.2: Algorithm for BSP-TW DLB_{accl}.

assumption that the event overhead, k_e, is 1 no matter how many processors are used. In reality, k_e should be greater than 1 and should remain constant regardless of the number of processors used. We also set the load-balance ratio, k_b, to 0 and use the average computation workload among the active processors as W_{par}. Load-balancing among the set of active processors will be handled by modules D1 to D3 in Figure 7.2.

We can simplify equation (7.10) to

$$S = \frac{P}{(1+k_o)(1+k_r)}. \tag{7.11}$$

Assuming a well balanced communication workload across all processors, the current communication workload H will be the average of the maximum amount of data sent or received by all processors in the current migration interval. We will make use of estimates to the values of H used to compute S_{add} and S_{rem}. The value of H for S_{add} will be twice

```
add_remove_cpu()
    flag := false;
    let P_i: the set of current active processor, (0< i < n-1);
    compute S_cur, S_add and S_rem;
    if S_add > S_cur and S_add > S_rem then
        flag := true;
        // adding n processors
        let P'_i: a set of inactive processor, (0< i < n-1);
        foreach P_i (0< i < n-1) do
            migrate half of P_i's simulation objects to P'_i;
            set P'_i as active;
        endfor
    else if S_rem > S_cur and S_rem > S_add then
        flag := true;
        // removing n/2 processors
        sort P_i in descending order by computation workload
        foreach P_i (0< i < n/2-1) do
            migrate all of P_i's simulation objects to P_{i+n/2};
            set P_i as inactive;
        endfor
    endif
    return flag;
```

Figure 7.3: Pseudo Code for `add_remove_cpu()` Procedure.

the current H value. For S_{rem}, half the current H value is used.

The current value of k_r is obtained using the ratio between the total number of events rolled back and the total number of events committed in the current migration interval. We assign twice the value of k_r for S_{add}; and half the value of k_r for S_{rem}.

Again assuming a well balanced computation workload between all processors, the computation workload W_{par} is taken to be the average of the total computation workload on all processors. We use half the value of W_{par} for S_{add} and twice the value of W_{par} for S_{rem}.

The assumptions for the values of H and k_r will be verified in section 7.6. Using the values of S_{cur}, S_{add} and S_{rem}, processors are added or removed based on the following conditions:

- If the estimated value of S_{add} exceeds both S_{cur} and S_{rem}, then the algorithm allocates a set of processors from the inactive processor pool. For each of the current active processor, half the simulation objects on the active processor are migrated to

one of the inactive processors and the status of the inactive processor is set to active.

- If the estimated value of S_{rem} exceeds both S_{cur} and S_{add}, then the algorithm merges the simulation objects between each pair of active processors and set the status of one of the processors to inactive. When selecting the pair of processors, the processor with the highest computation workload is grouped with the processor with the lowest computation workload.

- If the value of S_{cur} exceeds both S_{add} and S_{rem}, then the number of processors is kept unchanged.

Note that processors that are removed from computation are flagged as inactive and modules D1 to D3 in Figure 7.2 only act on the set of active processors.

7.5 Sematech Wafer Fabrication Model

The simulation model used to benchmark the two algorithms is a manufacturing process of a wafer fabrication plant. The data model is based on the Sematech Modeling Data Standard (MDS) project [24]. The aim of the project is to "develop a set of standard that will enable the seamless exchange, sharing and re-use of data among modeling applications and Manufacturing Execution Systems (MES)". The data models are realistic examples from real-world applications. The MDS uses several files to define the manufacturing processes. The following gives descriptions for each of the six files used in the Sematech data format.

Comment File This file describes the attributes of the factory model. It includes information such as the type of product manufactured, the number of process routes, whether the factory uses operator etc. It also includes sample simulation-run parameters, and simulation results in terms of cycle time and number of wafer-lots produced for each product.

Volume Release File This file gives information about the production order. It contains information such as the product name, process route used, average lot size and the release rate. Based on the information in this file, the simulation can generate wafer-lots to be processed in the production line.

Process Route File Each product-lot must undergo a series of process steps before it leaves the production line. This is commonly known as the "recipe", or process route, for

the product. This file gives the different process routes available in the factory. Note that different products can share the same process route.

Each process route consists of multiple process steps. Each process step describes the resources needed to complete the step, together with the related timing information. A typical process step contains information such as the machine name, operation set, processing time, rework probability and rework sequence etc.

The wafer-lot will flow through the individual step specified by the process route before it leaves the production line and be ready for shipment.

Rework Route File This file specifies the various rework process steps in the factory. It contains the various rework sequences available in the production environment.

Wafers can be sent for rework at any processing step, based on the reworking probability specified by the processing step the wafer is in. If the wafer-lot is to be sent for rework at a process step, the rework sequence entry in the process step will indicate which rework sequence to use.

A rework sequence resembles closely a process route, complete with individual rework step information.

Toolset File The toolset file contains information on the individual toolset (or machine set) in the factory. Toolset and machine set are used interchangeably to refer to a set of identical tools (or machines). Each toolset contains a number of identical tools, indicated by a quantity field. Information on machine downtime is also available. The machine downtime information includes information such as the mean-time-between-failure (MTBF), mean-time-to-repair (MTTR), type of failure etc.

Operator File This file describes the different operator-sets available in the factory. Each operator-set indicates a different skill group and includes a number of identical operators with the same skill. The file specifies the different break intervals for the operators and also provides information on the shift duration.

Table 7.1 shows some statistics for the six Sematech data-sets used in the experiments. The steps/machines ratio measures the average number of processing steps sharing a given machine. Since machines are directly mapped into simulation objects in BSP-TW, the steps/machines ratio gives an indication on the average workload of the simulation objects in the simulation model.

Figures 7.4 shows an example of the complexity of the machine configuration and connectivity in the Sematech data-set. The configuration is shown for data-set 3. The boxes

	Data-set					
	1	2	3	4	5	6
No. of products	2	7	11	7	177	9
No. of process flows	2	6	11	1	21	9
No. of process steps	455	1606	4138	111	4176	2541
No. of machines	83	97	73	35	85	104
Steps/machines ratio	5.48	16.56	56.68	3.17	49.13	24.43

Table 7.1: Statistics on Sematech Data-sets.

show the different machine sets and the lines between machines represent processing steps from different process routes.

The simulation models used in the experiments are strip-down versions of the detailed model described by Sematech. Some features such as operators, reworks and machine down time are not modeled. The models also use only first-come-first-served wafer-lot processing and do not split wafer-lots on batch-processing machines. These simplifications do not reduce the complexity of the simulation models in terms of the amount of sharing of machines between different processing steps.

7.6 Experiments using the BSP-TW DLB$_{accl}$ Protocol

In this section, we present experimental results comparing the new BSP-TW DLB$_{accl}$ and the BSP-TW DLB$_{ccl}$ algorithms using the Sematech model. For all the experiments, the GVT computation interval, n_g, is fixed at 50 supersteps. The migration interval λ is set to 5. The experiments are conducted on two different parallel platforms. The first set of experiment described in section 7.6.1 is conducted on a cluster of 16 350MHz Sun UltraSparc workstations connected via a 100Mbits TCP/IP network. The second set of experiments described in section 7.6.2 is conducted on a 24 processors Sun Blade shared memory machine. A total of eight processors are used in the second set of experiments. All execution times shown are the average of three runs. A fixed simulation run length of one year (525600 time unit) is used for all simulation runs.

7.6.1 Experiments on a Cluster of Workstations

In this first set of experiments, we use a cluster of 16 workstations to execute the six Sematech data-sets. The experiments are executed using different spin-loop values to artificially

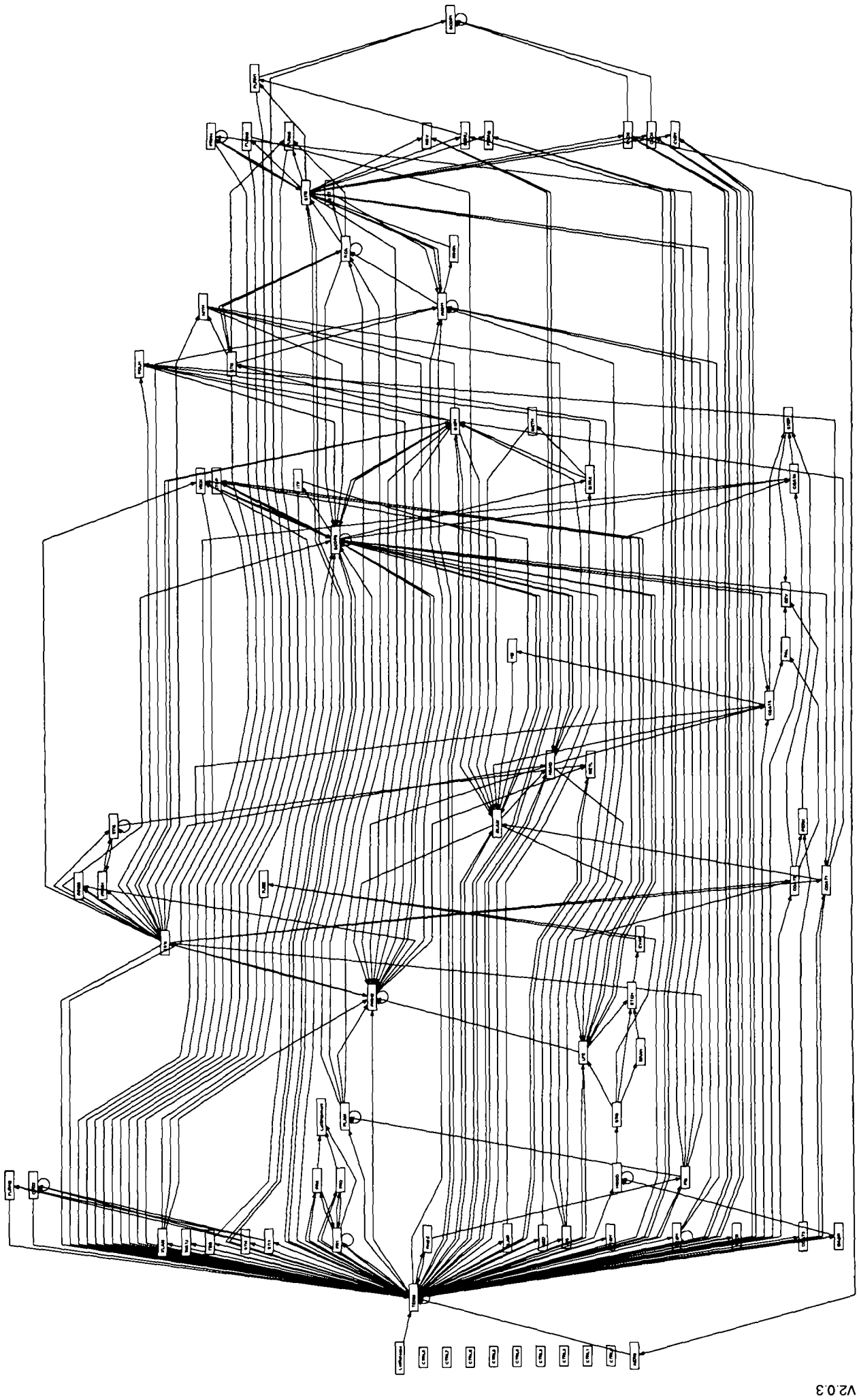


Figure 7.4: Machine Configuration for Sematech Data-set 3.

P	g (μs/byte)	l (μs)
1	0.0025	2.332
2	0.3275	1210.143
4	0.5225	2069.116
8	0.7235	3186.775
16	1.2695	8287.531

Table 7.2: BSP Parameters for a Cluster of Sun UltraSparc Workstations Connected via a 100Mbits TCP/IP Network.

	Event Granularity			
Data-set	0	10	100	1000
1	10.6	72.3	596.1	5851.4
2	14.7	97.1	850.0	8336.4
3	18.6	137.9	1198.3	11758.6
4	1.2	9.8	87.7	868.2
5	8.6	52.8	440.9	4323.5
6	7.5	54.9	467.5	4594.7

Table 7.3: Execution Times (sec.) for Sematech Data-sets using Sequential Simulation Engine.

increase the event granularity. We experiment with event granularities of 0, 10, 100 and 1000 μs for the complete duration of a simulation run. The typical event granularities in a detailed simulation model [48] are in the range of 10-100 μs.

Table 7.2 shows the values of BSP parameters g and l for different processor configurations. We see that a BSP algorithm executing under this parallel platform will be penalised for high synchronization cost as more processors are used.

Table 7.3 shows the execution times obtained using a sequential simulation engine for the six Sematech data-sets with different event granularities. Data-set 4 has the shortest execution times as it is the smallest model in the Sematech data-sets with only 35 machines and one short wafer-lot process flow.

Table 7.4 shows the execution times using the original BSP-TW on 16 processors. Table 7.5 shows the execution times using BSP-TW DLB_{ccl} on 16 processors. Comparing Tables 7.3, 7.4 and 7.5 we can see that the runs with low event granularities suffer from poor performance by using all 16 processors.

Table 7.6 shows the corresponding percentage of the execution time using BSP-TW DLB_{ccl} that is due to barrier synchronization overhead. We see that the poor performance

	Event Granularity			
Data-set	0	10	100	1000
1	1051.5	1028.2	1246.3	3436.7
2	849.2	869.2	998.1	2395.5
3	2671.4	2707.5	3145.6	7966.8
4	297.9	298.0	322.5	584.4
5	742.2	748.6	883.3	2453.0
6	619.0	627.4	742.3	2232.0

Table 7.4: Execution Times (sec.) for Sematech Data-sets using the Original BSP-TW.

	Event Granularity			
Data-set	0	10	100	1000
1	906.5	932.0	1107.9	2784.1
2	809.2	879.3	942.8	2093.8
3	2700.4	2680.5	3055.2	7570.8
4	302.7	301.1	338.2	632.0
5	630.9	600.2	757.3	1742.0
6	519.1	545.8	628.9	1443.2

Table 7.5: Execution Times (sec.) for Sematech Data-sets using BSP-TW DLB_{ccl} on 16 Processors.

	Event Granularity			
Data-set	0	10	100	1000
1	73.7	72.9	63.9	25.0
2	79.7	77.7	69.2	29.6
3	66.4	68.4	56.4	23.0
4	89.9	89.7	78.9	40.1
5	60.6	56.9	51.4	18.2
6	68.5	64.4	59.3	21.4

Table 7.6: Percentage of Execution Times Spent on Synchronization using BSP-TW DLB_{ccl} on 16 Processors.

for runs with low event granularities is due to the high synchronization overhead using all 16 processors. The performance improves with higher event granularities as the synchronization overhead decreases proportionally.

Table 7.7 shows the execution times using BSP-TW DLB_{accl}. The experiments are carried out using two configurations: C1 and C16. For configuration C1, 16 processors are allocated at the start of the run but all the simulation objects are initially partitioned onto a single processor. For configuration C16, the simulation objects are uniformly partitioned

	Config. C1				Config. C16			
	Event Granularity				Event Granularity			
Data-set	0	10	100	1000	0	10	100	1000
1	435.1	510.8	972.1	2574.9	494.6	556.3	1029.5	2437.2
2	400.6	510.3	978.7	2488.2	489.1	581.2	1038.5	2052.4
3	423.0	808.2	1902.7	6189.9	782.4	935.4	2110.2	5946.9
4	122.0	133.1	236.6	655.6	186.8	193.4	283.1	631.3
5	122.8	335.4	601.2	2496.7	249.1	321.9	573.9	1716.7
6	194.4	303.6	558.1	1724.8	285.4	337.2	661.3	1459.3

Table 7.7: Execution Times (sec.) for Sematech Data-sets using BSP-TW DLB_{accl} on 16 Processors.

	Conf. C1				Conf. C16			
	Event Granularity				Event Granularity			
Data-set	0	10	100	1000	0	10	100	1000
1	1	1	4	8	1	1	4	8
2	1	1	4	16	1	2	8	16
3	1	1	4	8	1	2	4	8
4	1	1	2	8	1	1	2	8
5	1	2	8	8	1	2	8	16
6	1	1	4	16	1	2	8	16

Table 7.8: Average Number of Processors Used by BSP-TW DLB_{accl}.

onto all 16 processors at the start of the run.

Table 7.8 shows the actual average number of active processors used during the simulation runs. Comparing Tables 7.5 and 7.7, we see that the BSP-TW DLB_{accl} protocol is able to automatically select the number of processors to use in order to achieve better performance.

However, the performance for the runs with low event granularities is still very much worse than sequential runs. This is attributed to the fact that the inherent synchronization overhead of 16 processors is still present using BSP-TW DLB_{accl} even though the protocol uses only a small subset of the processors for computation. To verify this, we carried out experiments using BSP-TW DLB_{ccl} with different fixed numbers of processors.

Table 7.9 shows the execution times for the Sematech data-sets for this set of experiments. The numbers in bold show the execution times using the number of processors determined by the BSP-TW DLB_{accl} protocol in Table 7.8 for configuration C1. We see that the runs using the number of processors determined by BSP-TW DLB_{accl} give the best

	Set1				Set2			
	Event Granularity				Event Granularity			
P	0	10	100	1000	0	10	100	1000
1	**40.3**	**96.6**	625.2	5913.5	**45.4**	**131.3**	887.1	8367.3
2	216.9	255.2	601.9	4107.3	208.8	269.5	704.7	5177.4
4	336.7	349.7	**551.9**	2498.6	308.3	330.8	**586.4**	3149.3
8	489.7	517.6	638.3	**2087.8**	480.7	517.5	689.3	2404.7
16	906.5	932.0	1107.9	2784.1	809.2	879.3	942.8	**2093.8**

	Set3				Set4			
	Event Granularity				Event Granularity			
P	0	10	100	1000	0	10	100	1000
1	**51.7**	**169.1**	1231.6	11803.5	**8.5**	**16.8**	95.4	880.7
2	365.9	443.0	1132.3	8176.9	38.9	43.5	**93.2**	591.5
4	631.2	673.5	**1044.6**	5023.2	94.3	94.6	127.9	439.0
8	1156.0	1221.2	1487.8	**4832.7**	170.5	184.9	198.9	**321.0**
16	2700.4	2680.5	3055.2	7570.8	302.7	301.1	338.2	632.0

	Set5				Set6			
	Event Granularity				Event Granularity			
P	0	10	100	1000	0	10	100	1000
1	**17.3**	62.6	450.3	4366.0	**22.2**	**68.8**	481.6	4609.7
2	115.3	**134.5**	362.9	2645.5	156.3	175.6	432.5	3001.3
4	183.4	194.0	328.9	1680.8	206.9	221.0	**370.0**	1837.3
8	274.4	279.7	**355.9**	**1227.3**	291.7	307.9	391.5	1279.7
16	630.9	600.2	757.3	1742.0	519.1	545.8	628.9	**1443.2**

Table 7.9: Execution Times (sec.) for Sematech Data-sets using BSP-TW DLB_{ccl} with P Processors.

performance achievable in most cases.

Comparing Tables 7.7 and 7.9, we can see the additional synchronization overhead in using a subset of processors out of the full 16 processors allocated. For example, the run for data-set 4 with event granularity 0 only takes 8.5 seconds to complete using BSP-TW DLB_{ccl} on one processor (with only one processor allocated), as opposed to 122.0 seconds on BSP-TW DLB_{accl} on configuration C1 (16 processors allocated, but all simulation objects are initially mapped onto a single processor).

In order to verify the assumptions on (doubling/halving) the values of rollback ratio k_r and communication workload H when processors are added or removed, the corresponding values of k_r and H on different number of processors are recorded during the simulation runs. Tables 7.10 and 7.11 show the percentages of events rolled back and the communi-

	Number of Processors				
Data-set	1	2	4	8	16
1	0.0	13.4	11.7	17.1	30.6
2	0.0	4.8	7.5	15.5	21.0
3	0.0	16.4	17.9	31.5	58.3
4	0.0	9.7	17.9	30.5	37.6
5	0.0	8.4	9.9	18.4	45.0
6	0.0	11.9	12.5	18.5	34.5

Table 7.10: Percentage of Events Rolled Back for Sematech Data-sets using BSP-TW DLB_{ccl} with Fixed Number of Processors.

	Number of Processors				
Data-set	1	2	4	8	16
1	0.0	9.3	15.7	21.7	49.7
2	0.0	9.0	14.9	17.3	21.0
3	0.0	17.1	31.3	54.6	140.6
4	0.0	1.0	2.4	3.9	7.4
5	0.0	5.8	10.3	12.1	31.7
6	0.0	7.3	10.9	10.4	21.1

Table 7.11: Communication Workload (sec.), H, for Sematech Data-sets using BSP-TW DLB_{ccl} with Fixed Number of Processors.

cation workload for the Sematech data-sets using BSP-TW DLB_{ccl}. Although our assumptions do not produce an exact match on the variations of k_r and H for all six data-sets, they do reflect the general trend in the increasing values of k_r and H when more processors are added to the system.

7.6.2 Experiments on a Shared Memory Machine

In this second set of experiments, we compare the performance of the BSP-TW DLB_{accl} algorithm obtained so far on the clusters of Sun workstations with results obtained from executing the same simulation model on OSWELL, a shared memory multi-processor system. OSWELL consists of 84 900MHz Sun UltraSPARC III processors arranged in three groups of 24 plus a group of 12 processors. The 24-processor groups each have 48Gb of memory; the processors in the 12 processor group has 24Gb of memory. As the system is heavily utilized, to ensure exclusive use of all the processors in the experiment, we used only a total of 8 processors out of a 24-processor group in all the runs.

P	g (μs/byte)	l (μs)
1	0.0000	0.450
2	0.0025	12.141
4	0.0050	13.692
8	0.0102	20.888

Table 7.12: BSP Parameters for OSWELL Shared Memory System.

	Event Granularity			
Data-set	0	10	100	1000
1	3.7	34.0	306.6	3031.9
2	5.1	48.2	436.6	4320.4
3	6.8	67.6	615.7	6094.7
4	0.4	4.9	45.5	450.1
5	2.6	25.0	227.1	2238.9
6	2.7	26.4	240.1	2369.3

Table 7.13: Execution Times (sec.) for Sematech Data-sets using Sequential Simulation Engine on OSWELL.

Table 7.12 shows the BSP parameters g and l for the different processor configurations on OSWELL. Compared with the BSP parameters in Table 7.2 for the Sun cluster, we see that both the cost of communication and synchronization on OSWELL are significantly lower.

Figure 7.13 shows the sequential execution times for the six Sematech data-sets with different event granularities. In comparison with the performance achieved on the Sun cluster shown in Table 7.3, the sequential performance of OSWELL is about twice that of the Sun cluster.

Figure 7.14 shows the execution times for the six Sematech data-sets using BSP-TW DLB_{ccl} with 8 processors. The performance of the parallel execution is comparable to the sequential performance at event granularity of $10\mu s$. The numbers in brackets are the corresponding speedup with respect to the sequential run in Table 7.13. For the runs with very small event granularities, the BSP-TW DLB_{ccl} algorithm did not manage to achieve any performance gain. For the runs with high event granularities, speedup in the range of 2.0 to 4.8 are achieved.

Figure 7.15 shows the percentage of execution times spent on barrier synchronization. Due to the low BSP parameters for synchronization on this platform, less than 5% of the execution times are spent on synchronization for all the runs.

	Event Granularity			
Data-set	0	10	100	1000
1	36.0 (0.1)	41.6 (0.8)	109.6 (2.8)	820.1 (3.7)
2	33.4 (0.2)	41.4 (1.2)	120.1 (3.6)	944.6 (4.6)
3	64.0 (0.1)	69.6 (1.0)	221.6 (2.8)	1858.2 (3.3)
4	14.4 (0.02)	13.3 (0.03)	23.0 (2.0)	153.8 (2.9)
5	13.8 (0.2)	16.0 (1.6)	56.8 (4.0)	477.8 (4.7)
6	19.0 (0.1)	20.7 (1.3)	62.9 (3.8)	493.2 (4.8)

Table 7.14: Execution Times (sec.) for Sematech Data-sets using BSP-TW DLB_{ccl} using 8 Processors on OSWELL.

	Event Granularity			
Data-set	0	10	100	1000
1	4.9	4.2	1.6	0.2
2	4.8	3.8	1.3	0.2
3	4.7	4.5	1.3	0.2
4	4.7	5.0	2.9	0.4
5	4.4	3.8	1.1	0.1
6	4.4	4.1	1.3	0.2

Table 7.15: Percentage of Execution Times Spent on Synchronization using BSP-TW DLB_{ccl} with 8 Processors on OSWELL.

	Config. C1				Config. C8			
	Event Granularity				Event Granularity			
Data-set	0	10	100	1000	0	10	100	1000
1	33.5	38.2	115.4	911.5	36.9	40.8	109.5	824.8
2	28.4	38.0	131.0	1061.7	32.2	40.3	119.9	939.7
3	58.4	66.0	247.6	2119.6	60.4	67.7	223.7	1875.4
4	9.4	12.0	27.1	196.1	12.0	12.4	23.0	152.5
5	12.9	22.1	100.0	807.9	13.2	16.2	57.2	500.2
6	15.2	19.2	80.3	689.2	17.4	21.5	62.4	498.3

Table 7.16: Execution Times (sec.) for Sematech Data-sets using BSP-TW DLB_{accl} with 8 Processors on OSWELL.

Next, the six Sematech data-sets are executed using the BSP-TW DLB_{accl} algorithm. We again use two different configurations, C1 and C8, to set the number of active processors at the start of the simulation runs to 1 and 8 respectively. Figure 7.16 shows the execution times using the two different configurations.

Figure 7.17 shows the average number of active processors used by the BSP-TW DLB_{accl} algorithm for the six Sematech data-sets under different event granularities. We see that for

	Conf. C1				Conf. C8			
	Event Granularity				Event Granularity			
Data-set	0	10	100	1000	0	10	100	1000
1	4	8	8	8	4	8	8	8
2	8	8	8	8	8	8	8	8
3	8	8	8	8	8	8	8	8
4	4	8	8	8	4	8	8	8
5	8	8	8	8	8	8	8	8
6	8	8	8	8	8	8	8	8

Table 7.17: Average Number of Processors Used by BSP-TW DLB_{accl} on OSWELL.

event granularities of $10\mu s$ and above, the algorithm chooses to make use of all 8 processors. This is attributed to the low synchronization and communication costs on the shared memory environment. For the runs with very small event granularities, the BSP-TW DLB_{accl} algorithm still chooses to make use of all 8 processors for data-sets 2, 3, 5 and 6. However, for data-sets 1 and 4, the BSP-TW DLB_{accl} algorithm uses only 4 out of the 8 processors available. This decision allows the BSP-TW DLB_{accl} algorithm to achieve better performance for data-sets 1 and 4 for configuration C1.

For the runs with high event granularities, the performance using configuration C1 suffers due to the overhead in expanding from one active processor to the maximum number of available processors.

Table 7.18 shows the execution times for BSP-TW DLB_{ccl} using different number of processors with different event granularities. The numbers in bold corresponds to the number of processors chosen by the BSP-TW DLB_{accl} shown in Table 7.17. For the runs with low event granularities, we see that the number of processors chosen by the BSP-TW DLB_{accl} algorithm do not yield the best result possible. However, for the runs with high event granularities, the number of processors chosen by the BSP-TW DLB_{accl} algorithm yield the best possible results for all six data-sets.

From this set of experiment, we see that the BSP-TW DLB_{accl} provides much more accurate adjustment for the distributed memory cluster compared to the shared memory multiprocessor system. With both the low synchronization and communication costs, the shared memory multi-processors environment allows the parallel simulation to use the maximum number of processors available without any performance penalty. The same is however not true for the distributed memory cluster due to the high synchronization and communication costs. A parallel simulation running under the distributed memory cluster may incur unnecessarily high overhead if it always chooses to use the maximum number of processors

	Set1				Set2			
	Event Granularity				Event Granularity			
P	0	10	100	1000	0	10	100	1000
1	10.2	40.6	313.1	3039.7	11.7	54.9	443.4	4328.7
2	15.6	36.3	220.5	2067.4	15.4	40.9	267.4	2463.0
4	**21.0**	33.1	134.7	1197.0	20.6	35.0	178.1	1584.9
8	36.0	**41.6**	**109.6**	**820.1**	**33.4**	**41.4**	**120.1**	**944.6**

	Set3				Set4			
	Event Granularity				Event Granularity			
P	0	10	100	1000	0	10	100	1000
1	14.2	75.2	623.1	6102.4	2.1	6.5	47.0	452.0
2	23.0	65.0	429.1	4010.4	3.2	6.0	31.6	285.8
4	29.8	50.1	262.3	2585.3	**5.8**	7.3	23.5	177.3
8	**64.0**	**69.6**	**221.6**	**1858.2**	14.4	**13.3**	**23.0**	**153.8**

	Set5				Set6			
	Event Granularity				Event Granularity			
P	0	10	100	1000	0	10	100	1000
1	4.8	27.1	228.7	2241.1	6.0	29.7	242.7	2372.5
2	7.3	20.9	139.3	1312.3	9.3	24.3	156.1	1457.6
4	9.3	16.8	89.0	1555.5	12.0	20.5	100.2	939.4
8	**13.8**	**16.0**	**56.8**	**477.8**	**19.0**	**20.7**	**62.9**	**493.2**

Table 7.18: Execution Times (sec.) for Sematech Data-sets using BSP-TW DLB_{ccl} with P Processors on OSWELL.

available.

7.7 Related Work

A study reported in [46] examines the behaviour of different load-balancing algorithms when the number of processors is dynamically changed during the lifetime of a multi-stage parallel computation. The approach is to add or remove processors at different stages of the parallel computation based on the memory requirement. While the partitioning algorithms used in the study do try to minimize communication cost among processors, the decision to add or remove processors does not take into consideration the communication cost for adding or removing them.

In another related study reported in [43], the authors use regular check-pointing to save the states of a BSP computation. The load of the individual processors are monitored at

a regular interval. When one or more of the processors used in the computation become heavily loaded, an algorithm is activated to search for a new set of processors with a lower system load. If such a set of processors is found, the BSP computation is terminated and restarted on the set of less loaded processors.

We note that this is a viable approach for dynamically varying the number of processors in the system without the overhead of synchronization cost due to the presence of inactive processors. In this case, the BSP-TW computation can be check-pointed, terminated, and restarted on a larger or smaller set of processors.

7.8 Summary

In this chapter, we illustrated with examples how the best possible performance may not always be achieved using the maximum number of processors available and that improved performance can sometimes be achieved using fewer processors instead.

Using the BSP cost model for the BSP-TW algorithm, a new procedure for deciding when to add or remove processors during simulation runtime is proposed. This decision procedure is incorporated into the new BSP-TW DLB_{accl} algorithm.

Using a set of real-world semiconductor wafer fabrication models, we carried out experiments to study the effectiveness of the new BSP-TW DLB_{accl} algorithm in automatically adjusting the number of processors for different type of event granularities.

Our experimental results show that with the flexibility of the BSP-TW DLB_{accl} algorithm to automatically adjust the number of processors in the parallel computation, significant performance improvement can be achieved compared to the conventional approach which always uses the maximum number of available processors. Our results also indicate that this approach is more suited to distributed memory system and less so for shared memory system due to its inherent low overhead in communication and synchronization costs.

Our study also shows that optimal performance could be achieved if the underlying runtime library support efficient synchronization and communication for a subset of processors. Further work will be carried out in this area.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

The main contribution of this thesis is the study and development of different adaptive techniques to optimize and improve the consistency, performance and resilience of BSP-TW simulation. These studies include eliminating risk hazards, adaptive tuning of event limits, dynamic load-balancing of computation and communication workload, optimization of lookaheads, management of external workload and dynamically adjusting the number of active processors during runtime.

On Consistency

We began the study by examining the issue of risk hazard in BSP-TW. Using a distributed mutual exclusion simulation model with strong consistency properties, we showed that inconsistency errors are present when the simulation model is executed using BSP-TW. We also showed that these errors can be eliminated using anti-message acknowledgement in BSP-TW. Although having anti-message acknowledgement in the simulation protocol solves the problem of rollback inconsistencies, it can cause severe performance degradation in the parallel simulation. Besides, it does not resolve the problem caused by stale states.

We carried out successive refinements to the BSP-TW protocol and proposed the technique of extended barrier. The extended barrier has been implemented in BSP-TW and shown to be capable of eliminating errors caused by rollback inconsistencies as well as stale states. Experimental results show that BSP-TW with extended barrier is able to achieve

comparable performance to that of the original BSP-TW with suitable level of throttling being applied on the superstep event limit.

On Adaptive Event Limit

To investigate the effects of varying event limit have on a BSP-TW parallel simulation run, we carried out experiments on BSP-TW that involves adaptive computation of the event limit to use in each superstep of the BSP-TW execution. Using the BSP cost analysis, we predicted that improved performance can be achieved from a BSP-TW execution by adaptively determining an event limit for each superstep that reduces the proportion of rolled-back events at the expense of using more supersteps. A heuristic scheme has been implemented in BSP-TW. The scheme seeks to minimize load-imbalance between processors across supersteps as well as reducing the event rollback ratio. We conducted experiments using different simulation models on a SGI Original 2000 system with low synchronization overhead. While our studies show that significant execution time reduction can indeed be achieved for different simulation models, they also point to limitations of performing load-balancing statically by regulating superstep event limit. For load-balancing to be truly effective, some mechanisms to migrate simulation workload between processors need to be developed.

On Dynamic Load-Balancing

With that in mind, the next set of optimizations on BSP-TW focus on dynamic load-balancing via migration of simulation objects between processors. We derived a dynamic load-balancing cost model which predicts that the performance of the dynamic load-balancing algorithm depends on its ability to balance both computation and communication load-imbalance, as well as the optimization of lookaheads between processors. The BSP-TW DLB_{ccl} algorithm with all these three capabilities was developed progressively. The performance of this algorithm has been tested using different simulation model with different partitioning methods. Experimental results show that the BSP-TW DLB_{ccl} is able to achieve significant performance improvement over the original BSP-TW on simulation models that have imbalance in computation or communication workload, as well as those that suffer from poor lookahead configurations due to poor partitioning strategies.

On Management of External Workload

While the BSP-TW DLB$_{ccl}$ algorithm is effective in situations where there is an imbalance in simulation workload or the presence of small lookaheads between processors, it is designed to work in a dedicated computing environment which is free from the interruption of external workload. We showed that the performance of the original BSP-TW and BSP-TW DLB$_{ccl}$ can degrade rapidly when external workload is introduced onto a small subset of the processors in the parallel computation.

In order to improve the resilience and predictability of BSP-TW computation in the presence of external workload, we further extended the BSP-TW DLB$_{ccl}$ to handle interruption due to the presence of external workload. We proposed and implemented two solutions to resolve this issue. The first solution implemented in BSP-TW DLB$_{ccle}$ rapidly evicts all the simulation out of a processor whenever its system load exceeds a given threshold. The second solution implemented in BSP-TW DLB$_{ccls}$ considers the allocated processor time slice for simulation workload on those heavily loaded processors and only migrates part of the simulation objects out of them.

Experimental results show that both algorithms out-perform the original BSP-TW algorithm in the presence of persistence external workload. However, as the BSP-TW DLB$_{ccle}$ algorithm works by removing all simulation objects from heavily loaded processors and excluding them from computation, its performance deteriorates rapidly when increasing number of processors in the system are loaded with external workload. On the other hand, the BSP-TW DLB$_{ccls}$ algorithm is able to maintain high performance in the same situation by sharing slices of processing cycles with other external workload on those heavily loaded processors.

On Varying the Number of Processors

Even if there is no interference from external workload, there are situations in which using fewer processors than the maximum available in the system can result in better performance. These situations can arise in systems where there are high communication and synchronization costs, as well as situations in which the event granularities in the simulation model do not offer any benefits in carrying out parallel computation. Using the BSP cost model, we explained how these situations can arise and proposed a decision procedure for adding and removing processors during the runtime of a BSP-TW simulation. This decision procedure has been implemented in the BSP-TW DLB$_{accl}$ algorithm.

Experiments have been conducted using the BSP-TW DLB$_{accl}$ algorithm on both distributed memory system and shared memory system. The simulation model used is a set of semi-conductor wafer fabrication models. The experimental results show that the BSP-TW DLB$_{accl}$ algorithm is able to both expand and shrink the set of active processors based on the event granularities of the simulation models as well as the characteristics of the underlying parallel platform.

8.2 Future Work

The different adaptive techniques that have been presented in this thesis lay the foundation for further research into improving the consistency, predictability and resilience for parallel simulation protocols such as BSP-TW. Further research should be carried in the following areas.

On Simulation Model and Workload

We have examined the performance BSP-TW DLB$_{accl}$ using a stripped-down version of the Sematech wafer manufacturing model. Further work needs to be carried out to include more details in the model. Having a more detailed simulation model changes not just the granularities of each event, but also the connectivity and lookaheads between simulation objects. For example, adding a new scheduling rule that requires instant information on the queue length of downstream machines could potentially introduce many zero-lookahead links into the simulation model.

Instead of simulating artificial external workload in a controlled environment, experiments involving external workload should also be carried out in real-world multi-user environments in order to carry out more realistic studies on the behaviour of the dynamic load-balancing algorithm.

On Partitioning

The lookahead optimization module in the BSP-TW DLB$_{ccl}$ algorithm can be considered as performing repartitioning of simulation objects onto the processors. Its main aim is to minimize lookaheads between processors. As the lookahead optimization module does not take into account effects on the load-balance of the computation and communication

workload of the processors when merging simulation objects with small lookaheads, load-imbalance problems such as those described in section 6.5 can arise and hence affecting performance of the simulation run.

More research should be carried out to devise new partitioning methods that take into account the workload distribution among processors when optimizing lookaheads between processors. To take this idea one step further, the modules for balancing both computation and communication workload should be re-examined to explore the possibility of improving lookaheads while performing load-balancing.

On Subset Synchronization

In section 7.6, we observed that the cost of a BSP barrier synchronization is not reduced even when the BSP-TW DLB_{accl} protocol has shrunk the set of active processors used for the parallel computation by migrating simulation objects out of some processors. The reason is that although these inactive processors do not have any computation workload, they are still involved in the global barrier synchronization between supersteps.

As described in section 7.7, one possible solution to reducing the communication and synchronization cost when reducing the number of active processors is to checkpoint and terminate the simulation. A new set of processors with the required size will be selected and the simulation restarted on this new set of processors.

As the Oxford BSP Toolset used in this thesis does not have facilities for synchronization of subset of processors, further work could be carried out to incorporate efficient subset synchronization into the library. Another approach is to explore other implementations of BSP libraries or parallel runtime systems that provide the functionality for BSP type of computation as well as supporting efficient subset communication and synchronization.

An example is the Paderborn University BSP-Library (PUB-Library) [4] which provides a set of programming routines for implementing BSP algorithm. Besides the standard BSP communication routine, the PUB-Library also provides some collective communication operations such as broadcast and parallel prefix. More importantly, it also provides support for partitioning the BSP processors into subgroups and each subgroup acts like an independent BSP computer. Further work should be carried out to examine the efficiency of the subgroup implementation in PUB-Library and the performance of the BSP-TW DLB_{accl} algorithm using this implementation of the BSP library.

Bibliography

[1] H. Avril and C. Tropper. The Dynamic Load Balancing of Clustered Time Warp for Logic Simulation. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS'96)*, pages 20–27, Philadelphia, Pennsylvania, USA, May 1996.

[2] R. Ayani and H. Rajaei. Parallel Simulation Using Conservative Time Windows. In *Proceedings of the 1992 Winter Simulation Conference*, pages 709–717, December 1992.

[3] R. Bagrodia. *PARSEC User Manual Release 1.0*. UCLA Parallel Computing Lab, 1997.

[4] O. Bonorden, N. Hüppelshäuser, B. Juurlink, and I. Rieping. PUB-Library, Release 7.0, User Guide and Function Reference. Technical Report tr-rsfb-00-070, Heinz Nixdorf Institute, Department of Computer Science, University of Paderborn, Germany, 2000.

[5] A. Boukerche, S.K. Das, A. Fabbri, and O. Yildiz. Exploiting Model Independence for Parallel PCS Network Simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation (PADS'99)*, pages 166–173, Atlanta, Georgia, USA, 1-4 May 1999.

[6] R.E. Bryant. A Switch-Level Model and Simulator for the MOS Digital Systems. *IEEE Transactions on Computers*, C-33(2):160–177, February 1984.

[7] C. Burdorf and J. MArti. Load Balancing Strategies for Time Warp on Multi-User Workstations. *The Computer Journal*, 36(2):168–176, 1993.

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

[9] C.D. Carothers, R. Fujimoto, Y.-B. Lin, and P. England. Distributed Simulation of Large-Scale PCS Networks. In *Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 2–6, January 1994.

[10] C.D. Carothers and R.M. Fujimoto. Background Execution of Time Warp Programs. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS'96)*, pages 12–19, Philadelphia, Pennsylvania, USA, May 1996.

[11] C.D. Carothers, K.S. Perumalla, and R.M. Fujimoto. Efficient Optimistic Parallel Simulations using Reverse Computation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation (PADS'99)*, pages 126–135, Atlanta, Georgia, USA, 1-4 May 1999.

[12] C.D. Carothers, B. Topol, R.M. Fujimoto, J. T. Stasko, and V. Sunderam. Visualizing Parallel Simulation in Network Computing Environments: A Case Study. In *Proceedings of the 1997 Winter Simulation Conference*, pages 110–117, Atlanta, Georgia, USA, 7-10 December 1997.

[13] 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):440–452, September 1979.

[14] K.M. Chandy and J. Misra. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM*, 24(11):198–205, November 1981.

[15] G. Chen and B.K. Szymanski. Lookback: A New Way of Exploring Parallelism in Discrete Event Simulation. In *16th Workshop on Parallel and Distributed Simulation (PADS 2002)*, pages 153–162, Washington, D.C., USA, 12-15 May 2002.

[16] M. Choe and C. Tropper. On Learning Algorithms and Balancing Loads in Time Warp. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation (PADS'99)*, pages 101–108, Atlanta, Georgia, USA, 1-4 May 1999.

[17] J.S. Dahmann. The High Level Architecture and Beyond: Technology Challenges. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation (PADS'99)*, pages 64–70, Atlanta, Georgia, USA, 1-4 May 1999.

[18] O. Damani and V. Garg. Fault-tolerant Distributed Simulation. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation (PADS'98)*, pages 38–45, Ban, Alberta, Canada, 26-29 May 1998.

[19] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: A Time Warp System for Shared Memory Multiprocessors. In *Proceedings of the 1994 Winter Simulation Conference*, pages 1332–1339, 1994.

[20] S.R. Das. Adaptive Protocols for Parallel Discrete Event Simulation. In *Proceedings of the 1996 Winter Simulation Conference*, pages 186–193, 1996.

[21] S.R. Das and R.M. Fujimoto. A Performance Study of the Cancelback Protocol for Time Warp. In *Proceedings of the 1993 Workshop on Parallel and Distributed Simulation*, pages 135–142, San Diego, California, USA, 16-19 May 1993.

[22] L.M. D'Souza, X. Fan, and P.A. Wilsey. pGVT: An Algorithm for Accurate GVT Estimation. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS '94)*, 1994.

[23] K. El-Khatib and C. Tropper. On Metrics for the Dynamic Load Balancing of Optimistic Simulation. In *Proceedings of the 32nd Hawaii International Conference on System Sciences*, 1999.

[24] G. Feigin, J. Fowler, J. Robinson, and R. Leachman. *Semiconductor Wafer Manufacturing Data Format Specification*. Sematech, 19 July 1994.

[25] S.L. Ferenci, R.M. Fujimoto, M.H. Ammar, and K. Perumalla. Updatable Simulation of Communication Networks. In *16th Workshop on Parallel and Distributed Simulation (PADS 2002)*, pages 107–114, Washington, D.C., USA, 12-15 May 2002.

[26] A. Ferscha and J. Johnson. A Testbed for Parallel Simulation Performance Prediction. In *Proceedings of the 1996 Winter Simulation Conference*, pages 637–664, 1996.

[27] A. Ferscha and J. Johnson. Shock Resistant Time Warp. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation (PADS'99)*, pages 92–100, Atlanta, Georgia, USA, 1-4 May 1999.

[28] R.M. Fujimoto. Time Warp on a Shared Memory Multiprocessor. In *Proceedings of the 1989 International Conference on Parallel Processing*, 1989.

[29] R.M. Fujimoto. Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp. *IEEE Transaction of Computer*, 33(10):68–82, 1990.

[30] R.M. Fujimoto. Performance of Time Warp under Synthetic Workloads. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 22, pages 23–28, January 1990.

[31] R.M. Fujimoto. Parallel Discrete Event Simulation: Will the Field Survive? *ORSA Journal on Computing*, 5(3):213–230, Summer 1993.

[32] R.M. Fujimoto. Zero Lookahead and Repeatability in the High Level Architecture. In *Spring Simulation Interoperability Workshop*, Orlando, Florida, USA, 3-7 March 1997.

[33] R.M. Fujimoto. Exploiting Temporal Uncertainty in Parallel and Distributed Simulations. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation (PADS'99)*, pages 46–53, Atlanta, Georgia, USA, 1-4 May 1999.

[34] R.M. Fujimoto and M. Hybinette. Computing Global Virtual Time in Shared-Memory Multiprocessors. *ACM Transaction on Modeling and Computing Simulation*, 7(4):425–446, 1997.

[35] R.M. Fujimoto and R.M. Weatherly. Time Management in the DoD High Level Architecture. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS'96)*, pages 60–67, Philadelphia, Pennsylvania, USA, May 1996.

[36] A. Gafni. Rollback Mechanism for Optimistic Distributed Simulation System. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 19, pages 61–67. SCS, February 1988.

[37] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM: Parallel Virtual Machine. THE MIT Press, 1994.

[38] J. Gilmer. An Assessment of Time Warp Parallel Discrete Event Simulation Algorithm Performance. In B. Unger and D. Jefferson, editors, *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 19, pages 45–49. Society for Computer Simulation, February 1988.

[39] D.W. Glazer and C. Troper. On Process Migration and Load Balancing in Time Warp. *IEEE Transactions on Parallel and Distributed Systems*, 4(3):318–327, 1993.

[40] J. H. Graham, I. S. Karachiwala, and A. S. Elmaghraby. Evaluation of a Prototype Visualization for Distributed Simulations. In *Proceedings of the 1998 Winter Simulation Conference*, pages 1469–1477, Washington, D.C., USA, 13-16 December 1998.

[41] S.P. Griffin, E.H. Page, Z. Furness, and M.C. Fischer. Providing Uninterrupted Training to the Joint Training Confederation (JTC) Audience During Transition to the High

Level Architecture (HLA). In *Proceedings of the 1997 Simulation Technology and Training Conference*, pages 197–201, Canberra, Australia, 17-20 March 1997.

[42] D.O. Hamnes and A. Tripathi. Investigations in Adaptive Distributed Simulation. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS '94)*, pages 20–23, 1994.

[43] J.M.D. Hill, S.R. Donaldson, and T. Lanfear. Process Migration and Fault Tolerance of BSPlib Programs Running on Networks of Workstations. In *EuroPar'98*, volume 1470, pages 80–91. LNCS, Springer-Verlag, September 1998. [see also Technical Report PRG-TR-41-97, Programming Research Group, Oxford University Computing Laboratory, December 1997].

[44] M. Hybinette and R. Fujimoto. Dynamic Virtual Logical Processes. In *12th Workshop on Parallel and Distributed Simulation (PADS'98)*, pages 100–107, Banff, Alberta, Canada, 26-29 May 1998.

[45] M. Hybinette and R.M. Fujimoto. Scalability of Parallel Simulation Cloning. In *Proceedings of the 35th Annual Simulation Symposium*, pages 275–282, San Diego, California, USA, 14-18 April 2002.

[46] S. Iqbal, G.F. Carey, M.A. Padron, J.P. Suarez, and A. Plaza. Load Balancing with Variable Number of Processors on Commodity Clusters. In *Proceedings of the High Performance Computing Symposium 2002*, San Diego, California, USA, 14-18 April 2002.

[47] S. Jain and S. Chen. Experiences with Backward Simulation Based Approach for Lot Release Planning. In *Proceedings of the 1997 Winter Simulation Conference*, pages 773–780, Atlanta, Georgia, USA, 7-10 December 1997.

[48] S. Jain, C.-C. Lim, B.-P. Gan, and Y.-H. Low. Criticality of Detailed Modeling in Semiconductor SupplyChain Simulation. In P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, editors, *1999 Winter Simulation Conference (WSC'99)*, pages 888–896. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey, 1999.

[49] D. Jefferson. Virtual Time. In *ACM TOPLAS*, volume 7, pages 404–425, 1985.

[50] D. Jefferson and H. Sowizral. Fast Concurrent Simulation Using Time Warp Mechanism. In *Distributed Simulation 1995*, pages 63–69, La Jolla, California, USA, 1985. SCS-The Society for Computer Simulation, Simulation Councils, Inc.

[51] JPL. *Time Warp Operating System User's Manual*, 1991.

[52] M. Liljenstam and R. Ayani. A Model for Parallel Simulation of Mobile Telecommunication Systems. In *Proceedings of the Fourth International Workshop on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'96)*, pages 168–173, San Jose, California, USA, February 1996.

[53] C.-C. Lim, Y-H. Low, W. Cai, W.-J. Hsu, S.-Y. Huang, and S.J. Turner. An Empirical Comparison of Runtime Systems for Conservative Parallel Simulation. In *2nd Workshop on Runtime Systems for Parallel Programming (RTSPP 1998)*, pages 123–134, Orlando, Florida, USA, 30 March 1998.

[54] C.-C. Lim, Y.-H. Low, B.-P. Gan, S.Jain, W. Cai, S.-Y. Huang, and W.-J. Hsu. Performance Prediction Tools for Parallel Discrete Event Simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation (PADS'99)*, pages 148–155, Atlanta, Georgia, USA, 1-4 May 1999.

[55] Y.-B. Lin, B.R. Preiss, W.M. Loucks, and E.D. Lazowska. Selecting the Checkpoint Interval in Time Warp Simulation. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS '93)*, 1993.

[56] M.Y.H. Low. Adaptive BSP Time Warp. In *Proceedings of the Fifth UK Simulation Society Conference (UKSim 2001)*, pages 14–20, Cambridge, UK, 28-30 March 2001.

[57] M.Y.H. Low. Dynamic Load-Balancing for BSP Time Warp. In *Proceedings of the 35th Annual Simulation Symposium*, pages 267–274, San Diego, California, USA, 14-18 April 2002.

[58] M.Y.H. Low. Managing External Workload with BSP Time Warp. In *Proceedings of the 2002 Winter Simulation Conference (to appear)*, San Diego, California, USA, 8-11 December 2002.

[59] M.Y.H. Low. Manufacturing Simulation using BSP Time Warp with Variable Number of Processors. In *Proceedings of the 2002 European Simulation Symposium (to appear)*, Dresden, Germany, 23-26 October 2002.

[60] M.Y.H. Low and D.M. Nicol. Consistent Modeling of Distributed Mutual Exclusion Protocol using Optimistic Simulation. In *15th Workshop on Parallel and Distributed Simulation (PADS 2001)*, pages 137–144, Lake Arrowhead, California, 15-18 May 2001.

[61] Y.-H. Low, B.-P. Gan, S. Jain, W. Cai, W.-J. Hsu, S.-Y. Huang, and S.J. Turner. Parallel Discrete-Event Simulation of a Supply-chain in Semiconductor Industry. In *4th High Performance Computing (HPC) Asia 2000*, Beijing, China, 14-17 May 2000.

[62] Y.-H. Low, C.-C. Lim, W. Cai, S.-Y. Huang, W.-J. Hsu, S. Jain, and S.J. Turner. Survey of Languages and Runtime Libraries for Parallel Discrete Event Simulation. *Simulation and Transactions of the Society for Computer Simulation (SCS), Joint Special Issue on Parallel and Distributed Simulation*, 72(3):170–186, March 1999.

[63] B.D. Lubachevsky. Efficient Distributed Event-driven Simulations of Multi-loop Network. *Communications of the ACM*, 32(1):111–123, 1989.

[64] D.G. Luenberger. *Linear and Nonlinear Programming*. Addison–Wesley, second edition, 1984.

[65] V. Madisetti, D. Hardaker, and R.M. Fujimoto. The MIMDIX Operating System for Parallel Simulation. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, pages 65–74, 1992.

[66] V. Madisetti, J. Walrand, and D. Messerschmitt. WOLF: A Rollback Algorithm for Optimistic Distributed Simulation Systems. In *Proceedings of the 1988 Winter Simulation Conference*, pages 296–305, December 1988.

[67] M. Marín. *Discrete-Event Simulation on the Bulk-Synchronous Parallel Model*. PhD thesis, Oxford University, November 1998.

[68] M. Marín. Time Warp on BSP Computers. In *Proceedings of the 12th European Simulation Multiconference*, 1998.

[69] F. Mattern. Efficient Algorithm for Distributed Snapshots and Global Virtual Time Approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993.

[70] W.F. McColl. General purpose parallel computing. In P. Spirakis A. Gibbons, editor, *Lectures on Parallel Compution, Cambridge International Series on Parallel Computation*, volume 4, pages 337–391. Cambridge University Press, 1993.

[71] W.F. McColl. Scalable Computing. In J. Van Leeuwen, editor, *Computer science today, recent trents and developments, Lecture Notes in Computer Science 1000*, volume 12, pages 46–61. Springer-Verlag, 1995.

[72] J. Misra. Distributed Discrete-Event Simulation. *ACM Computing Surveys*, 18:39–65, 1986.

[73] D. Nicol. Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks. *SIGPLAN Not.*, 23(9):124–137, September 1988.

[74] D. Nicol. The Cost of Conservative Synchronization in Parallel Discrete Event Simulations. *Journal of the ACM*, 40:304–333, 1993.

[75] D. Nicol. Parallel Discrete Event Simulation: So who cares? In A. Ferscha, R. Ayani, and C. Tropper, editors, *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, Lockenhaus, Austria, 10 -13 June 1997.

[76] D.M. Nicol and X. Liu. The Dark Side of Risk (What your mother never told you about Time Warp). In *Proceedings of the 11th workshop on Parallel and Distributed Simulation (PADS'97)*, pages 188–195, Lockenhaus, Austria, 10-13 June 1997.

[77] A.C. Palaniswamy, S. Aji, and P.A. Wilsey. An Efficient Implementation of Lazy Reevaluation. In *Proceedings of the 25th Annual Simulation Symposium*, pages 140–146, Los Alamitos, California, USA, April 1992. IEEE Computer Society Press.

[78] Oxford Parallel. The Oxford BSP Toolset and Profiling system (v1.4), 1999. URL : http://www.bsp-worldwide.org/implmnts/oxtool/download.html.

[79] K. Perumalla and R. Fujimoto. Interactive Parallel Simulations with the Jane Framework. To appear in *Special Issue of Future Generation Computer Systems*, 2000.

[80] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Receipes in C: The Art of Scientific Computing*. Cambridge University Press, January 1993.

[81] F. Quaglia and V. Cortellessa. Grain Sensitive Event Scheduling in Time Warp Parallel Discrete Event Simulation. In *14th Workshop on Parallel and Distributed Simulation (PADS 2000)*, pages 173–180, Bologna, Italy, 28-31 May 2000.

[82] P. Reiher and D. Jefferson. Virtual Time Based Dynamic Load Management in the Time Warp Operating System. In *SCS Distributed Simulation*, volume 22, pages 103–111. SCS-The Society for Computer Simulation, Simulation Councils, Inc, January 1990.

[83] P.F. Reynolds, Jr. Comparative Analyses of Parallel Simulation Protocols. In *Proceedings of the 1989 Winter Simulation Conference*, Washington, D.C., USA, December 1989.

[84] G. Ricard and A.K. Agrawala. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Communications of the ACM*, 24(1):9–17, November 1981.

[85] R. Ronngren and M. Liljenstam. On Event Ordering in Parallel Discrete Event Simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation (PADS'99)*, pages 38–45, Atlanta, Georgia, USA, 1-4 May 1999.

[86] R. Schlagenhaft, M. Ruhwandl, C. Sporrer, and H. Bauer. Dynamic Load Balancing of a Multi-cluster Simulator on a Network of Workstations. In *Proceedings of the 9th workshop on Parallel and Distributed Simulation*, pages 175–180, 1995.

[87] G.D. Sharma, R. Radhakrishnan, U.K.V. Rajasekaran, N. Abu-Ghazaleh, and P.A. Wilsey. Time Warp Simulation on Clumps. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation (PADS'99)*, pages 174–181, Atlanta, Georgia, USA, 1-4 May 1999.

[88] D. Skillicorn, J. Hill, and W. McColl. Questions and Answers About BSP. *Journal of Scientific Programming*, 6(3):249–274, 1997.

[89] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. MPI: The Complete Reference. THE MIT Press, 1996.

[90] J.S. Steinman. SPEEDES: Synchronous Parallel Environment for Emulation and Discrete Event Simulation. In *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, pages 95–101, January 1991.

[91] J.S. Steinman. Incremental State Saving in SPEEDES Using C++. In *Proceedings of the 1993 Winter Simulation Conference*, 1993.

[92] S.J. Turner, C.-C. Lim, Y.-H. Low, W. Cai, Hsu W.-J, and S.-Y. Huang. A Methodology for Automating the Parallelization of Manufacturing Simulations. In *12th Workshop on Parallel and Distributed Simulation (PADS'98)*, pages 126–133, Banff, Alberta, Canada, 26-29 May 1998.

[93] L.G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33:103–111, August 1990.

[94] F. Wieland. The Threshold of Event Simultaneity. In *Proceedings of the 11th workshop on Parallel and Distributed Simulation (PADS'97)*, pages 56–59, Lockenhaus, Austria, 10-13 June 1997.

[95] F. Wieland, L. Hawley, A. Feinberg, L. Blume M. Dilorento, P. Reiher, B. Beckman, S. Bellenot P. Hontalas, and D.R. Jefferson. Distributed Combat Simulation and Time Warp: The Model and Its Performance. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 21, pages 14–20. DIMACS Serise in Discrete Mathematics and Theoretical Computer Science, 1989.

[96] L.F. Wilson and D. Nicol. Experiments in Automated Load Balancing. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS'96)*, pages 4–11, Philadelphia, Pennsylvania, USA, May 1996.

[97] Y.-C. Wong, S.-Y. Hwang, and Y.-B. Lin. A Parallelism Analyzer for Conservative Parallel Simulation. *IEEE Transactions on Parallel and Distributed Systems*, 6(6):628–638, 1995.

[98] P. Wonnacott and D. Bruce. The APOSTLE Simulation Language : Granularity Control and Performance Data. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS'96)*, pages 114–123, Philadelphia, Pennsylvania, USA, May 1996.