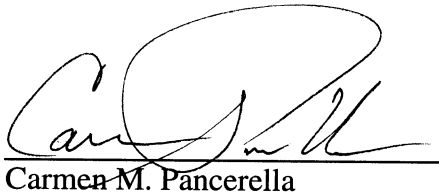
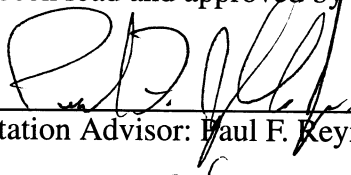
APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy (Computer Science)



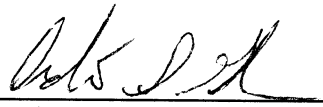
Carmen M. Pancerella

This dissertation has been read and approved by the Examining Committee:



Dissertation Advisor: Paul F. Reynolds, Jr.

Committee Chair: James Cohoon



Committee Member: Andrew S. Grimshaw

Committee Member: Worthy Martin

Curriculum Representative: Ronald Williams

Accepted for the School of Engineering and Applied Science:

Dean Edgar A. Starke, Jr.
School of Engineering and Applied Science

May 1994

Reduction Operations in Parallel Discrete Event Simulations

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Carmen M. Pancerella

May 1994

Abstract

Building on Reynolds's hardware/software framework for parallel discrete event simulation (PDES), we establish a number of novel and best known results based on the use of reduction-based computing to support PDES.

We demonstrate the utility of reduction-based computing to a spectrum of well-known PDES synchronization protocols, such as conservative techniques and Time Warp. We enhance the hardware portion of this framework at three levels: 1) we define a virtual computation model, 2) we develop a functional design, and 3) we present a detailed implementation of this design. Each of the preceding steps is based on correctness criteria we establish here. We develop novel algorithms for performing reduction-based message acknowledgments. We prove the correctness of one of them, a single phase acknowledgment algorithm that takes advantage of the existence of global virtual time. Finally, we introduce *target-specific reductions*, a very promising strategy for disseminating near-perfect state information in PDES's. A target-specific reduction is one where each logical process receives synchronization information (reduced values) only from those logical processes on which it is logically dependent. We demonstrate that the computation of target-specific values can have a sub-quadratic sequential time complexity. Supporting empirical results clearly demonstrate that target-specific reductions will provide significant time and space savings in PDES's.

Acknowledgments

First I thank my advisor Paul Reynolds for many things. His initial framework was a gold mine of exciting research, and I'm happy to have had the chance to develop this framework here. Second I thank Paul for pushing me when I needed a push and pulling me when I needed to be pulled. Finally, I thank Paul for his advice, even when I didn't take it.

I thank my committee members Andrew Grimshaw, Worthy Martin, Jim Cohoon, and Ron Williams for their effort and guidance during this entire research project. They have made many suggestions which make this work stronger. I extend a special thanks to Jim Cohoon for his helpful advice throughout my graduate career; he has been always been a strong supporter of my work, and I am grateful to him. I also thank Jim Ortega, the department chair, and the entire CS faculty, for supporting my graduation "under the wire".

The results that appear in Chapter 5 will never show the amount of time spent nor the number of dead ends travelled. I thank Jeff Salowe, Gabe Robins, and Jim Cohoon for the time I spent with them working on the theory behind this problem. Also, I discussed this problem with several students: Kevin Treu, Phil Dickens, Ray Wagner, Bronis de Supinski, Sudhir Srinivasan, and Craig Williams were all generous enough to take time away from their research to brainstorm about this problem. I thank Sudhir also for proof-reading the algorithms and proofs in Chapter 4 and for his help when I implemented the algorithms on our prototype hardware. His comments were invaluable. Finally, I thank Ken Ruggaber, whose simulation code I used to implement the acknowledgment algorithms. His programming efforts on the PRN made my life much easier.

I thank Mark Smith, Ray Lubinsky, Gina Bull, and Ann Bailey for their excellence in systems administration. I thank Ginny Hilton, Carolyn Duprey, Kim Gregg, Barbara Graves, Pam Evans, Chris Byars, Brenda Lynch, and Tammy Ramsey for all the paperwork they've prepared on my behalf. I especially thank Tammy for working with me to graduate on time.

I treasure the friendships that I have made while in graduate school. Bryan Catron and Paula Gabbert Catron, Kevin and Julie Treu, Pat and Cindy Heck, Ray Wagner, and Phil Dickens are my old-timer friends, in whose footsteps I followed. I cherish the memories of cookouts, softball championships, pinochle games, and good times. I thank Rachel Lorey and Karen LeMaire, both roommates and close friends, for sharing cooking responsibilities and for being supportive during difficult times. Finally, I thank my friends and drinking buddies Mark and Ann Bailey, Ed Loyot, Mike Krell, and especially Mike Delong for the many happy hours, lunches, and griping sessions that we shared.

I thank Tim Strayer, my best friend, for many things: the pleasant diversions from my graduate studies, the dinners he cooked for me while I studied for comps, the innumerable pep talks that he gave me along the way, the strong shoulders that I have cried on many times, the encouraging and entertaining long-distance email that he sent me during the final days of my dissertation, and his continued love and support. I look forward with great anticipation to spending time with him now that we're both out of grad school.

I personally thank Amy Fellin Caputo, Donna Makara Dudeck, Megan Evans, Karen Yeager Gorel and their families for the encouraging words they've given me through high school, college, and graduate school. I am blessed with their friendships. I especially thank Amy and Nick for the hospitality of the Caputo Country Inn in Richmond, Virginia, a home away from home.

I am blessed with a wonderful brother and a sister. I thank Tony and his wife Mary Beth for being supportive of my ambitions. I thank Chris for her love and friendship; the eight years between us have disappeared, and I feel her presence with me everyday. I'll never forgive myself for missing her graduation from Penn State to finish my dissertation.

My parents are the people who have taught me the importance of education. This is one of the most valuable lessons I have learned from them. I thank them for their encouragement while I pursued my Ph.D. and for always seeing the light at the end of this tunnel, even when I couldn't. I'll never be more than a heartbeat away from them.

This work was supported in part by NSF Grant #CCR-9108448, JPL Contract #957721, NASA Grant NAG-1-1529, and NSF IIP Grant CDA-89-22545-03.

Dedication

I dedicate this dissertation to my parents, Barbara and Mauro Pancerella, whose love and support are immeasurable. My love and gratitude are immeasurable.

Table of Contents

List of Figures and Tables

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Chapter 7

List of Symbols

$\eta_i(t)$	next event time of LP_i at real time t
$\eta'(t)$	minimum next event time of all LP's at real time t
$\upsilon_i(t)$	smallest unreceived message time of LP_i at real time t
$\upsilon'(t)$	minimum unreceived message time of all LP's at real time t
$\sigma_i(t)$	logical clock of LP_i as observed by AP_i at real time t
$\sigma'(t)$	smallest logical clock of all LP's at real time t
$\bar{\sigma}_i(t)$	logical clock of LP_i at HP_i at real time t
$\rho_i(t)$	*T*-value of primary acknowledgment by LP_i at real time t
$\rho'(t)$	smallest primary acknowledgment of all LP's at real time t
$\tau_i(t)$	*T*-value of secondary acknowledgment by LP_i at real time t
$\tau'(t)$	smallest secondary acknowledgment of all LP's at real time t
$\mu_i(t)$	minimum timestamp of LP_i that can be read by PRN at real time t; $\mu_i(t) = \text{MIN}(\sigma_i(t), \upsilon_i(t))$

1 Introduction

Parallel discrete event simulation (PDES), or parallel simulation, is the execution of a discrete event simulation on a parallel computer. It is an atypical parallel computing problem: the computation is asynchronous yet there can be a significant amount of inter-dependence among processes. This makes parallel simulation a challenging problem. During his keynote address at the *Seventh Workshop on Parallel and Distributed Simulation (May 1993)*, Mani Chandy identified two key research contributions that parallel discrete event simulation (PDES) [FUJI90] has made and is making to parallel computing: 1) the development of techniques for efficient asynchronous computation and 2) the exploration of reduction operations (binary, associative operations). The majority of the research in parallel simulation has been the development of synchronization protocols, and this research is concentrated in the first research area. The only significant research in the computation of reduction operations has been the development of algorithms for the computation of global virtual time (GVT) in a Time Warp parallel simulation. The utility of reduction operations has been greatly overlooked until now. The results presented in this thesis make important contributions to the efficient computation of reduction operations in support of parallel simulation.

A novel framework for parallel simulation was presented by Reynolds [REYN91]. This framework is a software/hardware ensemble for the efficient computation of reductions and the dissemination of reduced values in support of parallel simulations. There are three components to this framework: 1) *reduced values* which characterize the state of a parallel simulation, 2) *framework hardware* to compute and disseminate the reduced

values, and 3) *correct algorithms* which execute on the hardware and use the reduced values to support synchronization in a parallel simulation.

In this thesis we advance this framework in four major areas: 1) a computation model for computing reductions in a parallel simulation, 2) a novel implementation of the hardware portion of the framework, 3) novel, verified message acknowledgment algorithms which execute on the framework hardware and are used to maintain a minimum outstanding message time necessary in the computation of critical synchronization values, and 4) a new class of reductions, namely target-specific reductions. We discuss motivations for each.

1.1. Motivation and Objectives of this Research

When a discrete event simulation is partitioned for parallel execution, *logical processes* (LP's) model physical processes from the corresponding physical system. Each logical process has an *events list*, a list of events to be executed, and a *local clock*, indicating how far the simulation has progressed at that LP. Logical processes communicate through the use of timestamped *event messages*, where an event message indicates a time that an event will be scheduled in the events list of the receiving logical process. Logical processes are largely asynchronous.

A result of this model of parallel discrete event simulation is a difficult synchronization problem: each LP must determine when it is permissible to advance its logical clock. If an LP advances its logical clock too far ahead of other LP's, it may receive an event message with a timestamp in its logical past, i.e., less than its local clock. A *causality error* can occur when an LP receives a message with a timestamp in its logical past. A causality error can led to incorrect results in the parallel simulation. Parallel discrete event simulation *synchronization protocols* (sometimes called *synchronization algorithms*)

are mechanisms of guaranteeing the correct execution of parallel simulations. Broadly these protocols fall into two major classes, though there is a wide range of synchronization protocols. One class is protocols that are *accurate*, *non-aggressive*, and *without risk* (using terminology developed by Reynolds [REYN88]); these protocols are commonly referred to as *conservative* protocols. Protocols that are non-aggressive and without risk do not allow an LP to process an event with timestamp *t*, if it is possible that it will receive an event message with timestamp *r*, $r < t$, at some point in the future. The second major class is protocols that are *accurate*, *aggressive*, and *with risk*; these protocols are commonly called *optimistic* protocols, and Time Warp [JEFF85] is the most common of the optimistic protocols. Protocols that are aggressive and with risk allow an LP to process any event in its event list, and any causality errors that result from aggressive processing are corrected through a *rollback* mechanism. (See Chapter 2 for a more complete discussion of parallel simulation synchronization protocols.)

The need for special-purpose hardware to support parallel simulations is well established. The computation requirements for parallel simulations continue to grow. The simulation of large communication networks and battlefield scenarios, for example, both require a significant amount of computation time (sometimes weeks or more). In addition, simulation programs, especially those employing aggressive processing, often utilize a large amount of memory. The importance of research in the area of hardware support for PDES has been recognized in a recent article on the state of the art in parallel simulation [NIFU92].

Most of the research in hardware support for parallel simulations has been in support of state saving and rollback in Time Warp simulations. The need for efficient synchronization in parallel simulations has also been recognized, yet only one known research effort [LUBA88] has considered hardware to support the efficient computation of

reductions for synchronizing PDES logical processes. Lubachevsky's effort is quite limited in scope, supporting of only the parallel simulation protocol he proposes.

Some ideal features of framework hardware to support synchronization in parallel simulations are:

- Speed — The hardware must be designed to compute and disseminate synchronization information very rapidly and with little overhead.
- Scalability — The hardware must be scalable.
- Adaptability — The hardware should be adaptable to current and future parallel computers. Also, the framework should be designed to adapt to current technology with ease.
- Generality — The hardware must be able to support a spectrum of parallel simulation synchronization protocols [REYN88].
- Low cost — The hardware to support parallel simulation synchronization protocols should not be expensive.

One of the primary motivations of our effort is the efficient computation of global virtual time in an optimistic PDES protocol. Global virtual time is comprised of two reduced values: the smallest logical clock in the system and the smallest time of a message that has been sent but not yet received and processed by its intended receiver. In support of this second component of GVT, it is necessary to keep track of outstanding messages or to acknowledge messages once they have been received. Our solution to this problem is to use the high-speed network, which computes reductions, to also acknowledge messages.

It is critical that the computation of reduced values proceed asynchronously to the execution of the simulation and that neither the computation of the reductions nor the simulation is blocked. Also, it is critical that the algorithms which support synchronization and event message acknowledgments be correct. We have based our framework on formal correctness criteria and have derived correctness proofs for message acknowledgment algorithms.

The framework as proposed by Reynolds [REYN91] includes the computation of globally reduced values. We believe that in some cases globally reduced information is not sufficient for logical processes in a PDES to make the necessary event processing decisions. Globally reduced values only capture information about one logical processor (e.g. the smallest logical clock in the system) or all processors (e.g. the total number of outstanding messages). The dissemination of global information means that each logical process receives information that is derived from the whole group's inputs. In many PDES's, each logical process is only affected by the processing of a subset of the logical processes. By providing target-specific reductions, where each logical process receives reduced information from only those logical processes that have an impact on its performance (e.g. all predecessors in a precedence graph), we expect significant performance gains (measured in execution time, memory usage, or a combination of both) for parallel simulations. A network to compute and disseminate target-specific reductions can provide near-perfect state information to parallel simulation synchronization protocols. This is the motivation for exploring the cost of computing and disseminating target-specific reductions.

1.2. Contributions

This thesis contributes to the field of parallel discrete event simulation by

- the development of a hardware/software framework to support the wide range of existing parallel simulation synchronization protocols and to influence the development of new parallel simulation synchronization protocols
- the first significant research on the importance of reductions in parallel simulations
- the exploration of several event message acknowledgment algorithms which use a reduction network and remove all related load from a host communication network
- the introduction of target-specific reductions to parallel discrete event simulations

- the exploration of the cost of computing target-specific reductions, in general, trade-offs in time and space complexities
- the best known results on the cost of computing target-specific reductions
- the utility of the target-specific reductions to both conservative and optimistic parallel simulation synchronization protocols.

1.3. Thesis Overview

This thesis is organized as follows.

Chapter 2 offers a review of parallel discrete event simulation, reduction operations, architecture support for computing reduced values, and related work.

Chapter 3 describes the computation model of our framework for parallel discrete event simulation and its correctness.

Chapter 4 presents several algorithms for the acknowledgment of messages in a reduction network and a proof of correctness of one of the more promising ones. Also, performance issues of algorithms are discussed.

Chapter 5 explores time and space complexity issues related to target-specific reductions.

Chapter 6 presents experimental results demonstrating the need for the rapid dissemination of target-specific synchronization information in both conservative and optimistic parallel discrete event simulations.

Chapter 7 offers conclusions drawn from this research, and offers avenues for future work.

2 Background

Our research touches on the areas of parallel discrete event simulation, reduction operations, networks which compute and disseminate the results of reduction operations, and directed graph theory. We review the relevant literature and introduce terminology in each of these areas.

Parallel discrete event simulation is the application which we support with the computation of efficient reduction operations. Fujimoto has written an excellent survey of PDES research prior to 1990 [FUJI90], and Nicol and Fujimoto have published current PDES research topics since 1990 [NIFU92]. We present background research and related work in PDES in Section 2.1.

Our research applies the efficient computation of reduction operations in hardware to parallel simulation synchronization protocols. In Section 2.2. we discuss reduction operations, and in Section 2.3. we present other networks which compute and disseminate reduction results, including a discussion of sorting networks in Section 2.3.5. We include the discussion of sorting networks because our best cost of computing target-specific reductions (See Chapter 5) requires a sort, and we believe a solution to designing an efficient target-specific parallel reduction network will likely have a sorting network as a component.

The computation of target-specific reductions only supports parallel simulations where the set of potential communicants is known prior to the execution of the simulation. We review directed graph theory in Section 2.4. since many of these definitions are necessary to the understanding of the target-specific problem and its solutions.

2.1. Parallel Discrete Event Simulation

Discrete event simulation is a common technique for modelling large systems, for example, military applications, logic circuits, traffic systems, and telecommunication networks. A sequential discrete event simulation may require the execution of millions of events and as a result can take hours or days to complete. *Parallel discrete event simulation* (*PDES*) is the process of executing a discrete event simulation on multiple processors with the goal of reducing the finishing time of the equivalent sequential simulation.

Discrete event simulations often contain a high degree of potential parallelism; however, in practice, they can be difficult to parallelize. Parallel simulation is a hard problem because its very nature is *data-dependent* and *asynchronous*. As we will explain in subsequent sections, most PDES's deal with asynchronous systems where events are not synchronized by a global clock but rather are dependent on other events in the system. Synchronization of logical processes in a PDES has been the major focal point of PDES research.

In the next section, we present a model of parallel simulation and introduce the terminology necessary for describing our research contributions to PDES. We discuss common synchronization protocols for implementing parallel simulations. Our framework for PDES, described in Chapter 3, will support each of the protocols mentioned in this chapter. Finally, we discuss hardware support for parallel simulations.

2.1.1. Model of PDES

A general model of a PDES has been described by Misra [MISR86]. A PDES consists of a set of *logical processes* (*LP's*) that model *physical processes* in a system. All interactions among physical processes are modeled by *event messages*, or *events*, sent among LP's. Each message contains a timestamp indicating the logical time at which a

scheduled event will occur. Each LP maintains a relative counter, its *logical clock*, that denotes how far that LP has progressed. It is very likely that logical clocks among LP's logical clocks will differ, and this asynchronous property of parallel simulation can introduce a synchronization problem, the central problem in PDES since its inception. Each LP maintains a data structure of events to be processed, its *event list*, and its logical clock is advanced to the time of the event with the next logical time. Knowing when it is *safe* to advance the logical clock of an LP is a difficult problem.

A *causality error* in a PDES occurs when event B depends on event A, and event B is executed before event A. Observe the instance in Figure 2.1. In Figure 2.1(a), LP_1 begins to process its earliest event at logical time 30 and LP_2 processes its event at logical time 60. In Figure 2.1(b), LP_1 has finished the processing of its event at time 30 which results in a timestamped message sent to LP_2. The receipt of this message causes an event to be scheduled at LP_2 with timestamp 40. In this example a causality error has occurred since LP_2 has executed events out of timestamp order. A parallel simulation will be correct if and only if each LP ultimately processes events equivalent to a nondecreasing timestamp order. Adherence to this *local causality constraint* is sufficient, though not always necessary, to guarantee the absence of causality errors [FUJI90]. However, due to both the asynchronous

nature of LP's logical clocks and communication delays among LP's, there is no way of guaranteeing that messages received by LP_i occur in a specific order.

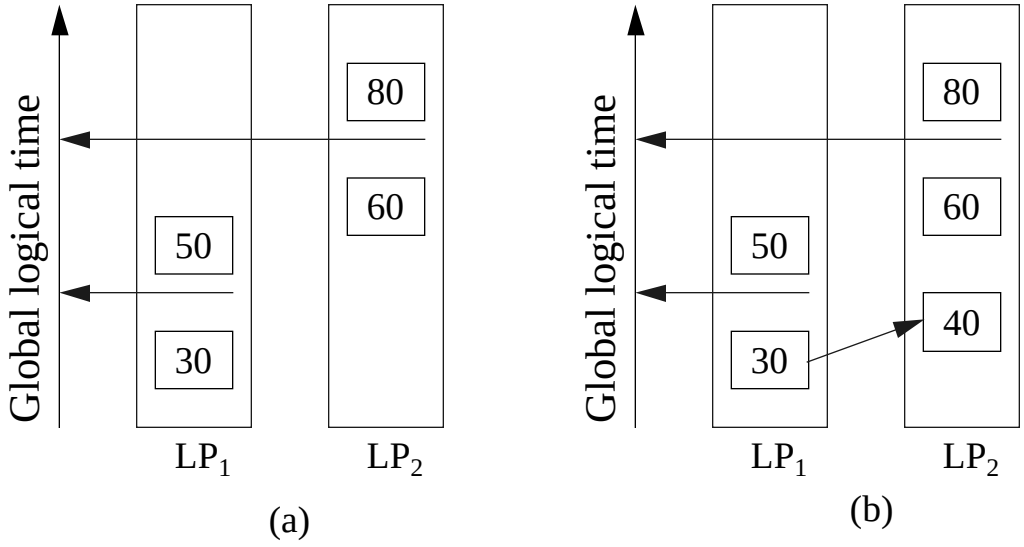


Figure 2.1 Example of a Causality Error in a Parallel Simulation.

It would appear that parallel simulation is a natural candidate for parallel processing. However, unlike the many applications that exhibit good parallelizing properties, PDES's typically have irregular data-dependent properties. The LP's operate asynchronously, and global time does not easily map to a parallel computer. The major challenges facing researchers in parallel simulation are to maximize the parallelism that can be obtained in a PDES and, at the same time, to synchronize logical processes such that each LP processes events in non-decreasing timestamp order.

Most of the research in PDES has been directed at solving the synchronization problem such as the one depicted in Figure 2.1. Many parallel simulation synchronization protocols have been proposed; Fujimoto [FUJI90] gives a good survey of this literature. Each of the proposed protocols performs well under certain conditions and poorly under

others. The performance of one protocol over another depends on the multiprocessor architecture and the application being simulated. Traditionally, researchers have categorized PDES synchronization protocols as either conservative or optimistic. This classification is too restrictive; Reynolds has developed a set of design variables that define a spectrum of options for parallel simulation synchronization protocols [REYN88]. Although nine design variables are defined, the four that are most relevant to our research are aggressiveness, accuracy, risk and synchrony.

- *Aggressiveness* involves relaxing the requirement that event messages are processed in a strict monotonic order with respect to message times. In other words, an event can be processed by an LP without the guarantee of freedom from causality errors. An LP that adheres to a *non-aggressive* policy will only process a message if no message that logically precedes it can arrive at that LP.
- *Accuracy* requires that events within LP's are *ultimately* processed in monotonically non-decreasing order. In most cases, this requires that the set of all final states for a given parallel simulation be equivalent to the set of all final states for a sequential counterpart. Accuracy requires that all events have the *effect* of having been processed in a monotonic sequence; aggressiveness does not address this issue and is independent of accuracy.
- *Risk* is the design variable that allows an LP to send messages that have been processed based on aggressive or inaccurate processing assumptions. A protocol that employs aggressiveness without risk guarantees that all rollbacks are strictly local to that LP. A *risk message* [REYN88] is a message that is the product of actions taken based on incomplete (conditional; see [CHMI87]) knowledge or as a result of processing that leads to the transmission of out-of-order messages. An LP is *at risk* if there exists a risk message at the head of at least one of its input queues or if at least one of its input queues is empty. Aggressive processing coupled with risk can create or pass along risk messages.
- *Synchrony* describes the amount of synchronization among LP's. Parallel simulation synchronization protocols can be asynchronous, loosely synchronous, or synchronous.

It is important to remember that these design variables are orthogonal; hence, there are several levels of synchronization protocols and not just the conservative/optimistic dichotomy often found in the literature. We present a brief overview of parallel simulation

synchronization protocols categorized with respect to aggressiveness, inaccuracy, risk and synchrony. The names of the categories are common in the literature.

2.1.2. Conservative PDES Synchronization Protocols

Chandy and Misra [CHMI79] and Bryant [BRYA77] performed pioneering work in this area independently. This class of protocols will never permit an LP to do incorrect work; hence, each LP cannot proceed until it is guaranteed not to receive a message in its logical past. In a conservative protocol, an LP will block until determines that it can safely process an event. These protocols are accurate, non-aggressive, asynchronous, and without risk.

The most significant disadvantage of conservative PDES synchronization protocols is that the potential parallelism is not always fully exploited due to *artificial blocking* [REYN82]. Artificial blocking occurs when LP_i is blocked and waiting for a message from an LP whose logical clock has already exceeded the message time of LP_i's pending message. This artificial blocking occurs because LP_i has insufficient information to determine that it can safely advance its logical clock.

Blocking introduces the potential for deadlock. Deadlock occurs when a parallel simulation has cycles where each LP in the cycle is blocked and waiting for a message from another LP in the same cycle. There are many research efforts in deadlock handling. (See [CHMI79], [MISR86], [PEWM79], [PEWM79b], [REYN82], [NICO84], [CHMI81], [YUGD91], [DEGY91], and [LITR90] for different approaches to dealing with the deadlock problem.) Either deadlock avoidance or deadlock detection and recovery is an overhead for the simulation.

The performance of conservative PDES synchronization protocols just described is greatly affected by the amount of *lookahead* that is possible in the specific application.

Lookahead is the ability of an LP to predict its future behavior; in other words, if an LP has received event messages up to logical time *t*, it can predict that any message it sends in the future will have a timestamp of at least ($t + \varepsilon$) for some $\varepsilon > 0$. Researchers have shown that minimum event processing times and lookahead values can produce significant performance improvements in a non-aggressive PDES synchronization protocol. (See [NIRE84], [FUJI87], [FUJI88], [REMM88], [NICO88], [NICO88b], [LILA89], [WALA89], and [FEKL92b] for performance results on the effects of lookahead values.)

Conservative protocols require the communication topology to be static and known a priori. Furthermore, these protocols do not efficiently support fully-connected communication graphs. This usually requires excessive overheads such as broadcast communication necessary to determine when it is safe to proceed. In general, conservative protocols perform well on simulations with sparse topologies and good lookahead properties.

2.1.3. Optimistic PDES Synchronization Protocols

The most common optimistic protocol is Time Warp [JEFF85]. Time Warp employs maximal aggressiveness: each LP executes without regard to whether there are synchronization conflicts (i.e. potential causality errors) with other LP's. A protocol in this class may do incorrect processing, and at some point, the incorrect results must be undone, and the work redone correctly. These protocols are accurate, aggressive, asynchronous, and without risk.

Under the Time Warp protocol, an LP processes messages from its input queue(s) in monotonically non-decreasing timestamp order until it exhausts the queue(s) [JESO85]; hence, it never blocks or waits until it can safely process the next message. Time Warp guarantees that a parallel simulation cannot deadlock if and only if all LP's process events

unconditionally. Since events can be processed out-of-order, Time Warp employs state saving and rollback as methods for repairing potential incorrect event sequencing.

Each LP_i executing a Time Warp simulation maintains its own *local virtual time* (*LVT*), which acts as the simulation clock for LP_i. Each LP will save its state periodically. When a causality error is detected, the LP restores a previous state that was saved before the error occurred. State restoration, or *rollback*, involves resetting an LP's local clock to a time in the logical past. After an LP has completed a rollback, it can begin processing forward in time. Unfortunately, due to the sending of event messages, a causality error could be spread to other LP's. To ensure accuracy, other LP's must be notified of erroneous messages.

If LP_i discovers a causality error at logical time *t*, then any messages it has sent with a timestamp greater than *t* are potentially in error. Time Warp employs *antimessages* as a way to *cancel* event messages: each antimessage has the exact same content and format as its corresponding event message, but it has a different *sign*. An antimessage may cause the receiving LP to roll back, which may in turn cause more antimessages to be sent to other LP's. Event messages can be cancelled with either *aggressive cancellation*, where a rollback message causes the immediate cancellation of all event messages sent with timestamps greater than the timestamp on the rollback message, or *lazy cancellation*, where events are cancelled only if during subsequent forward processing they are determined to be in error. One rollback can cause a *cascade* of rollbacks. An LP, however, can never roll back past the *global virtual time* (*GVT*) [JEFF85].

GVT is maintained across all LP's: this is a simulation-wide safe time. GVT is the guaranteed time for which all events with timestamps less than or equal to it have been processed accurately. In other words, at any real time *r*, GVT(*r*) is defined to be the minimum of all local clocks and of the timestamps of all transient messages [JEFF85]. GVT

limits the amount of saved state information that an LP must have in memory at any time; it also prevents rollback to the beginning of simulation time (unless GVT is equal to this time). Also, GVT is used to show a guarantee of progress in a Time Warp simulation [JESO85]. Furthermore, GVT can be used in termination detection, crash recovery, and input and output handling.

There are three major challenges in optimistic protocols. First, the cost of periodic state saving can be very high. The use of special-purpose hardware, the rollback chip [FUTG92], can almost eliminate this cost. The problem with the rollback chip is that its memory capacity is limited.

Second, optimistic protocols use more memory than sequential simulations due to state saving and aggressive processing. *Fossil collection*, or garbage collection, is necessary for an efficient implementation of Time Warp. Fossil collection involves destroying state information that is older than GVT as a method of freeing up available memory for currently saved states. Since state saving is such an integral part of the Time Warp protocol, efficient memory management is critical to its performance. The cancelback protocol [JEFF90], proposed by Jefferson, provides optimal storage management. Performance studies [DAFU93] have shown that the GVT maintenance scheme is critical to this memory management protocol. There are many proposed GVT computation schemes. Representative ones include [JEFF85], [JESO85], [SAMA85], [LILA89b], [BELL90], [COKE91], and [TOGA93].

The third main problem with a rollback-based simulation is that the process of rolling back computation can degrade performance. A *cascading rollback* is a “chain reaction” of rollbacks where the number of LP’s increases without bound [LUSW89]. *Echoing* is a pattern of self-fueled rollbacks whose amplitude increases without bound [LUSW89]. Turner and Xu [TUXU92] have reported cascading rollbacks and echoing

which significantly degrade performance in their telephone switching network simulation. Mitra and Mitrani [MIMI84] and Lubachevsky [LUSW89] have developed models to show that echoing can occur.

Many experimental results have been published on the performance of the Time Warp parallel simulation synchronization protocol [BERR86], [JEBH85], [GILM88], [LOCU88]. It has been demonstrated that Time Warp is a robust protocol across a wide range of workload parameters [FUJI90b] and is not as sensitive to lookahead as the conservative protocols. (See [GAFN88], [FUJI89b], [LILA89c], [LILA89d], [LILA89e], [LILA90], and [LILA90b] for additional performance studies on Time Warp.)

The major advantage of optimistic PDES synchronization protocols is that more parallelism can be extracted from some applications than with other PDES synchronization protocols. Time Warp does not suffer from artificial blocking. This advantage, however, comes at the cost of extra memory requirements and a more complex mechanism for state saving and rollback. Fujimoto found that as the size of the state increased by a modest size of 2000 bytes, the degradation of performance was reported to be 50% [FUJI89b]. The degradation of performance in this case is a result of both state saving overhead and the increased frequency of fossil collection.

2.1.4. Iterative PDES Synchronization Protocols

Conservative and optimistic protocols are both asynchronous protocols. Several researchers have introduced PDES synchronization protocols which are *loosely synchronous* (See [FOJO88]), such that LP's process asynchronously and synchronize periodically at barriers. These protocols proceed iteratively and synchronize at the end of each of three phases. Phase one requires LP's to determine the event with the smallest timestamp in the system. In phase two LP's determine which events are safe to process.

Finally, in phase three LP's process all events that have been determined to be safe. The primary difference between any of these synchronization protocols is the method used to determine which events can safely be processed concurrently. Iterative protocols are accurate, aggressive or non-aggressive, with or without risk, and loosely synchronous.

Examples of iterative PDES synchronization protocols include the Bounded Lag protocol [LUBA88a], the Moving Time Window protocol [SOBW88], Chandy and Sherman's protocol for converting conditional events into unconditional events [CHSH89], an iterative algorithm based on the distance between objects [AYAN89], Nicol's iterative algorithm [NICO90] [NICO91], and the Global Windowing Algorithm [DICK93].

2.1.5. Hardware Support for PDES

The need for special-purpose hardware to support PDES is well established. In a recent survey on the state-of-the-art in parallel simulation, Nicol and Fujimoto recognize hardware support as one of six important areas of future research [NIFU92].

The use of special-purpose hardware to improve the performance of simulation programs is not novel. Logic simulation engines have been constructed that yield significant speedups [FRWW84]. Lubachevsky suggests using a special-purpose network to broadcast a minimum event time in his Bounded Lag protocol [LUBA88]. This network is a binary tree implemented in hardware in order to support synchronization barriers and to compute and broadcast a minimum next event time. His control synchronization network is presented strictly in support of the bounded lag protocol; nonetheless, this has served as a motivating factor in our approach.

Hardware enhancements for Time Warp have been prevalent in the research; for example, Livny and Manber suggested using token rings for disseminating GVT [LIMA85]. One trend in hardware support for Time Warp is to design a high performance,

discrete event simulation engine. Fujimoto initially targeted the Virtual Time Machine [FUJI89] as hardware support for discrete event simulation, but this machine is now intended to utilize an aggressive style of execution in a general purpose parallel computer. Fujimoto, *et. al.*, developed the *rollback chip* [FUTG92] as a hardware enhancement to a Time Warp engine. The rollback chip is a memory management unit that facilitates the state saving and restoration that is inherent in aggressive protocols such as Time Warp. As reported in a study by Buzzell and Robb [BURO90], the chip has excellent performance capabilities.

Filoque, *et. al.*, [FIGP91] proposed the use of a processor network with programmable logic for efficient global computations, such as the computation of GVT in a Time Warp simulation. This hardware is not a single network like we introduce in Chapter 3; it is, however, a distributed system of *sockets*, one per processor. The reprogrammable sockets are connected in a pipelined ring, forming the computation engine. A token is inserted into the ring by a designated control socket. It travels around the ring, performing partial computations at each socket. When the token returns to the controller, the global computation is complete. Therefore, their proposed hardware performs global computations in $O(n)$ time whereas, our synchronization network computes reductions in $O(\log n)$ time. (See Chapter 3.) Furthermore, the proposed synchronization algorithms for computing GVT in Filoque's network [FIGP91] rely on the host communication network for message acknowledgments and our framework uses the framework hardware for this purpose. (See Chapter 4.) The goals of both approaches are similar, but our framework is more efficient, more flexible, and more scalable, as will be shown throughout this thesis.

2.2. Reduction Operations

We begin by defining a *reduction*. A reduction is a binary, associative operation $\oplus$ computed across n inputs. Examples of reductions include minimums, maximums,

summations, logical AND's, and logical OR's. In the next sections we discuss types of reduction computations and hardware efforts to compute reductions.

2.2.1. Parallel Prefix Operations

The *all-prefix-sums* operation takes a binary, associative operator $\oplus$, and an ordered set of *n* elements $[a_0, a_1, \ldots, a_{n-1}]$, and returns the ordered set $[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \ldots a_{n-1})]$. An array all-prefix-sums operation is called a *scan*. A reduction operation, therefore, generates the final element of the scan. A *segmented scan* operation allows an array to be operated on by a scan such that it is broken into segments with flags that allow the scan to start again at each segment boundary. This requires two arrays, *a*, the data array, and *f*, the flag array. The segmented scan can be defined by the following recurrence:

$x_i = a_0$, if $i = 0$

$x_i = a_i$, if $f_i = 1$, $0 < i < n$

$x_i = (x_{i-1} \oplus a_i)$, if $f_i = 0$, $0 < i < n$

These formal definitions and applications of scans have been published by Blelloch [BLEL90], although scans were first introduced for the language APL [IVER62], and segmented scans were first suggested by Schwartz [SCHW80].

Blelloch [BLEL89] proposed a tree-structured hardware implementation of parallel prefix operations with *O(n)* components and *O(log n)* time complexity. The computation of parallel prefix reductions is a subset of our problem of computing and disseminating target-specific synchronization information, as discussed in Chapters 5 and 6.

2.2.2. Minima of Interval Computation

A problem related to the segmented scan operation is the minima of intervals operation [GIRY88]. This operation takes a vector *a* of integers and a vector *int*, $int_i = [l_i..r_i]$, of intervals, both of size *n*, and returns the array *x*, defined as follows:

$$x_i = MIN(a_k), k \in int_i$$

With the same complexity as the parallel prefix problem, the minima of intervals problem can be solved in *O(log n)* with *O(n)* processors using a balanced binary tree method.

2.2.3. Broadcasting with Selective Reduction

Lindon and Akl [LIAK93] introduced the Broadcasting with Selective Reduction (BSR) as an extension to the CRCW PRAM model of parallel computation. This operation permits a broadcast operation to shared memory, such that a binary, associative operator $\oplus$ is applied to each data item whose tag satisfies the condition *tag* σ *limit*, where σ is the *selective operation*. Linden and Akl show an optimal implementation for the BSR operation of size *O(n log n)* and time complexity *O(log n)*, where *n* is the total number of processors and memory locations. This implementation combines sorting circuits, parallel prefix circuits, and merging circuits. The sorting circuit is assumed to be an AKS sorting circuit [AJKS83] which has optimal space and time complexities but an impractically large multiplicative constant.

The Multiple Criteria *n*-processor BSR [LIAK93] allows multiple selection of the data items to be reduced. Akl and Stojmenovic [AKST94] present a $O(n^2)$ switch solution to this problem with time complexity of *O(log n)* and also state that it is an open problem if there is a Multiple Criteria *n*-processor BSR implementation that requires a number of switches asymptotically smaller than $O(n^2)$. The single criteria *n*-processor BSR implementation is restricted to solving problems whose inputs obey a strict linear order. An

optimal solution to the Multiple Criteria *n*-processor BSR problem is directly applicable to the target-specific dissemination problem.

2.3. Related Architectures

Using a separate synchronization network for improving system performance is not a new idea. The IBM RP3 [PFBG85] was designed as a shared memory multiprocessor that houses both a combining network for synchronization traffic and a low latency network for regular message traffic. In this section we discuss related hardware efforts for barrier synchronization, computing reductions and sorting.

2.3.1. Hardware for Barrier Synchronization

Several researchers have proposed the use of hardware to implement barrier synchronization. Hoshino [HOSH85] has an efficient barrier synchronization in the PAX computer. Stone [STON90] suggests the use of global busses to compute maximum values and to implement fetch-and-increment. The hardware that we propose, on the other hand, provides support for a larger class of algorithms than barrier synchronization algorithms.

2.3.2. Intel iPSC/2

Many parallel architectures provide for *global* binary, associative operations across all processors. Global operations on the Intel iPSC/2 [INTE89] are provided for arithmetic and logical operations. There is no separate network to support this computation. All computation is performed on the host processors and all communication is done in the data network.

2.3.3. Finite Element Machine

The Finite Element Machine (FEM) [JOSC79], a NASA prototype, utilizes a binary tree-structured max/summation network to perform the *global* sum and maximum

calculations necessary to support structural analysis algorithms. The sum and max calculations in the FEM are calculated alternately without processor synchronization. Unlike the FEM, our hardware is designed to operate on a a set of input and output registers which are treated as a single state vector, whereas the FEM uses a single input and a single output register. (See Chapter 3.)

2.3.4. Thinking Machines CM-5 Supercomputer

The Thinking Machines CM-5 [THIN92] contains two separate networks for different types of communication and synchronization: the data network is the primary message-passing network in the machine and the control network provides hardware support for common *cooperative operations*. The CM-5 control network [LEAD92] supports "soft" barrier synchronization, global arithmetic and logical reduction operations, parallel prefix and parallel suffix operations, and segmented parallel prefix operations. As in the iPSC/2, the reduction operations the CM-5 require the complete synchronization of all processors. All processors must call global operation functions with a contributed value, and a global operation blocks until all processors enter it. Our framework hardware, on the other hand, computes and disseminates globally reduced values on state vectors *without* the coordination of the host processors; the reduction operations on the PRN are performed continuously and asynchronously. Furthermore, the hardware design employs dedicated processors to manage the high-speed I/O between from the reduction network and itself.

2.3.5. Sorting Networks

There is a multitude of literature on parallel sorting and sorting networks. We limit our discussion to a class of practical sorting networks and a theoretically optimal sorting network. Batcher's two sorting networks [BATC68], the odd-even merge network and the bitonic sorting network, are both based on parallel merge sort and have similar properties.

Both of these networks have width *O(n)* and depth $O(\log^2 n)$, so the time complexity to sort *n* items is $O(\log^2 n)$. The AKS sorting network [AJKS83] is an optimal sorting network with width *O(n)* and depth *O(log n)*, but the associated constants are too large for this network to be considered practical.

2.4. Directed Graph Theory and Terminology

Most PDES applications demonstrate a spatial locality: event messages tend to follow *static* communication channels. Real life simulations that exhibit complete static communication properties include logic networks, many queueing networks, and physics simulations such as the Ising model. Hence, in practice, an LP only communicates with a known, small set of other LP's. Target-specific synchronization information for LP_i is computed using inputs from all LP's that can directly or indirectly send it an event message, based on these known properties of the communication topology. This is our motivation for introducing some directed graph theory and its terminology.

A *directed graph* is a finite, nonempty vertex set *V* together with a (possibly empty) edge set *E* of ordered pairs of distinct vertices. An edge from LP_i to LP_j, indicating that LP_i sends event messages to LP_j, this is represented by (LP_i, LP_j). If there is a directed *path* from LP_i to LP_j in the LP communication graph, then LP_i is a *predecessor* of LP_j. If there is a directed *edge* from LP_i to LP_j, then LP_i is an *immediate predecessor* of LP_j. Likewise, if there is a directed *path* from LP_i to LP_j, then LP_j is a *successor* of LP_i. If there is a directed *edge* from LP_i to LP_j, then LP_j is an *immediate successor* of LP_i.

Given a directed graph representing an LP communication topology, it is desirable to find out if there is a path from LP_i to LP_j for all vertex pairs because each predecessor of LP_j can have a direct or indirect impact on its state because event messages are sent along

paths. The *transitive closure* of a directed graph $G = (V, E)$ is defined as the graph $G^* = (V, E^*)$, where $E^* = \{(LP_i, LP_j)$: there is a *path* from LP_i to LP_j in $G\}$.

2.5. Summary and Conclusions

We have discussed concepts and terminology for parallel discrete event simulation, reduction operations, networks which compute reductions, sorting networks, and directed graph theory in preparation for the presentation of our results on the computation of reductions in support of parallel discrete event simulation.

In the next chapter we present a reduction-based framework for parallel discrete event simulations as introduced by Reynolds [REYN91] [REYN92]. In this chapter we develop this framework with respect the applicability of the framework, the computation model, the detailed hardware design, and correctness criteria. In Chapter 4 we present our contributions with respect to the algorithmic component of the framework. In Chapter 5 and Chapter 6 we present our contributions on the computation of target-specific reductions.

3 A Framework for Parallel Discrete Event Simulations

Our parallel discrete event simulation framework, first introduced by Reynolds [REYN91], is a combination of hardware and algorithms in support of parallel simulations. This framework, a low-level characterization of all PDES synchronization protocols, provides a model for the efficient computation of critical synchronization values, such as GVT, particularly in support of the rollback chip [FUTG92] and adaptive aggressive PDES synchronization protocols [SRIN93]. The original framework, as proposed by Reynolds [REYN91] [REYN92], outlined the three components of the framework and provided low-level algorithms to support a conservative PDES. Our research contributions are built on this work.

Reynolds and Pancerella [REPA92] first described specific functional and implementation details of the hardware component of the framework. We presented the prototype hardware design and applications of the hardware-based framework to a wide range of PDES synchronization protocols [REPS92]. Finally, the framework was further developed by Reynolds, Pancerella, and Srinivasan [REPS93] with criteria for its correct operation, functional characteristics, description of a prototype hardware design, and performance results first reported by Srinivasan [SRIN92]. Our contributions to the framework, as presented in these publications, will be presented in this chapter.

The framework itself is comprised of both hardware and software to support PDES. There are three components to this framework: (1) reduced values that characterize the state of a parallel simulation; (2) an abstract computation model which is the mechanism for the

reduction and dissemination of these synchronization values; and (3) synchronization algorithms that guarantee that the reduced values represent a correct state. The computation model is realizable on any closely coupled cluster of processors or any parallel machine. This computation model is designed to ride the technology wave, such that as processors and memories get faster, the framework can exploit these gains.We use the term "framework" to describe the entire software/hardware ensemble meant to support PDES.

Our framework has been described extensively to date. (See [REYN91], [REYN92], [REPA92], [PANC92], [REPS92], and [REPS93].) In this chapter we discuss our contributions to the framework in three areas: the applicability of the framework to a wide range of PDES synchronization protocols, the correctness criteria on which the framework hardware is based, and the hardware component of the framework. In Section 3.1. we discuss reduced values in parallel simulations. In Section 3.2. we present correctness criteria for the computation and dissemination of multiple reduced values in a parallel simulation. In Section 3.3. we describe the computation model for the computation and dissemination of reduced values. This computation model adheres to the established correctness criteria. In Section 3.4. we describe a suggested high-level implementation of the computation model, and in Section 3.5. we give low-level details of this implementation, based on the four-node prototype designed and built by the Departments of Computer Science and Electrical Engineering at the University of Virginia. Details of the electrical design for the prototype hardware have been completed by McGraw [MCGR93] and Brown [BROW93]. In Section 3.6. we present some algorithms which execute on the framework hardware and support parallel simulations. Throughout this chapter we make it evident how the framework supports a wide range of PDES synchronization protocols.

3.1. Reduced Values in Parallel Simulations

As defined in Chapter 2, a reduced value is the result of a binary, associative operation. In this section we present reduced values that are useful in parallel simulations. For simplicity, we assume all reduced values are computed *globally,* across *n* logical processes. In Chapter 5, we introduce the notion of *target-specific* reductions.

A small set of globally reduced values can describe the synchronization state of any parallel simulation. To do so, each logical process (LP) must maintain local counterparts to each global value. These values are logical timestamps, so we refer to them as *T-values*. The upper case "*T*" in the "*T*-values" signifies logical time, which is the range of these functions, as against real time, which is their domain. While each *T*-value is a actually function of real time, for example $\sigma_i(t)$ is the *T*-value for the local clock of LP_i at real time *t*, we will only use the function notation when it is necessary to reference a *T*-value at a particular instance along the real time line. To compute a global *T*-value, each LP maintains a corresponding local *T*-value, the local counterpart to the global *T*-value computation. In the next few sections, we explore various *T*-values in parallel simulations. Each of the *T*-values we present are computed in a reduction network.

3.1.1. Reduced Values in Conservative Parallel Simulations

A set of *T*-values to support a conservative PDES synchronization protocol have been presented in detail [REYN92] [REPA92]. The *T*-values computed are η', the minimum next event time, and υ', the minimum logical timestamp of messages that have been sent but not acknowledged. These global *T*-values are computed as follows:

$$\eta' = MIN_{LP_i}(\eta_i) \text{ , for all } LP_i,\ i = 1, 2, \ldots, n$$

$$\upsilon' = MIN_{LP_i}(\upsilon_i) \text{ , for all } LP_i,\ i = 1, 2, \ldots, n$$

where η_i is the timestamp of the first event in LP_i's events list (which is assumed to be sorted in non-decreasing timestamps order), υ_i is the logical timestamp of the smallest of all messages that LP_i has sent out which may or may not have been received by their intended receivers and about which LP_i does not know, and *n* is the number of LP's. η_i is the *next event time* of LP_i, and υ_i is the *smallest unreceived message time* of LP_i. The next event time η_i of LP_i is maintained as the smallest logical time in LP_i's event list. The smallest unreceived message time can be updated at two times during the execution of the simulation: (1) when a message is sent to another LP and (2) when a message acknowledgment is received. These two values are maintained locally. Accordingly, η' is the *minimum next event time* and υ' is the *minimum unreceived message time* for all LP's.

One method of correctly maintaining υ' across all processors is to use message acknowledgment schemes which are based on the computation of reduced values. (See Chapter 4.) Messages can be acknowledged with a *tagged selective reduction operation*, such that a particular message can be identified. (See Section 3.5.4.).

These values, along with synchronization algorithms to correctly maintain them, are sufficient to eliminate causality errors and support deadlock-free parallel simulation even when message traffic is always present. The maintenance of υ' takes into account the messages that are in transit in the host communication network. The elimination of causality errors allows an LP to recognize when it can commit to processing an irreversible act such as I/O. Details can be found in [REYN92].

3.1.2. Reduced Values in Optimistic Parallel Simulations

In an optimistic PDES synchronization protocol, such as Time Warp [JEFF85], GVT can be efficiently computed by an LP *at any time* using our framework (See [SRRE93b]). To compute GVT, two globally reduced values across all LP_i's, $i = 1, 2, \ldots, n$, where *n* is

the number of LP's, must be computed: σ', the smallest logical clock in the system, or *minimum logical clock time*, and υ', the logical timestamp of the smallest outstanding message in the system, or *minimum unreceived message time*. These global *T*-values are computed as follows:

$$\sigma' = MIN_{LP_i}(\sigma_i) \text{ , for all LP}_i, i = 1, 2, \ldots, n$$

$$\upsilon' = MIN_{LP_i}(\upsilon_i) \text{ , for all LP}_i, i = 1, 2, \ldots, n$$

where σ_i is the logical clock of LP_i, υ_i is the logical timestamp of the smallest of all messages that LP_i has sent out which may or may not been received by their intended receivers but have not been acknowledged, and *n* is the number of LP's. The logical clock σ_i of LP_i is updated at two times during the execution of an optimistic PDES: (1) at the start of a new event to be executed, and (2) when a message is received that causes a rollback to a time in LP_i's logical past. In an optimistic PDES, the *T*-value υ_i is updated to include outstanding message times of both event messages and antimessages. The two values σ_i and υ_i are maintained locally by each LP_i. GVT, by definition, is computed as the minimum of these two values:

$$GVT = MIN(\sigma', \upsilon')$$

GVT computation and dissemination in this framework is a significant improvement in algorithm complexity and implementation efficiency over existing GVT maintenance schemes ([JEFF85], [JESO85], [SAMA85], [LILA89], [BELL90], [COKE91], [TOGA93]).

3.1.3. Reduced Values as Lookahead Values in Parallel Simulations

Minimum event processing times and lookahead values can be computed as globally reduced values. For example, the smallest future time that an LP can send event messages can be computed as a globally reduced value. Each LP computes the value it submits to this reduction based on its current local clock and its minimum processing time.

In addition to lookahead values, other possible reductions which could be applied to parallel simulations are estimations of the maximum (or minimum) rate at which an LP is processing events. If each LP submits a current estimate of its rate of simulation, the fastest (or slowest) LP (with respect to logical time) can be identified. Felderman and Kleinrock [FEKL92] show analytically that a Time Warp simulation can be more efficient if a faster LP is slowed down; they do not propose how the information might be propagated. We have a framework for disseminating this information easily.

3.1.4. Reduced Values in Iterative Parallel Simulations

Iterative PDES synchronization protocols, such as Bounded Lag [LUBA88a], Moving Time Window [SOBW88], and the aggressive Global Windowing Algorithm proposed by Dickens [DICK93], require the computation and dissemination of *ceiling values* or *fault values*. Lubachevsky [LUBA89] requires global establishment of minimum next event time and other values to compute opaque periods. Global windowing protocols, such as those proposed by Nicol [NICO93] and Dickens [DIRE92], require establishment of parameters for the window. These values are defined as reductions across all LP's. Furthermore, additional global reduction values, such as a minimum outstanding message time or a minimum next event time, could enhance iterative PDES synchronization protocols. For example, a window may be enlarged by including this additional knowledge.

3.1.5. Reduced Values as Termination Detection Conditions in Parallel Simulations

The challenge of global termination detection and the calculation of output measures in a PDES [ABRI91] can be realized easily using reduction operations within our framework. Many global termination conditions — for example, sums and boolean operations — can be calculated and disseminated efficiently as globally reduced values. Unlike Chandy and Lamport's distributed snapshot algorithm [CHLA85], a framework

consisting of synchronization values and related algorithms can be used to evaluate termination conditions even when there are outstanding messages in the parallel simulation. As mentioned above, computing minimum unreceived message times and acknowledging messages in a reduction network can be used in order to detect outstanding messages in a system. Moreover, a sum of the number of all messages sent minus messages received at all LP's can be computed as reductions to detect outstanding messages in the system. If this value is maintained correctly, a sum of zero indicates that there are no outstanding messages in the host communication network.

We now develop the necessary correctness criteria for computing and disseminating multiple reduced values across LP's.

3.2. Correctness Criteria for Computing Multiple Reduced Values

As shown in the previous section, many examples have appeared in the parallel simulation literature demonstrating the need for globally computed values. The hardware-based framework we describe in this chapter is meant to compute these values correctly and expediently.

We begin by formally defining the concept of *globally reduced value*. We assume the existence of n logical processes (LP's). Let the state of LP_i be represented by the m-tuple $<V_i^1, V_i^2, ..., V_i^m>$. A global reduction function, F, one which produces a *globally reduced value*, operates on n values, one from each LP_i: the n-tuple $<V_1^k, V_2^k, ..., V_n^k>$ for a given k in $1 \ldots m$ is the input to the global reduction function, F_k. The reductions we desire can be characterized by:

$$F_k = \sigma_k \langle V_1^k, V_2^k, \ldots, V_n^k \rangle, \text{ for } k = 1, \ldots, m$$

where σ_k is an associative, binary operator applied to the k^{th} n-tuple. So, for example, if V_i^1 is LP_i's next event time, then $F_1 = min<V_1^1, V_2^1, \ldots, V_n^1>$ would be the minimum next event time for all n LP's.

In the discussion that follows we refer to $<V_i^1, V_i^2, \ldots, V_i^m>$ as LP_i's *state vector*, and the V_i^k's, k in $1 \ldots m$, as components of LP_i's state vector.

When computing globally reduced values, it is best to allow the computation of these values to proceed asynchronously with the simulation. Ideally, we desire a total ordering with respect to this asynchronous computation. That is, if at some real time t, LP_i computes a new V_i^k and at some time greater than t, LP_j, j not necessarily distinct from i, computes a new V_j^l, l not necessarily distinct from k, the computation of F_k should complete before the computation of F_l.

A total ordering is desirable because it carries sequencing information with it that could be exploited by the LP's. For example, the order in which globally reduced values are computed (along with the ID's of the LP's that caused the computations to occur) can reveal information about which LP's are sending messages to others. That information, in turn, could be used within an LP to determine, for example, its probability of receiving a simulation-related message over some interval of time.

While a total ordering is desirable, it is both expensive and unnecessary for parallel simulations. Guaranteeing total ordering requires cooperation among LP's with respect to the order in which global functions should be invoked. This kind of cooperation can be achieved, but at a cost. The cost can be temporal: LP's must continually execute the equivalent of barrier synchronizations, or the cost can be monetary: specialized networks such as those proposed by Ranade, Bhatt, and Johnnson [RABJ88] and Reynolds, Williams, and Wagner [REWW89] [REWW92] would be required in addition to our framework. We have chosen to weaken the desire for total ordering because doing so leads to more efficient

and cost-effective solutions and because we can accomplish all we need with a simpler and less expensive approach.

In place of total ordering we consider the concept of *sequential consistency* [LAMP79], which is defined as follows: for a given sequence of changes to the values of components in the state vector of an LP_i, the order in which global reductions appear to be applied to those values must be the same as the order in which the values were changed. An interesting characteristic of sequential consistency is its appearance of correct execution order. Consider a sequence, *S*, of changes to the values of components of a given LP's state vector. The order in which members of *S* are used in global reductions only matters at those times when the result of a particular global reduction applied to, say, the i^{th} member of *S* is used in a computation. At that point in time, all global reductions on the first *(i-1*) members of *S* should be completed. Thus, sequential consistency guarantees a "no later than" property with respect to the order of application of global reductions: at those points in time at which we observe the results of a global reduction, all global reductions that should precede it are also complete, but we have no way of determining in which order they completed.

We define *observable sequential consistency* to be the case where all global reductions are treated as though their values are used in subsequent computations; that is, the order in which members of *S* are used in global reductions is identical to the order in which the respective component values appear in *S*. The difference between sequential consistency and observable sequential consistency is that the former only guarantees the "no-later-than" property with respect to the order of reductions, where the latter guarantees a strict ordering. Observable sequential consistency is necessary if it is required that LP's be able to observe the effects (on global reductions) of any change in any component in any LP's state vector.

Sequential consistency and observable sequential consistency do not address the ordering of reductions between pairs of LP's, as does total ordering. Instead, they only address the ordering of reductions applied to the sequences of value changes in components of a given LP's state vector. If we consider applying a *sequence* of global reductions in response to changes in component values in LP state vectors, as we have considered so far, then the issue of sequencing global reductions still exists. Consider the case where LP_i changes V_i^k and then changes V_i^l while LP_j first changes V_j^l and then V_j^k. Independent of the temporal interleaving of these changes, if they all occur in an interval short enough so that all changes are completed before any global reductions are initiated, then a sequencing problem exists. Either order of applying F_k and F_l violates the order of component value changes in either LP_i or LP_j. To prevent this, an LP must be able to control the timing of the application of global reductions to changed component values in its state vector. One approach would be to guarantee the following conditions: 1) whenever LP_i changes the value of V_i^k, the application of F_k must occur in a finite, bounded amount of time and 2) there must be a way for LP_i to determine that F_k has been applied to V_i^k.

These conditions can be met by ensuring that the ordering of global reductions is fixed and known. For example, if global reductions were applied cyclically, $F_1, F_2, \ldots, F_m$, then LP's could submit changes to component values with temporal spacing between the changes that equaled or exceeded the time required to complete a cycle of *m* reductions. This approach is sufficient to guarantee observable sequential consistency as long as no data loss occurs. We discuss the effects of data loss next.

If LP's pace their changes to state vector values so that only one value change occurs per cycle of *m* global reductions and if LP's process all information produced by the cyclic application of *m* reductions then observable sequential consistency is guaranteed. The first condition is easily achieved, however, we do not expect LP's will be able to

process a constant flow of globally reduced values. Consider the speed at which such values could be produced. Our prototype hardware [REPS93] computes global reductions, with pipelining, such that new results are produced on the order of every 150 nanoseconds. The absolute timing is not the factor here; rather it is that time relative to a typical processor's instruction cycle time. Given current processor technology at most tens of instructions could be executed during the time a global reduction is performed. We conclude it is not reasonable to expect a processor to keep up with a flow of global reductions.

We could consider slowing the reduction rate so that processors could keep up with the flow of output. However, low latency is critical. When an LP computes a new *T*-value the corresponding global reduction should be completed as quickly as possible. The importance of this is established in our performance analysis of the hardware [REPS93]. This analysis concluded that under normal load, GVT computed on our hardware lags behind the actual GVT by 5-10 microseconds.

As discussed above, cyclic application of the global reductions is a satisfactory replacement for the dataflow approach that a total ordering represents; if changes to state vector components can be properly paced and globally reduced values can be processed without loss then observable sequential consistency can be guaranteed. However, even if loss does occur, we can ensure sequential consistency. Consider treating the application of $F_1, F_2, \ldots, F_m$ as a globally reduced state vector: $<F_1, F_2, \ldots, F_m>$. If, rather than allowing results of the application of individual F_k to be lost, we required the granularity of a loss to be global state vectors, then it is possible to ensure sequential consistency. We elaborate.

At the beginning of a cycle a snapshot is taken of the state vectors for each of the LP_i. Then $F_1, F_2, \ldots, F_m$ are applied to these state vectors. In the meantime the LP_i can change components of local copies of their state vectors. These changes will not be observed until another snapshot is taken at the beginning of the next cycle. At the end of a

cycle the globally reduced state vector $<F_1, F_2, \ldots, F_m>$ is made available to all of the LP's. Some of the LP's may succeed in processing the information and others may not, resulting in a loss.

If, preceding one cycle, LP_i changes V_i^k and just preceding the next cycle it changes V_i^l, then the pair of resulting globally reduced state vectors have a desirable property: the first incorporates the change to V_i^k only and the second incorporates the changes to both V_i^k and V_i^l. Observable sequential consistency is maintained for any processors that process both vectors. For those that lose the first but observe the second, sequential consistency is maintained; that is, the ordering of changes to V_i^k and V_i^l is not necessarily preserved, but never violated. Thus, the "precedes" relation maintained by observable sequential consistency is relaxed to the "no-later-than" sequential consistency.

Many of the ordering requirements that arise in parallel simulation can be satisfied with sequential consistency. For example, computation of GVT requires each LP maintain a current simulation time and a smallest unreceived message time. When an LP completes an event and sends a message to another LP, both its simulation time and its smallest unreceived message time may change. A simulation-wide invariant that must be maintained, as demonstrated by Reynolds [REYN92], is that the event or message in the system with the smallest logical time must always be represented in at least one LP's simulation time or smallest unreceived message time. An LP just completing an event must not allow its new simulation time (which may be infinity) to be a part of a global reduction of all simulation times before its new smallest outstanding message time. However, it is sufficient to allow them to be used in global computations simultaneously. That is, sequential consistency is sufficient to support maintenance of the invariant.

We note that the effects of a particular value for a particular V_i^k need not be observed. It is possible for LP_i to produce a sequence of new values for V_i^k at a rate where

the effects (possible changes in F_k) of only some members of that sequence are observed. If it is important that all changes in state vector component values be used in global reductions, then LP's must control the rate of changes to component values.

Given the state vector $<V_i^1, V_i^2, \ldots, V_i^m>$ for each of n LP_i's, and the desire to produce $<F_1, F_2, \ldots, F_m>$ in a sequentially consistent manner, we have determined that a framework can accomplish this by doing the equivalent of cyclically (1) taking a snapshot of the $<V_i^1, V_i^2, \ldots, V_i^m>$'s, (2) performing each of the m global reductions, F_k, on the captured m-tuples, and (3) presenting the resulting global reductions, $<F_1, F_2, \ldots, F_m>$, to each of the LP_i atomically. While the global reductions are being performed, each LP_i should be able to update local copies of state vector components in preparation for the next snapshot.

In the sections that follow we describe a computation model and an implementation of this model that meet the requirements just given. A simple extension described in later sections enables the framework to produce observable sequentially consistent results as long as no loss of globally reduced m-tuples $<F_1, F_2, \ldots, F_m>$ can be guaranteed.

3.3. Abstract Computation Model

An abstract computation model for computing and disseminating reduced values can be found in Figure 3.1. This model describes the relationship of event processing to logical process synchronization in any parallel simulation. In the following paragraphs we explain each of the components in the computation model. In Section 3.4. we describe a possible hardware implementation of this model, with specific details of this implementation described in Section 3.5.

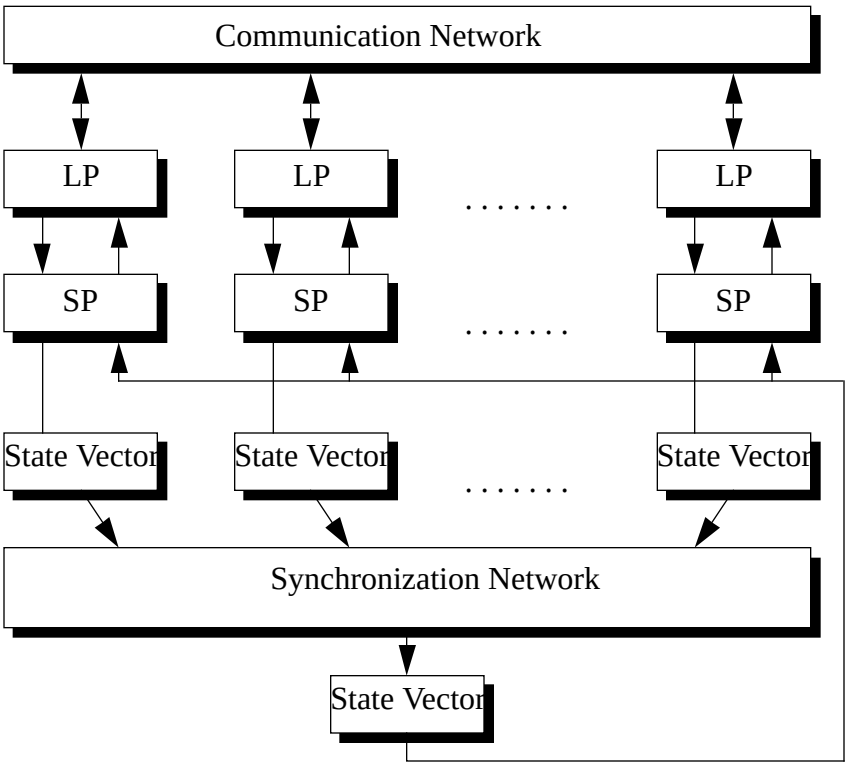


Figure 3.1 Abstract Computation Model.

All event processing and PDES synchronization protocols are executed by the *logical processes (LP's*). LP's send and receive event messages through a *communication network*. The computation model is not tied to a distributed memory model; in a shared memory implementation of the computation model, the communication network will be, in fact, a global shared memory.

Each LP has a corresponding *synchronization process (SP)* associated with it. An LP will notify its SP of changes to its local processing state. Examples of changes to the local state include an advancement in the LP's local clock, notice of sending an event message, notice of receiving an event message, and rollback. Other changes will be dependent on the PDES synchronization protocol, desired reduction operations to be

computed, and the low-level algorithms executed by the SP's. Each SP will maintain an *input state vector* containing local counterparts to each global value. The SP will in turn process the information from its LP and submit a corresponding input state vector to the *synchronization network*, which computes reduced values. As established in Section 3.2., some small set of globally reduced values in a *state vector* can describe the state of any parallel simulation. The SP executes the synchronization algorithms described in future sections so that the input values reflect a correct state and the output values are computed correctly across all LP's. Functionally, the communication from the LP to the SP must be a FIFO as established by the correctness criteria in the previous section.

The synchronization network computes and disseminates reduced values to the SP's. An *output state vector* is written at each SP. An SP reads the output state vector, processes this information, and then writes relevant reduced values to a memory location readable by its LP. Functionally, this shared interface is a single set of registers, as established by the correctness criteria.

In the next section we discuss our implementation of this computation model. There are many possible implementations, including using existing machines such as the CM-5. In our implementation, we employ separate processors for the LP event processing and the SP synchronization processing. LP event processing and SP synchronization processing could be implemented on the same processor; however, the synchronization processing may be a potential overhead, and this will have a direct effect on the finishing time of the simulation. We also employ two separate networks in our implementation; one network (and possibly some processors) could be used for both event messages and reductions, but in that case the computation of reductions may not be performed rapidly and continuously as in our implementation.

3.4. Functional Hardware Description

In this section we focus on a functional implementation of the computation model. A high-level hardware description of this implementation is shown in Figure 3.2. The shaded components represent hardware which is built and interfaced to an existing parallel machine or cluster of computers. This hardware description was first proposed by Reynolds and Pancerella [REPA92]. The host system in our description is a closely coupled network of high speed processors with its own network for interprocess communication. The *host communication network* is independent from the synchronization network, or *reduction network*. Each *host processor* (*HP*) is paired with an *auxiliary processor* (*AP*) which interfaces to the high speed reduction network.

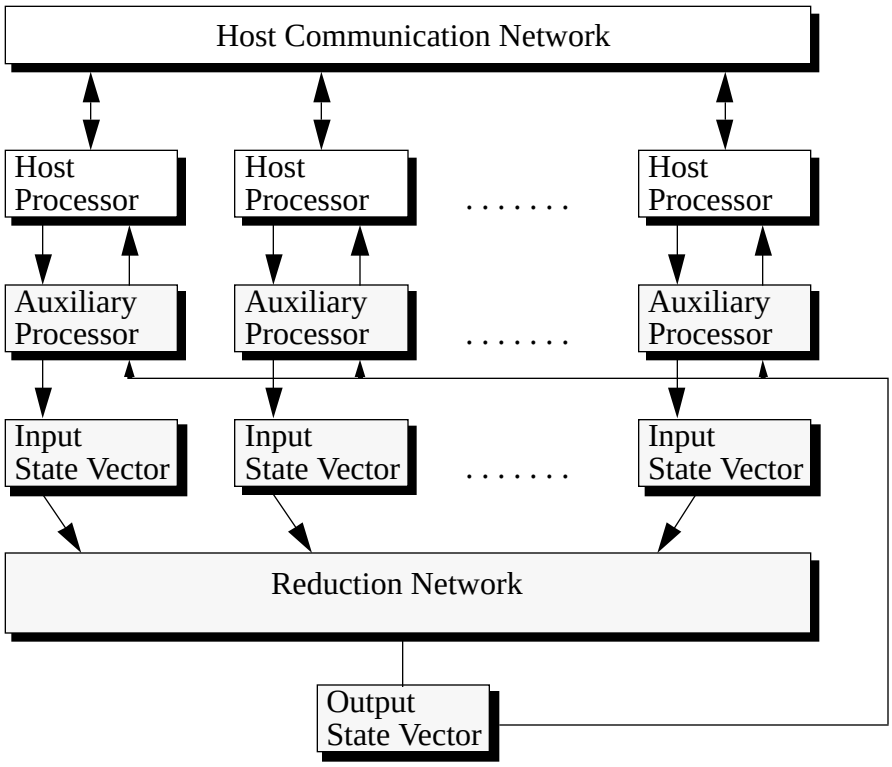


Figure 3.2 High-Level Hardware Description.

The auxiliary processors are dedicated processors which manage the high frequency I/O from the reduction network and execute the synchronization algorithms. A high-speed bidirectional communication channel exists between a host processor and its corresponding auxiliary processor. All interfaces between the host processor and the auxiliary processor (See Section 3.5.3.) and between the auxiliary processor and the reduction network (See Section 3.5.5.) must be designed to ensure the correctness criteria.

An integral part of the hardware is the reduction network. The reduction network rapidly computes and disseminates different binary, associative operations across state vectors of values. Each component of a state vector is an input to a binary, associative operation. For example, it can be specified that all first components are to be summed, all second components OR'ed, and the minimum is to be taken of all third components in a three component state vector. As per the correctness criteria, the state vector is the basic unit of operation in an implementation of the hardware. Interfaces into and out of the reduction network must preserve state vectors: the hardware reads state vectors of size *m*, computes *m* reduced values, and writes a reduced state vector. The hardware guarantees that a partial or incomplete reduced state vector is never read by software running on an AP. The success of disseminating synchronization values is contingent on the high speed at which these values are made available to all processors.

Employing auxiliary processors provides a separation of the synchronization activity (performed on auxiliary processors) and the application being simulated (performed on host processors). The synchronization processes (SP's) execute the high speed synchronization activity in the parallel simulation framework (See Section 3.6.) on the dedicated AP's. The logical processes (LP's) execute on the host processors, and all event messages are sent and received at the host processors. HP's communicate some simulation activities to the AP's. When an SP executing on an AP reads a new reduced state

vector from the reduction network, it writes selected groups of these values into the HP-AP interface readable by the host processor. An LP executing on a host processor can compute GVT, avoid deadlocks, and make processing decisions based on the synchronization values. Other than simple tests such as these, the execution of the framework algorithms does not interfere with an LP's event processing. A further advantage of a dedicated processor interfacing with the host processor and the reduction network is that an AP can compute the input reduction values based on multiple LP's executing on one host processor and coordinate the synchronization activity of multiple LP's. In sum, this framework implementation can off-load all parallel simulation synchronization overhead from host processors and the host network.

3.5. Detailed Hardware Design

In the sections that follow we discuss the specifics of the hardware design of our prototype. This prototype guarantees the established correctness criteria, and in our discussion we focus on how we ensure the criteria. We note that this is not the only hardware design which guarantees the correctness criteria.

3.5.1. Auxiliary Processor

The general layout of an auxiliary processor is depicted in Figure 3.3. Auxiliary processors are fast, general purpose 32-bit microprocessors. Each AP has its own memory to store synchronization programs and related data structures (See Section 3.6.). Furthermore, each AP has EPROM to store a boot-up monitor which is executed upon reset.

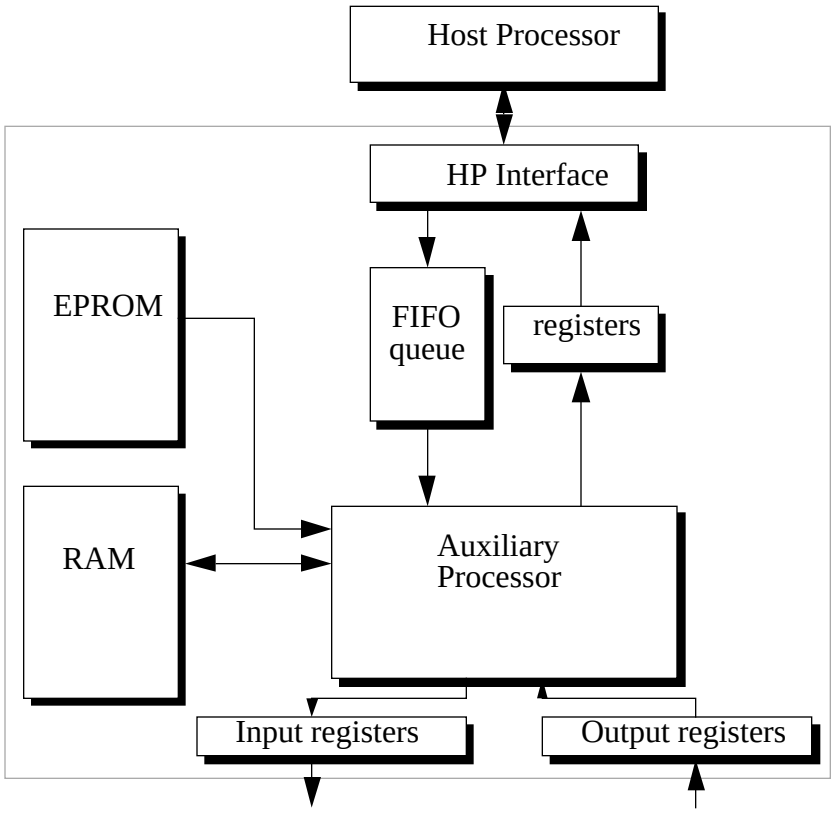


Figure 3.3 Auxiliary Processor.

3.5.2. Setup

Each auxiliary processor boots up in a “listening” state in which it monitors its host processor interface. A host processor sends tagged data to its auxiliary processor representing a program to be loaded and executed by the AP. The physical interface between a host processor and its auxiliary processor is described in the next section.

One of the host processors in the system and its corresponding auxiliary processor are designated as a *master pair* of processors. The master pair communicates reduction network programming information to the state machine controlling the reduction network.

Critical information to be passed to the state machine includes the number of components in a state vector and the operations to be performed on components.

The master host processor can send tagged data representing new reduction network programming information to its auxiliary processor at any time. Similarly, host processors can send data to their respective auxiliary processors indicating they are to receive new programs to execute. This will permit dynamic reprogramming of the AP's and the reduction network. We assume that applications running on the HP's and programs running on the AP's are sufficiently robust to support this dynamic reprogramming.

3.5.3. Host Processor - Auxiliary Processor Interface

Functionally, there are two data paths between a host processor and auxiliary processor: one from the HP to the AP and the other from the AP to the HP. The HP occasionally writes tagged information to the interface which the AP processes, based on the tag, and generates state vectors to input into the reduction network. Similarly, the AP writes globally reduced values to the interface which are subsequently read by the HP.

The addition of dedicated processors requires correctness criteria to be preserved between a host processor and its auxiliary processor. There are two requirements on the data path from an HP to its AP: (1) no information sent by the HP is lost and (2) the AP processes the data in the order in which it is sent by the HP. Under the established correctness criteria in Section 3.2., an application executing on the HP does not need to read and process all globally reduced values; a recent version of globally reduced values, however, is expected to be available to the HP. This suggests an implementation requires at least a FIFO queue from HP to AP and a set of registers that can be written and read atomically from AP to HP.

As seen in Figure 3.3, the host processor can access the FIFO queue and the registers via an HP interface. The HP interface isolates the particular host processor from the rest of the system. If the host system changes, this HP interface is the only thing that will need to be redesigned. Isolating the HP interface provides adaptability to other parallel computers or closely coupled networks. For example, the HP interface could be changed from a SCSI to a VME interface, and all that would be required is the logic to respond to requests by the HP on the FIFO queue and register bank.

3.5.4. The Parallel Reduction Network

The *parallel reduction network* (*PRN*), is the reduction network which computes and disseminates the results of global reduction operations. As seen in Figure 3.4, the PRN is a binary tree of depth $\log_2 n$, where n is the number of host (and auxiliary) processors. Each node of the tree is an Arithmetic Logic Unit (ALU) with some logic for tagged selective operations. Each stage of the PRN consists of half as many ALU's as the stage above it, with the first stage having *n/2* ALU's. The PRN's binary tree properties allow a global reduction operation to be computed and disseminated in *O(log n)* time. We note that

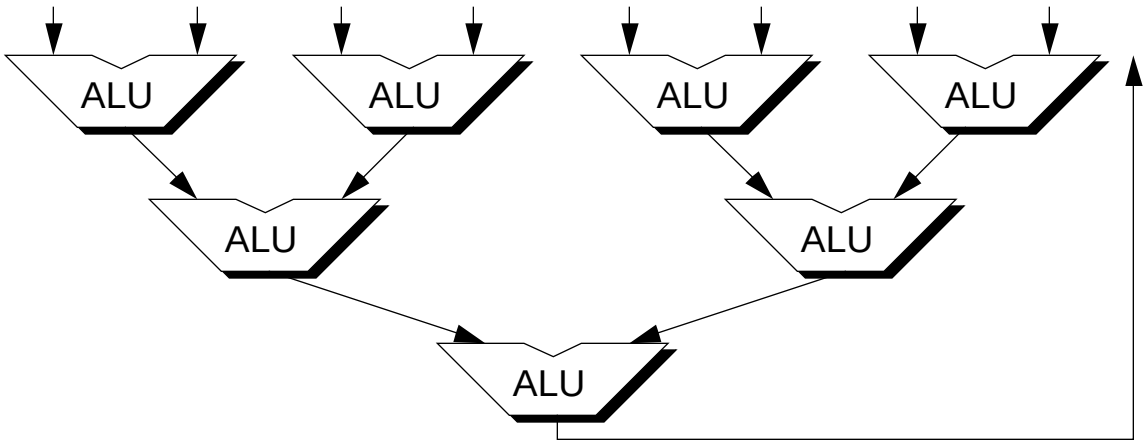


Figure 3.4 Parallel Reduction Network.

the PRN does not have to be a binary tree; it could, for example, be constructed as a quad tree.

A single ALU node is shown in Figure 3.5. The ALU's perform reduction operations, i.e., binary, associative operations on two inputs based on a programmed operation code which accompanies the inputs; operations include sum, minimum, maximum, logical AND, logical OR, etc. Each input *data register* is paired with a *tag register*. The ALU's support *tagged selective operations*; in a selective reduction operation, such as minimum or maximum, a tag accompanies the "winning" value of the binary operation. The PRN propagates the tag of the input that "wins" a selective operation, a minimum or maximum operation, so that the tag of the smallest or largest component emerges from the bottom of the PRN for a minimum or maximum operation. In the case where there is no single choice in a selective operation (i.e., both operands are equal), the PRN selects deterministically the tag which is propagated. A selective operation requires two operations in the ALU: a compare and a select.

As shown pictorially in Figure 3.5, two inputs and two reduction operation codes arrive at an ALU node. An error check is performed on the reduction operation codes; if the two operations are not equal, an error flag is placed in the tag, and the tag is propagated through the reduction network. After a reduction is performed, the resultant value and the operation code are propagated to the next stage of the PRN.

Pipelining is employed in order to use the reduction network efficiently: partial results are pipelined through the *log n* stages of the PRN such that each stage of ALU's is always busy. The PRN can pipeline reduction operations at a rate equal to the delay time of a stage. The time for a value to pass from one level of the PRN to the next is a *minor cycle time*. The time required for the top row of LP's to read all the elements of the state vectors is called the *input cycle time*. Since the stages of the reduction network are pipelined, an

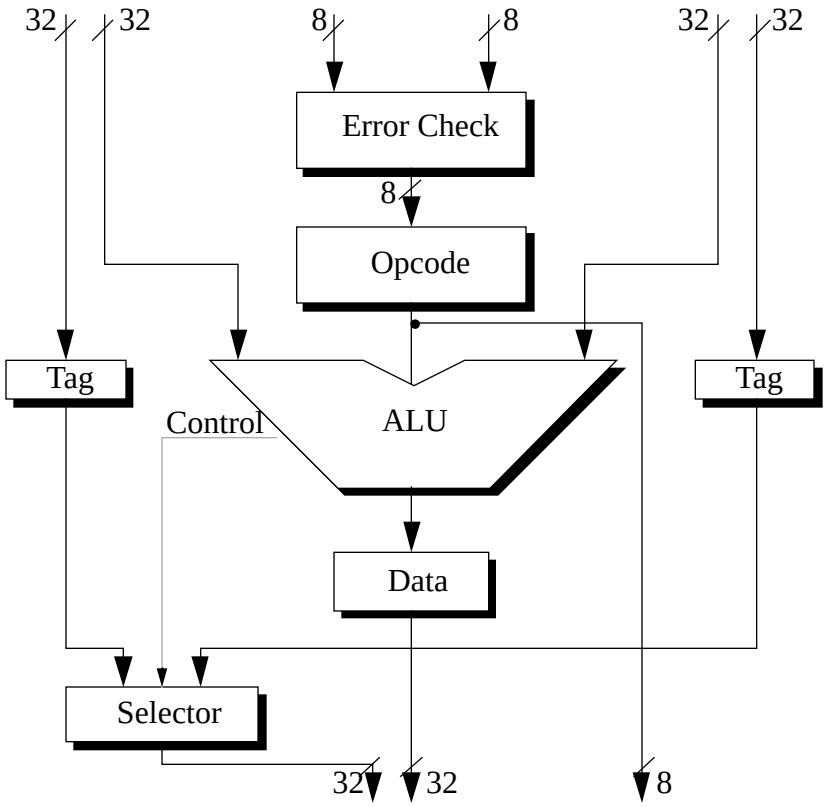


Figure 3.5 An ALU Node in the Parallel Reduction Network.

input cycle will consist of *m* minor cycles, where *m* is the size of the state vectors. Thus, the time to produce a globally reduced state vector of length *m* is $c \cdot m$ nanoseconds, where *c* is the minor cycle time, plus the time to fill the pipe which is $c \cdot log_2 n$ nanoseconds. We refer to the time to compute a globally reduced state vector as a *reduction cycle*. Since speed of computing reduced state vectors is the primary design goal of the hardware, it is important that *c* be small. The minor cycle time in our prototype hardware is 150 nanoseconds, giving a reduction cycle of time of 1.2 microseconds for 32 processors with 4-element state vectors.

As seen in Figure 3.3, the interface to the PRN from each AP is identical. Each AP has sets of memory-mapped input registers and memory-mapped output registers. A processor can write to the input registers and read from the output registers; the PRN will read values from the input registers and write the corresponding globally reduced results into the output registers. This memory-mapped interface is a possible source of memory contention if both the PRN and the auxiliary processor attempt to access the input or output registers simultaneously. We discuss next how the interface between the auxiliary processor and the PRN is constructed in order to minimize memory contention, to facilitate atomic writes with and without overwrite capabilities, and to preserve state vectors.

3.5.5. Auxiliary Processor-PRN Interface

The AP-PRN interface is designed to operate on state vectors in order to support both atomic accesses of globally reduced values and order preservation of input values to the reduction network. From an SP's point of view, it feeds a *valid* state vector to the PRN, where "valid" is defined by the application using the framework hardware. The PRN reads the state vectors, processes them by performing the corresponding reduction on each component, and writes a globally reduced state vector at each AP. Furthermore, the hardware provides an atomic read access to a single output state vector so that an AP can read an entire state vector. The application software should access whole state vectors, not individual components, if consistent states are required by the application.

An auxiliary processor and the reduction network operate asynchronously with respect to one another. As shown in Figure 3.6, three banks of input and output register pairs provide an interface of isolation, such that both can access the register banks with minimal interference. This interface is designed to guarantee that *the PRN never blocks* while waiting to read a value or write a value. The PRN is expected to read and process state vectors at a rate much faster than an AP produces them; the PRN, therefore, may read

and process the same state vector repeatedly. Even if an input value changes with high frequency, it will very likely be used more than one time in the computation of a reduction due to the relative speeds of AP's and the PRN. Similarly, on the output side, the PRN will produce globally reduced state vectors faster than an AP can read and process them, and as a result the AP's may lose some state vectors. Applications executing on the framework hardware will have to tolerate the loss of globally reduced state vectors. All reads to registers from the PRN or an AP are nondestructive. We now discuss the input and output interfaces in greater detail.

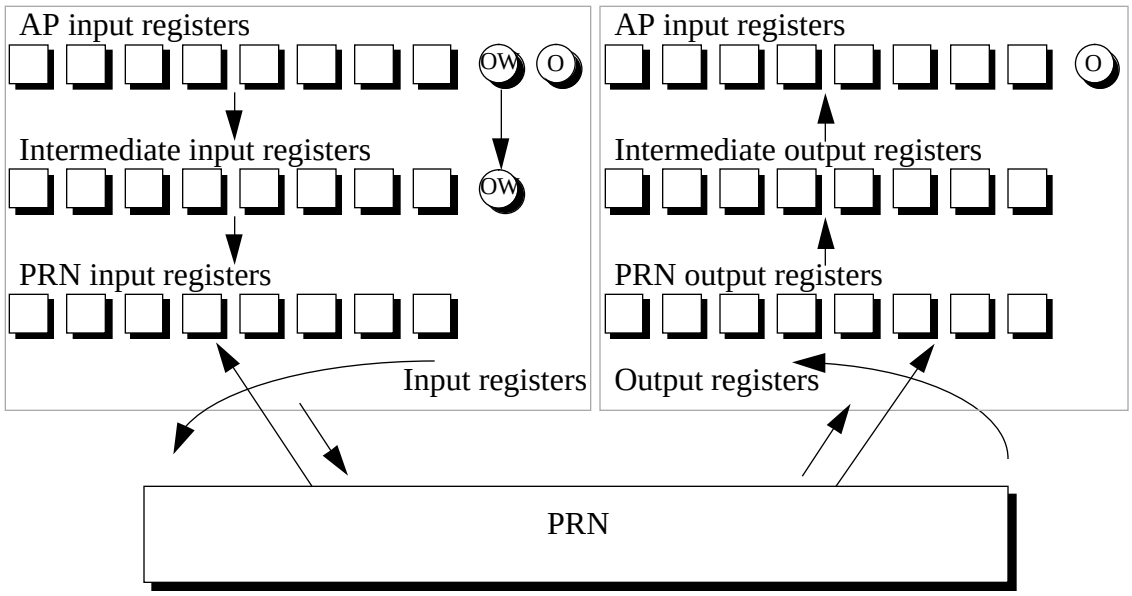


Figure 3.6 Interface Between an Auxiliary Processor and the PRN.

3.5.5.1. Auxiliary Processor-PRN Interface: Input

The interface from an auxiliary processor to the PRN consists of three banks of register pairs: the AP input registers, the Intermediate input registers, and the PRN input registers. The AP writes state vectors of size *m* to the top row of registers, the AP input registers, and the PRN reads state vectors of size *m* from the bottom row, the PRN input

registers. The state machine which controls the interface transfers state vectors from the AP input registers to the Intermediate input registers and then to the PRN input registers. The transfer is done so as to minimize interference. Intermediate registers facilitate getting snapshots of valid local state vectors to be passed on to the PRN input registers without blocking the PRN.

When an auxiliary processor has completed writing a new state vector, it sets two single-bit control flags: the *overwrite bit* (OW) and the *owner bit* (O). The owner bit is *always* set when the AP has finished writing a valid state vector into the AP input registers; this indicates that the interface controller now owns the top level of registers. When the interface state machine transfers this state vector to the Intermediate input registers, it resets the owner bit indicating that the AP once again owns the AP input registers. If the AP attempts to write to the AP input registers while the owner bit is still set, it will be blocked. However, given the relative speeds of the PRN and the AP, this is not expected to happen often.

The overwrite bit gives the application some control over what values are eventually fed into the reduction network. Specifically, if the AP marks a state vector as “non-overwritable”, it is guaranteed that the entire vector will be processed by the PRN. When the control logic transfers the AP input registers to the intermediate level, the overwrite bit is also transferred. If the AP indicates a state vector is overwritable then the state machine controlling the register banks can allow subsequent state vectors written by the AP to overwrite the state vector in the Intermediate input registers. If the AP signals a state vector as non-overwritable and it is transferred to the intermediate registers, the overwrite bit will prevent the transfer of a newly written AP level state vector until the contents of the Intermediate input registers are transferred to the PRN input registers. The control logic guarantees that AP input registers are only moved to the Intermediate input

registers when this process does not cause the PRN to block or when it does not lead to a loss of integrity of a state vector. Finally, we note that due to the relative speeds of an AP and the PRN, it is very unlikely that an overwritable state vector will be overwritten prior to being read by the PRN; however, we have designed the reduction network to provide the guarantee anyway, for future use.

The combination of non-overwrite on input to the PRN and no loss of state vectors on output is sufficient to guarantee the observable sequential consistency introduced in Section 3.2.. If either of these conditions cannot be met then observable sequential consistency cannot be guaranteed. However, neither is required to guarantee sequential consistency. We discuss this further at the end of this section.

The PRN reads state vectors of a specified size cyclically, starting with the m^{th} component and proceeding to the first component. Thus, the PRN reduces the m^{th} component, followed by the $(m-1)^{st}$, and so on. The PRN is pipelined; thus the processing of the $(i-1)^{st}$ components commences as soon as the top level of ALU's completes processing the i^{th} components. The PRN reads the i^{th} register pair from each of the n input banks simultaneously. The time for the PRN to read an entire state vector is an *input cycle*. An input cycle finishes when the first components of the state vector are consumed. At the end of an input cycle, the controller transfers the Intermediate input registers to the PRN input registers. The transfer can be overlapped with the last PRN read in the input cycle; thus, the hardware requires a minimum state vector size of two so that this transfer can be performed as efficiently as possible. The transfer from the intermediate registers to the PRN registers has a higher priority than the transfer from the AP registers to the intermediate registers so that the PRN never blocks.

We note that $\log_2$ of n and m are not necessarily equal. Therefore, while the PRN is reading from the i^{th} input register pair from all n processors, it is not necessarily writing the

i^{th} output register pairs. That is, the PRN may complete reading state vectors from each of n input register banks at a different time than when it completes writing new reduced state vectors. The writing of a reduced state vector for a set of input state vectors will lag by (((*m*-1) + $\log_2 n$) · *c*) nanoseconds, where the minor cycle time is *c* nanoseconds, the state vector size is *m*, and there are *n* processors.

3.5.5.2. Auxiliary Processor-PRN Interface: Output

As shown in Figure 3.6, the three banks of output registers are constructed to preserve state vectors and to minimize AP-PRN interference in a similar fashion to the input register banks. Once every *m* minor cycles (assuming a full pipe in the PRN), the PRN generates a globally reduced state vector, which is written to the *PRN output registers*. This state vector is transferred to the *Intermediate output registers* and finally to the *AP output registers*, which are readable by the AP. Once again the interface controller guarantees that the PRN never blocks, and transfers between output register levels are prioritized to prevent this.

Each time the PRN completes writing a state vector into the PRN output registers, the values are shifted into the Intermediate output registers. When the bottom row is shifted, the values in the intermediate row are concurrently shifted into the AP output registers unless the AP has locked the top row because it is reading the AP output registers. In that event, the Intermediate output registers are overwritten by the PRN output registers, and the contents of the intermediate registers are lost forever. The AP output registers have a control bit, an owner bit (O), that is set and reset by the auxiliary processor. The owner bit determines whether Intermediate output registers can be written to the AP output registers or are lost; it also ensures an atomic read of a state vector by the AP. The AP sets the owner bit prior to reading the state vector in the AP output registers and resets it after it is done reading the complete state vector. The AP may block momentarily if it attempts to set the

owner bit while the intermediate values are being written in parallel to the registers readable by the AP. Applications using the framework hardware must be robust enough to tolerate the loss of state vectors emerging from the PRN. Consistent with the correctness criteria set forth in Section 3.2., we note that an AP never sees a partial state vector. State vectors are either seen in their entirety or not at all.

The three levels of registers on the input side guarantee sequential consistency by preserving state vectors, as discussed in Section 3.2. Observable sequential consistency requires that the overwrite bit be used whenever the values in a state vector must be used in a global operation. Furthermore, there can be no loss of state vectors on the output side of the PRN — that is, AP's must process every state vector that emerges from the reduction network — if observable sequential consistency is to be maintained. Since this AP-PRN interface does not prevent state vector loss, an alternative, which is the equivalent of observable sequential consistency, is to use two extra input registers and compute tagged selective operations to perform a double handshake [PANC92], as discussed in Chapter 4. We note, however, that it is expensive (in terms of computation time) to implement observable sequential consistency in the framework hardware and it should be avoided when possible.

Specific details about our prototype hardware have been published by Reynolds, Pancerella, and Srinivasan [REPS93]. We now discuss the algorithms that execute on the host and auxiliary processors in order to support parallel simulations.

3.6. Framework Algorithms

Synchronization algorithms are the third component of our PDES framework. Given the correctness criteria in Section 3.2. and the functionality of the hardware described in Section 3.5., framework algorithms execute on both the host processors and

the auxiliary processors and guarantee that the reduced values correctly represent the state of the simulation. We discuss the algorithmic requirements of both the host and auxiliary processors in the following sections. Specific details about framework algorithms can be found in [REYN92] (conservative PDES algorithms), [SRRE93] (optimistic PDES algorithms), and the next chapter (message acknowledgment algorithms).

3.6.1. Host Processor Algorithms

In our parallel simulation framework, all processing of an LP — event processing and event message sending and receiving —occurs on the HP. The HP must communicate any simulation events which represent a change to the simulation state vector to its AP in order to correctly compute reduced state vectors in the reduction network. Examples of events which affect GVT computation include a change in an LP's local clock, due to either an event execution or a rollback; the receipt of a message or antimessage; and the sending of a message or antimessage.

In general, the additional processing on the host processor will be minimal. The host processor only needs to notify its AP of a change in its local state as described above. As seen in Figure 3.3, the interface between an HP and its AP is functionally a FIFO. An HP enqueues tagged entries into the FIFO with the tag indicating the nature of the communication.

3.6.2. Auxiliary Processor Algorithms

All processing of the SP's will be executed on the auxiliary processors. The AP will alternately read the output from the reduction network, write this output to the host processor buffer or registers, perform low-level synchronization algorithms dependent on the PDES synchronization protocol and the required computations in the reduction network, and read the host processor FIFO for a change in the LP's state. A typical AP

algorithm will have the format in Figure 4.7, where the algorithm is executed continually with the simulation.

```
AUX_PROC:   WHILE simulation is executing
                  Read the PRN output;
                  IF    global state has changed
                  THEN  Write global state vector to HP interface;
                  Perform synchronization algorithms;
                  IF    FIFO is not empty;
                  THEN  Get next entry from FIFO;
                        Process entry;
            END WHILE
```

Figure 4.7 Auxiliary Processor Algorithm Format.

In the next chapter we present specific auxiliary processor algorithms for acknowledging event messages in a reduction network in support of the computation of υ', the minimum unreceived message time for all LP's in a PDES.

3.7. Summary and Conclusions

We have presented in this chapter our contributions to Reynolds's original framework for parallel discrete event simulation. This framework provides novel and efficient support of PDES synchronization protocols.

We identify three primary contributions of the work presented in this chapter. First, we have demonstrated the applicability of this framework to a wide range of PDES synchronization protocols. The applicability of the framework to PDES synchronization protocols began in [REYN91] with reduced values and low-level algorithms to support conservative protocols. It continued in [SRIN92] with the applicability to optimistic protocols. In this chapter, we have expanded on this and shown the applicability to the computation of lookahead values, the execution of iterative PDES synchronization protocols, and the detection of termination conditions in a PDES.

Second, we have provided sound correctness criteria for this framework. The correctness criteria define the computation and dissemination of multiple reduced values, where LP's are executing asynchronously. Furthermore, the correctness criteria do not require the blocking of either the processors executing events or the reduction network computing the critical synchronization values.

Finally, we have developed the hardware design at three levels. At the highest level, we have defined a computation model which decouples the LP processing from the synchronization of LP's. This model provides an abstraction of PDES processing such that the model can be realizable by many implementations. Furthermore, this model is able to advance with advances in technology. At the functional level, we have described an implementation of the computation model that adheres to the established correctness criteria. This functional implementation employs separate processors for event processing and synchronization processing, and separate networks for reduction operations and event messages. This functional description allows the synchronization processing to occur with near-zero overhead to the PDES. Finally, at the detailed hardware design we have proven the feasibility of designing detailed components and interfaces which are both correct and efficient.

In the next chapter we explore the acknowledgment of event messages in a reduction network. The algorithms and computed values assume the detailed hardware design presented in this chapter. The acknowledgment of messages is important to the computation of a minimum outstanding message time, which is useful in non-aggressive PDES synchronization protocols, aggressive PDES synchronization protocols, and adaptive aggressive PDES synchronization protocols.

4 Acknowledgment Messages in a Reduction Network

A probable source of performance degradation in Reynolds's original PDES framework algorithms [REYN91] (See Chapter 3.) (from here on known as Reynolds's framework algorithms) is the communication of synchronization information among LP's, i.e., acknowledgment messages, outside the reduction network. This is undesirable. Message acknowledgments are critical to the computation of the minimum unreceived message time which is used in the computation of global virtual time. The hardware-based framework presented in the previous chapter provides for the high speed computation of GVT, yet the acknowledgment of messages in a host communication network can reduce the performance gains of the framework. This chapter presents several novel approaches which use a reduction network for message acknowledgments.

In order to make the presentation of our algorithms simpler, we assume the PDES synchronization protocol executing on the host processors is an aggressive one, and the reduction network will be used to compute global reductions, including GVT. All message acknowledgment algorithms presented in this chapter can be used to correctly compute *T*-values in a non-aggressive PDES. Since our acknowledgment algorithms will execute on auxiliary processors in the framework hardware configuration depicted in Figure 3.2, their execution will in no way interfere with normal PDES event processing on the host processors or the event message traffic in the host communication network.

First we show efficiency problems with Reynolds's framework algorithms. In Section 4.2. we discuss the necessary algorithmic requirements for using a reduction

network to acknowledge messages. In Section 4.3. we present a solution which employs a two reduction handshake algorithm in the reduction network. This work was first introduced by Pancerella [PANC92]. In Section 4.4. we discuss an alternative to this algorithm, one that requires a single reduction operation to be computed in the reduction network. We prove the correctness of this alternative in Section 4.5. In Section 4.6. we present improvements for all message acknowledgment algorithms which use the reduction network. In Section 4.7. we discuss important issues which must be considered when selecting one particular algorithm. Finally in Section 4.8., we present performance results of two different acknowledgment algorithms executing on our four-node prototype framework hardware.

4.1. Efficiency Considerations of the Framework

As shown in Chapter 3, GVT can be computed as the minimum of two globally reduced values across all LP_i's, $i = 1, 2, \ldots, n$: σ', the minimum logical clock time, and υ', the minimum unreceived message time.

In order to maintain υ' all event messages must be acknowledged. In the algorithms proposed by Reynolds [REYN92] acknowledgment messages are sent through the host network of the parallel machine. There are three problems with using the host communication network, and not the high-speed reduction network, for acknowledging messages. We discuss these next.

4.1.1. A Significant Lag Time for Critical Synchronization Values

If messages are acknowledged in the host network, this has an impact on the accuracy of the computation of υ': the value of υ' will lag behind σ' by at least the host network latency time. Since the reduction network will have a lag time typically 10^{-3} that

of a host network, the framework can benefit greatly if υ' or a good approximation of υ' were computed in the reduction network.

4.1.2. Additional Message Traffic in the Host Network

The use of acknowledgment messages in the host communication network can degrade the performance of a PDES in another way. Sending one acknowledgment message for each event message in a distributed memory machine doubles the message traffic (in a shared memory machine the analogous problem is memory contention); hence, the arrival of both event messages and acknowledgment messages is potentially delayed. A doubling of message traffic can often have a serious impact on performance, since performance often degrades super-linearly with message volume.

4.1.3. A Potential Race Condition

If two separate networks are used to disseminate synchronization information, as in Reynolds's framework algorithms, there is a possible race condition between acknowledgments in the host network and computations of υ' in the reduction network. Assume the communication topology in Figure 4.1, where LP's are executing an optimistic PDES synchronization protocol executing on top of our framework hardware. (See Section 3.1.2.) The reduction network computes two globally reduced values in this PDES: υ', the minimum unreceived message time, and σ' the minimum logical clock across all LP's. A race condition can occur with the following sequence of events:

1) $\sigma_1 = 4$ and $\sigma_2 = 7$. $\sigma' = 4$ and $\upsilon' = \infty$. GVT = $\min(\sigma', \upsilon') = 4$.
2) LP_1 finishes processes its event at time 4.
3) LP_1 sets υ_1 to 4, which causes υ' to change to 4, one reduction cycle later.
4) LP_1 sends a message to LP_2 with timestamp 4.
5) LP_1 sets its σ_1 to 5 and begins to process the event at time 5.

6) All global values now reflect the changes made in LP_1: $\sigma' = 5$ and $\upsilon' = 4$. GVT = $\min(\sigma', \upsilon') = 4$.

7) LP_2 receives the message, sets σ_2 to 4, and sends an acknowledgment to LP_1.

8) *No* global values yet reflect the changes made in LP_2.

9) LP_1 receives the acknowledgment message and sets υ_1 to ∞.

10) υ' is computed to be ∞. $\sigma' = 5$. GVT = $\min(\sigma', \upsilon') = 5$. This is the incorrect computation of GVT since $\sigma_2 = 4$.

The race condition is not prevented since the hardware supporting the framework *cannot* control the order that acknowledgments are received with respect to the order in which the globally reduced values are updated.

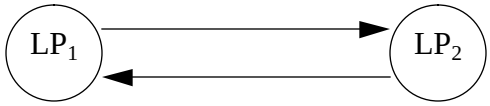


Figure 4.1 A Simple PDES Communication Topology.

Given the problems with using the host network for acknowledgments, we conclude it may be better to use the synchronization network for message acknowledgments. As shown in the sequel this conclusion is quite appropriate.

4.2. Acknowledging Messages in a Reduction Network

Without loss of generality, assume that each LP_i, $i = 1, 2, \ldots, n$, occupies a unique physical processor pair, HP_i and AP_i. We note that all acknowledgment algorithms can be extended to support multiple LP's executing on a processor pair. Synchronization tasks execute on the AP, and PDES protocol-dependent and application-specific tasks execute on the HP. In an aggressive PDES, the HP will perform event execution, sending and receiving event messages, state saving, and rolling back the computation while the AP will perform

all tasks required to maintain local *T*-values, including event message acknowledgment algorithms. Each HP_i maintains the local clock, the events list, and the outgoing message list (list of antimessages) required in an aggressive parallel simulation. Each AP_i maintains data structures in support of the event message acknowledgments and reduction operations supporting the PDES synchronization protocol. Specific details regarding the correctness of an aggressive PDES executing on a processor pair in the framework hardware can be found in [SRRE93].

4.2.1. Host Processor Requirements for Acknowledgment Algorithms

Recall that the communication channel between an HP and its AP is functionally a FIFO (See Section 3.5.3.). An HP enqueues tagged entries into the FIFO with the tag indicating the nature of the communication. The HP must communicate the following simulation events to its AP in order to correctly compute GVT and perform message acknowledgments in the reduction network: a change in an LP's local clock, due to either an event execution or a rollback; the receipt of a message or antimessage; and the sending of a message or antimessage.

Event messages will be acknowledged in the reduction network by employing a tagged selective operation. This requires a unique tag for each message, since multiple messages may have the same logical timestamp. Each message is assigned a *globally unique* message ID, a unique tag consisting of a message sequence number between a sender-receiver pair concatenated with a sender ID and a receiver ID, where sequence numbers are issued in numerical order for each sender-receiver pair. A message can be viewed as a tuple, *{message time, message ID}*, where the message time is a logical timestamp and the message ID is a unique identifier. Message sequence numbers are maintained by each LP_i executing on its HP_i as follows:

- $S_i(j)$— the sequence number for each LP_j, $j = 1, 2, \ldots, n$, to which it sends a message.When LP_i sends a message to LP_j, it increments the counter $S_i(j)$ and uses its value in the message identifier. Message identifiers form a contiguous sequence for each ordered pair of communicating LP's. This will be crucial for batched acknowledgments described later.

4.2.2. Data Structures and Values Maintained by Each AP

Each AP_i maintains the following local *T*-values and data structures:

- σ_i — local clock of LP_i.
- For each LP_j to which LP_i sends a message, an *outstanding message list* of message tuples. A message is removed from an outstanding message list once AP_i reads a message acknowledgment from the reduction network and processes this acknowledgment.
- υ_i — smallest timestamp of a message in all outstanding message lists in LP_i.
- For each LP_k which sends messages to it, AP_i maintains an *unacknowledged message list*. The unacknowledged message list is sorted by sequence number. A message will be removed from the unacknowledged list when it has been determined that the AP of the sending LP has processed its acknowledgment.

An auxiliary processor AP_i *processes an acknowledgment* by removing the corresponding message from the correct outstanding message list and updating υ_i to be the current smallest timestamp in all of its outstanding message lists.

4.2.3. A New *T*-Value for Message Acknowledgments

A new local *T*-value is defined for acknowledging messages in the reduction network: ρ_i is the ordered pair $\{message\ time, message\ ID\}_i$ such that *message time* is the smallest timestamp of a message that has been received by LP_i but for which an acknowledgment has not completed and the *message ID* is its message identifier. If an LP has no messages to acknowledge, it sets its ρ_i to $\{\infty, \Phi\}$, denoting a null acknowledgment. Each LP_i sends its ρ_i into the pipelined reduction network, and ρ', the global minimum *{message time, message ID}* across all LP's, is computed and disseminated in the reduction network. The global minimum of the ordered pairs is computed as minimum message time;

the message ID serves only to identify the specific message that is acknowledged. Hence, ρ' is computed across all LP's and disseminated to those LP's:

$$\rho' = MIN_{LP_i}(\rho_i) \text{ , for all LP}_i, i = 1, 2, \ldots, n$$

4.2.4. Auxiliary Processor Algorithms for Message Acknowledgments

In Section 3.6. we described the general format of an auxiliary processor algorithm. In the first message acknowledgment algorithm we present, each auxiliary processor executes the specific algorithm in Figure 4.2. An auxiliary processor alternately reads the output from the reduction network, writes this output to the host processor registers, performs the acknowledgment algorithm CHK_ACK which we describe next, and reads the host processor FIFO for a change in the LP's state. In the event of a message receipt, the AP performs the process RCV_MSG in Figure 4.3, which initiates an acknowledgment.

```
AUX_PROC:   WHILE simulation is executing
                  Read the PRN output;
                  IF    global state has changed
                  THEN  Write global state vector to HP interface;
                  Perform CHK_ACK;
                  IF    FIFO is not empty;
                  THEN  Get next entry from FIFO;
                        CASE (entry_type):
                              NEW_CLOCK: σ_i := new_clock_value;

                              SENT_MSG:   IF    message_time < υ_i
                                          THEN  υ_i := message_time;
                                          Add message to ordered
                                          outstanding message list;

                              RCVD_MSG:   Perform RCV_MSG;
            END WHILE
```

Figure 4.2 Auxiliary Processor Algorithm.

The algorithms in Figure 4.2 and Figure 4.3 operate as follows. The ordered pair ρ_i in LP$_i$ is initialized to $\{\infty, \Phi\}$, indicating no event messages have been sent through the network. ρ_i is maintained for messages received, not messages sent; however, it is

monitored by each LP_i (actually all auxiliary processors serving LP's) since it replaces a physical acknowledgment message in the host network. Specific details follow.

```
RCV_MSG:    IF    ρ_i = {∞, Φ}
            THEN  ρ_i  := {message time, message ID};
            ELSE  Add message to unacknowledged message list;

CHK_ACK     IF    ρ' = ρ_i                                    -- RECEIVER
            THEN  Update ρ_i ;
                  Remove message ID from unacknowledged message list;

            IF    ρ' in outstanding message list              -- SENDER
            THEN  Remove message ID from outstanding message list;
                  Update υ_i , if necessary;
```

Figure 4.3 Algorithms for Receiving Messages and Processing Acknowledgments.

The algorithms RCV_MSG and CHK_ACK are defined in order to maintain υ'. The process RCV_MSG is performed by an AP when a message is received at the HP, and the process CHK_ACK is performed repeatedly as the output of the reduction network changes

Without loss of generality assume that LP_s refers to any LP sending a message and similarly LP_r refers to any LP receiving a message. When LP_s sends a message and HP_s notifies AP_s, the SENT_MSG process from Figure 4.2 is executed by the AP_s. When LP_r receives a message and HP_r notifies AP_r, the RCV_MSG process from Figure 4.3 is executed by AP_r.

In process SENT_MSG, an entry of {*message time, message ID*} is made to the outstanding message list for LP_s. In process RCV_MSG, an entry of {*message time, message ID*} is made to the unacknowledged message list for LP_r, indicating a new message is received but unacknowledged. The local value of ρ_i is updated immediately by AP_r only if its current value is equal to $\{\infty, \Phi\}$, indicating no message is unacknowledged. No physical acknowledgment message is sent through the host network in this algorithm. We discuss the implications of updating ρ_i after we explain process CHK_ACK.

CHK_ACK is a process that executes once each time through the auxiliary processor loop as shown in Figure 4.2. In CHK_ACK each AP monitors the value of the global minimum unacknowledged message ρ'. The ordered pairs emerging from the reduction network are equivalent to acknowledgment messages. If the ordered pair ρ' is in LP_s's outstanding message list, indicating that LP_s had sent the message, then AP_s removes the corresponding message from this outstanding message list. The value of υ_s is modified accordingly, i.e., updated when ρ' is the message acknowledgment for the outstanding message υ_s. If $\rho' = \rho_r$, indicating that LP_r's acknowledgment is the minimum, then ρ_r is updated and the corresponding entry in the unacknowledged message list is removed by AP_r. In Section 4.2.5. through Section 4.2.7. we discuss performance and correctness issues of process CHK_ACK.

It is tempting to have AP_r update ρ_r in RCV_MSG if a new message with a smaller message time arrives. This preemption of ρ_r guarantees that ρ_r represents the smallest received but not acknowledged message for LP_r, yet there are two reasons why it is necessary to update ρ_r in a nonpreemptive manner. First, a problem occurs if the preempted ρ_r is equal to the computed ρ'; this situation causes AP_s to assume that the message is acknowledged, yet AP_r, if it failed to recognize the equality of ρ_r and ρ' before changing ρ_r, would fail to determine that the acknowledgment was read and processed by AP_s. AP_r would submit the same ρ_r to the network at some later time and would eventually read and process an output state vector from the reduction network such that $\rho_r = \rho'$. AP_s, the auxiliary processor of the sender of the message, reads and processes the acknowledgment, i.e., ρ' is in its outstanding message list, before AP_r has removed ρ_r from its unacknowledged message list. This approach is inefficient since the performance of the framework is degraded by the dual submissions of the same ρ_r by AP_r. A lesser problem occurs with a steady flow of messages into LP_r: ρ_r could be modified with sufficient

frequency that *none* of LP_r's received messages are ever acknowledged. In other words, there is a potential, albeit unlikely, problem with starvation: no ρ_r will ever be equal to ρ', and hence, no message received by LP_r will ever be acknowledged.

4.2.5. Performance

An undesirable feature of the algorithms shown in Figure 4.2 and Figure 4.3 is that the acknowledgment of messages is serialized. The time complexity of this algorithm to acknowledge *one* message is *O(log n)*, where *n* is the number of LP's. This proposed acknowledgment algorithm may not scale well to large numbers of LP's. If each LP always has at least one message to acknowledge then the expected time until the acknowledgment for each LP's first message is complete is 1/2**n** log n*, or *O(n log n)* complexity.

A further problem is that the amount of time between acknowledgments is greater than one reduction cycle time of the reduction network. When $\rho_r = \rho'$, AP_r must read the output register containing ρ', must recognize the acknowledgment, and then rewrite ρ_r; this is not an insignificant amount of processing and should take an additional reduction cycle since ρ_r will not be updated before it is again read by the network. Scalability is also an issue in Reynolds's framework algorithms since a moderate number of additional acknowledgment messages may flood the host network. Recall, however, that in practice the reduction network can be orders of magnitude faster than the host communication network.

We have presented work using two reduction networks to compute synchronization values: one network produces the PDES state *T*-values, such as υ' and σ', and the other network is dedicated to the acknowledgment of messages [PANC92]. This has an advantage over the serial acknowledgment algorithm since its best case performance can be shown to be *O(1)* and this best case *O(1)* complexity occurs under heavy load. This suggests that υ'

will be more accurate because acknowledgments are performed in less time. With two reduction networks, however, there is again a potential race condition between updating PDES state *T*-values and acknowledgment *T*-values. This race condition can be eliminated if the two reduction networks are synchronized. By synchronizing the two networks, the order of changes to *all T*-values, including ρ_i, for a given LP is guaranteed to be preserved in global counterparts.

4.2.6. Batched Acknowledgments

From the description above, it is clear that acknowledgments in the reduction network are serialized, such that only the acknowledgment with the smallest timestamp will complete in a reduction cycle. This serialization can be alleviated by acknowledging a batch of messages in a single physical acknowledgment. This enhancement improves the efficiency of employing the reduction network to acknowledge messages. Messages arriving from the same sender can be acknowledged as *sequences* by acknowledging the message ID of the largest in-sequence message; this same scheme is common in computer networks [TANE89].

Batched acknowledgments can be implemented by adding a third component to ρ_r and ρ'. Now these message acknowledgment *T*-values have the form: {*message time, message ID, batch size*}. An acknowledging AP_r searches its unacknowledged message list for a batch of received messages with contiguous sequence numbers and sets its ρ_r to the triple {*the smallest timestamp in the batch, the message sequence number of the first message in the batch, the number of messages in the batch*}. Note that in PDES's with aggressive processing, the first message in a batch will not always have the smallest timestamp. A null batched acknowledgment, indicating that there are no unacknowledged messages, is denoted as $\{\infty, \Phi, 0\}$.

To implement batched acknowledgments on the reduction network, a concatenation of the smallest sequence number of a contiguous batch and the number of messages acknowledged in the batch are placed in the tag register and the message timestamp in the data register. (See Section 3.5.4.)

Simulations [SRIN92] demonstrate that the batching of acknowledgments makes the system more robust. As the load increases on an AP, its unacknowledged message lists start growing. As a consequence, contiguous batches of messages form, and therefore, with batching of acknowledgments, AP's perform more work per unit time than with single acknowledgments. A second, important conclusion of Srinivasan's simulations is that since with batched acknowledgments the hardware saturates at smaller event granules, the batched acknowledgment enhancement improves the stability of the PRN (effectively increases its bandwidth).

We assume in the remainder of this chapter that all message acknowledgments are batched.

4.2.7. Correctness

Unlike changes to σ' and υ' in the framework, ρ' must be monitored closely by each AP for the modified algorithms to be correct and efficient. The process CHK_ACK is sensitive to output data from the reduction network and is expected to evaluate every ρ' that emerges from the network. The auxiliary processor of the receiver of the message, AP_r, must recognize $\rho_r = \rho'$ as soon as possible in order to update ρ_r promptly. Similarly, the auxiliary processor of the sender of the message, AP_s, must detect an acknowledgment from AP_r before AP_r stops submitting the ρ_r, representing the acknowledgment, to the reduction network. AP_s then removes ρ' from LP_s's outstanding message list and modifies υ_s accordingly.

In practice, AP_s may miss an acknowledgment (a value of ρ' intended for AP_s) emitted from the reduction network since an AP_r could update its $\rho_r = \rho'$, the smallest acknowledgment in the system, prior to AP_s reading and processing it. Hence, the modified synchronization algorithms do *not* guarantee that all acknowledgments are read and processed by their intended recipients or that υ' is maintained correctly without some severe assumptions regarding the detection of outputs from the reduction network. Hence, the theoretical results presented in the previous section are only obtainable if the hardware can guarantee that *all* ρ''s are read by the AP's that need to process them. Recall that the detailed hardware design presented in Chapter 3 *does* allow state vector loss on the output side of the reduction network. We now present an alternative `CHK_ACK` process, an acknowledgment handshake, which guarantees correctness even when reduction network output results are overwritten prior to being read, i.e., even with output state vector loss.

4.3. Two-Phase Acknowledgment

The following *two-phase acknowledgment* (*TPA*) uses the computation and dissemination of two reduction operations in order to guarantee that all acknowledgments are read and processed by AP's of sending LP's and that υ' is correct. We introduce another local *T*-value, τ_s, which is the ordered pair $\{$*message time*, *message ID*$\}_s$ where the message tagged by *message ID* (which contains information about batches) has been acknowledged by both the receiver LP_r (actually AP_r) and then by the sender LP_s (actually AP_s). τ', the global minimum acknowledged message, is computed across all LP's and disseminated to those LP's:

$$\tau' = MIN_{LP_i}(\tau_i), \text{ for all } LP_i,\ i = 1, 2, \ldots, n$$

The modified `CHK_ACK` algorithm in Figure 4.4 shows how the two-phase acknowledgment is implemented. The `CHK_ACK` procedure consists of two parts. The first `IF` statement implements the part of TPA executed by the AP for an LP receiving a

message, LP_r, while the second `IF` statement is executed by the AP for an LP sending a message, LP_s. The `RCV_MSG` procedure in Figure 4.3 is executed by AP_r to begin the acknowledgment algorithm of the first received message or of a message received when there are no other unacknowledged messages.

Once AP_s detects that ρ' is an acknowledgment for a batch of messages it has sent, it sets its τ_s to ρ' to echo the acknowledgment. AP_s then removes the messages in the batch from LP_s's outstanding message list, if they have not been previously removed, and updates υ_s, if the received acknowledgment was for its smallest outstanding message. If AP_s detects that ρ' is *not* acknowledging a batch from another sender, it sets its second acknowledgment τ_s to $\{\infty, \Phi, 0\}$, the idle acknowledgment, so that other acknowledgments can be answered with a second phase acknowledgment. The AP_r with the smallest handshake acknowledgment (i.e., $\tau' = \rho_r$) can then update ρ_r to its smallest unacknowledged batch since it has determined that AP_s has read and processed the acknowledgment.

```
CHK_ACK:    IF    (τ' = ρ_i) AND (ρ_i ≠ {∞, Φ, 0})          -- RECEIVER
            THEN  IF    unacknowledged list is not empty
                  THEN  Remove next batch to be acknowledged from
                              unacknowledged list;
                        Set ρ_i to acknowledge this batch;
                  ELSE  ρ_i := {∞, Φ, 0};

            IF    ρ' has been sent to this LP               -- SENDER
            THEN  τ_i := ρ';
                  IF    ρ' messages in outstanding message list
                  THEN  Remove the acknowledged batch from outstanding
                              message list;
                        Update υ_i, if necessary;
            ELSE  τ_i := {∞, Φ, 0};
```

Figure 4.4 Modified Synchronization Algorithm Using Two-Phase Acknowledgments.

The advantage of this double acknowledgment approach is that acknowledgments can take place as fast as the algorithm will allow since they are not tied to any other

activities (e.g. the computation of GVT). As discussed, all acknowledgments will be processed by both sending and receiving AP's. This comes at the cost of having to do two reductions in the reduction network. A proof of correctness for TPA can be found in [SRRE93].

4.3.1. Performance

In practice, we expect good performance from the version of TPA in Figure 4.4 since there are dedicated auxiliary processors monitoring the high-speed output from the reduction network and executing all acknowledgment algorithms. Furthermore, simulations [SRIN92] show that under normal load, the mean time to complete a two-phase acknowledgment is around 10 microseconds in a 32-processor system. In these simulations, the time to acknowledge a message was measured from the moment an AP starts an acknowledgment until it receives a second acknowledgment from the AP whose message it is acknowledging. This is competitive with the current technology for existing communication systems; for example, a zero byte message sent from node to node on the Intel Paragon [INTE93] can take 30-70 microseconds using commercial messaging layers and the latency can be reduced to about 5 microseconds with a lower overhead layer [CHIE94]. In comparable time our acknowledgments are processed at the *software* level on AP's; process to process acknowledgments an a Paragon can be orders of magnitude more expensive.

Next we present a single phase acknowledgment which eliminates the need for one of the two reduction operations in the reduction network.

4.4. Single Phase Acknowledgment

TPA requires two acknowledgments in a handshake. Two values must be reduced and disseminated in the reduction network, and this may delay subsequent

acknowledgments. Now we present a potential improvement to acknowledging messages in the reduction network which is correct even with state vector loss; this algorithm requires only a single minimum value to be computed and disseminated in the reduction network. We name this algorithm *single phase acknowledgment* (*SPA*).

Recall that GVT is computed as the minimum of two globally reduced values: minimum local clock σ' and minimum unreceived message time υ'. The acknowledgment algorithm described in this section uses the advance of *computed GVT, $GVT_c(t)$*, the global virtual time computed in the reduction network at real time *t*, as the second acknowledgment in a handshake. (See Section 4.4.1.) We observe that a message acknowledgment must have been received if GVT has increased beyond the message's timestamp since υ' must be greater than the received message's timestamp, and so the message is removed from an LP_r's unacknowledged message list when this happens. The acknowledgment algorithm makes no assumption that the PDES synchronization protocol is Time Warp; GVT is simply computed as the minimum time in the simulation, regardless of the synchronization protocol used. Hence, in a non-aggressive PDES, this algorithm may not reduce the number of operations computed in the reduction network but it does eliminate the lag to handshake acknowledgments.

The algorithms `SENDER`, `RECEIVER`, and `NEW_GVT` in this section eliminate the second reduction operation and together replace the `CHK_ACK` routine, described in previous sections. Also, the algorithm `SENT_MSG`, which executes on an AP, has been modified, and the algorithm `NEW_MSG` replaces `RCV_MSG`.

We present two cases for this acknowledgment algorithm: (1) a non-FIFO case, which makes no assumption about the order in which messages are received between a sender-receiver pair, and (2) a FIFO case, which assumes that all messages between a sender-receiver pair will be received in the order in which they are sent. The difference

between the two is the method in which the unacknowledged message list is stored. Also, we suggest a round robin scheme for increasing the total number of acknowledgments that can complete before the next advancement of computed GVT. Before presenting these algorithms, we discuss in the next two sections the computation of global virtual time in a reduction network and the importance of unique logical timestamps to the correctness of these algorithms. Both of these concepts are critical to the understanding of SPA and its correctness proof.

4.4.1. Computing Global Virtual Time in a Reduction Network

When global virtual time is computed in a reduction network, it lags behind the actual global virtual time in the system being simulated. Before explaining this lag we require some definitions:

Definition: $\sigma_i(t)$ is the local clock time of LP_i as observed in AP_i at real time *t*. $\sigma_i(t)$ is a *T*-value which may be read by the PRN from AP_i's PRN input state vector at real time *t*. Similarly, $\upsilon_i(t)$ is the minimum unreceived message time *T*-value which can be read by the PRN from AP_i's PRN input state vector at real time *t*.

Definition: $\mu_i(t) = MIN(\sigma_i(t), \upsilon_i(t))$. $\mu_i(t)$ is the *minimum timestamp* that may be read by the PRN from AP_i's state vector at real time *t*. Note, however, that there could be a smaller timestamp in the FIFO between HP_i and AP_i because of rollback.

Definition: $\bar{\sigma}_i(t)$ is the local clock value of LP_i at the host processor HP_i at real time *t*. Due to the delay between the time the host processor writes this value to the host processor-auxiliary processor interface, the *T*-value $\sigma_i(t)$ at the same time *t* is not necessarily equal to $\bar{\sigma}_i(t)$. The following equality holds at all real times *t* for some positive finite Δt: $\bar{\sigma}_i(t) = \sigma_i(t + \Delta t)$. Note $\bar{\sigma}_i$ is not necessarily monotonic with respect to *t*, and

therefore neither is σ_i. What we are stating here is that for any value taken on by $\bar{\sigma}_i(t)$, σ_i will have that value at some later time $t+\Delta t$.

Definition: $GVT_a(t)$ is the actual value of GVT in the simulation at real time *t*.

Definition: $GVT_p(t)$ is the value of GVT that can be potentially calculated from the input set of state vectors submitted to the PRN at real time *t*. The input state vectors contain the values of $\sigma_i(t)$ and $\upsilon_i(t)$, for all $i=1\ldots n$. Note that the values of $\sigma_i(t)$ and $\upsilon_i(t)$ do *not* change in the reduction network, so it is only necessary to consider the different values in the host processor and the *T*-values which can be used in a computation in the reduction network.

Definition: $GVT_c(t)$ is the computed value of GVT that is emitted from the reduction network at real time *t*.

At times it will be important to bind the value of computed GVT, GVT_c, to a particular instance in real time. If this is the case, we will use the notation $GVT_c(t)$. If binding $GVT_c(t)$ to a particular real time *t* is not important to the discussion at hand, we use the notation GVT_c. Similarly we use GVT_a and GVT_p.

Figure 4.5 shows the relationship of $GVT_p(t)$ to both $GVT_a(t)$ and $GVT_c(t)$ at real time *t*. The following relation is invariant: $GVT_c(t) \leq GVT_p(t) \leq GVT_a(t)$ at real time *t*, $\forall t$.

Definition: the *advancement of GVT$_c$* occurs if, for a given real time *t* and some finite real time Δt, $GVT_c(t) < GVT_c(t+\Delta t)$.

Next we discuss the necessity of unique timestamps in the reduction network to guarantee the advancement of GVT_c when GVT_a advances.

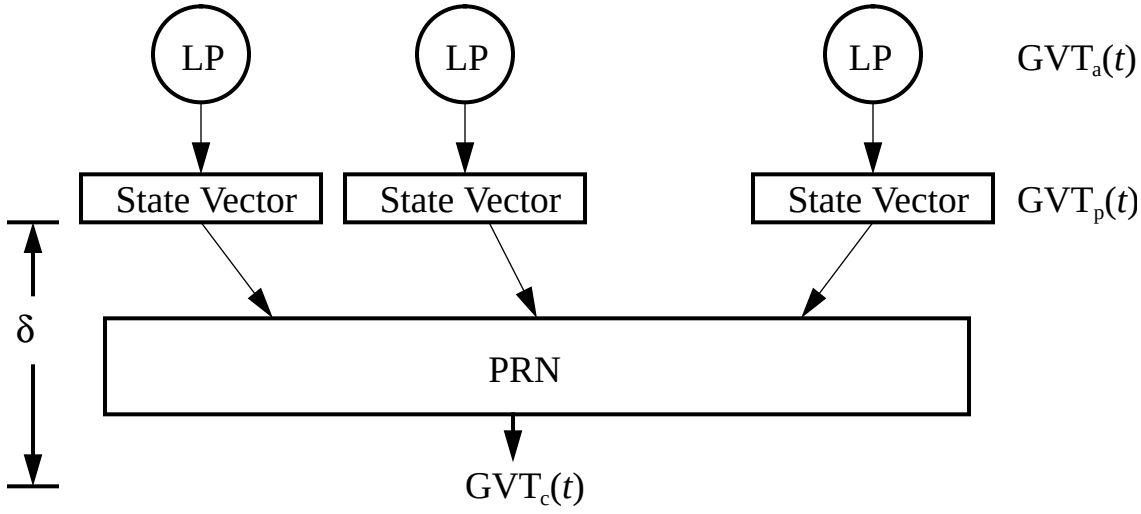


Figure 4.5 GVT Computation Model.

4.4.2. Guaranteeing Unique Timestamps

In SPA the advancement of GVT_c will be used as a second acknowledgment and it is critical that *all* messages are acknowledged so that GVT_c advances. To accomplish this we require that all local clocks and event messages have unique logical timestamps. If unique timestamps were *not* present in the system, we could have a livelock in the computation of GVT_c, as illustrated by the following sequence of events, assuming a linear topology of four LP's:

> 1) $\sigma_1(t_0) = 4$, $\sigma_2(t_0) = 11$, $\sigma_3(t_0) = 3$, and $\sigma_4(t_0) = 10$. $\sigma'(t_0) = 3$ and $\upsilon'(t_0) = \infty$. $GVT_c(t_0) = 3$ at some real time t_0.
>
> 2) LP_1 sets its local clock $\overline{\sigma_1}(t_1)$ to 5, at some real time t_1, $t_1 > t_0$, writes this clock update to its FIFO, and processes an event at logical time 5.
>
> 3) AP_1 reads the entry in its FIFO and updates $\sigma_1(t_2)$ at some real time t_2, $t_2 > t_1$. So, $\sigma_1(t_2) = 5$.
>
> 4) LP_1 enqueues a `SENT_MSG` for logical time 5 at some real time t_3, $t_3 > t_2$, and writes this to its FIFO. This causes $\upsilon_1(t_4)$ to be set to 5 by AP_1 at real time t_4, $t_4 > t_3$. This in turn causes $\upsilon'(t_5)$ to change to 5 at some real time t_5, where t_5 is greater than t_4 by at least time δ.

5) LP_1 sends a message to LP_2 with timestamp 5.

6) All global values now reflect the changes made in LP_1: $\sigma'(t_5) = 3$ and $\upsilon'(t_5) = 5$. $GVT_c(t_5) = 3$.

7) LP_3 sets its local clock $\overline{\sigma_3}(t_6)$ to 5, at some real time t_6, $t_6 > t_5$, writes this clock update to its FIFO, and processes an event at time 5.

8) AP_3 reads the entry in its FIFO and updates $\sigma_3(t_7)$ at some real time t_7, $t_7 > t_6$. So, $\sigma_1(t_7) = 5$.

9) LP_3 enqueues a `SENT_MSG` for logical time 5 at some real time t_8, $t_8 > t_7$, and writes this to its FIFO. This causes $\upsilon_3(t_9)$ to be set to 5 by AP_3 at real time t_9, $t_9 > t_8$. No visible change is made to υ'.

10) LP_3 sends a message to LP_4 with timestamp 5.

11) All global values now reflect the changes made in LP_3: $\sigma'(t_{10}) = 5$ and $\upsilon'(t_{10}) = 5$. $GVT_c(t_{10}) = 5$ at some real time t_{10}, $t_{10} > t_9$.

12) At some real time t_{11}, $t_{11} > t_{10}$, LP_2 receives the message from LP_1, sets $\overline{\sigma_2}(t_{11})$ to 5, and writes these entries to its FIFO. At real time t_{12}, $t_{12} > t_{11}$, AP_2 reads the entries in the FIFO, sets $\sigma_2(t_{12})$ to 5, sets its acknowledgment $\rho_2(t_{12})$ to $\{5, s_{1\text{-}2}, 1\}$, where $s_{1\text{-}2}$ is the sequence number from LP_1 to LP_2. LP_2 will continue to submit this acknowledgment until $GVT_c(t_{12}+\Delta t) > 5$, in some finite real time Δt.

13) At some real time t_{13}, $t_{13} > t_{12}$, LP_4 receives the message from LP_3, sets $\overline{\sigma_4}(t_{13})$ to 5, and writes these entries to its FIFO. At real time t_{14}, $t_{14} > t_{13}$, AP_4 reads the entries in the FIFO, sets $\sigma_4(t_{14})$ to 5, and sets its acknowledgment $\rho_4(t_{14})$ to $\{5, s_{3\text{-}4}, 1\}$, where $s_{3\text{-}4}$ is the sequence number from LP_3 to LP_4. LP_4 will continue to submit this acknowledgment into the reduction network until $GVT_c(t_{14}+\Delta t) > 5$, in some finite real time Δt.

14) At some real time t_{15}, $t_{15} > t_{14}$, all global values now reflect the changes made in LP_2 and LP_4: $\sigma'(t_{15}) = 5$, $\upsilon'(t_{15}) = 5$ and $\rho'(t_{15}) = \{5, s_{1\text{-}2}, 1\}$ (a *deterministic* selection). $GVT_c(t_{15}) = 5$.

15) At some real time t_{16}, $t_{16} > t_{15}$, LP_1 sets its local clock $\overline{\sigma_1}(t_{16})$ to 13, writes this clock update to its FIFO, and processes an event at time 13. AP_1 sets $\sigma_1(t_{17})$ to 13 at some real time t_{17}, $t_{17} > t_{16}$.

16) At some real time t_{18}, $t_{18} > t_{17}$, LP_2 sets its local clock $\overline{\sigma_2}(t_{18})$ to 9, writes this clock update to its FIFO, and processes an event at time 9. AP_2 sets $\sigma_2(t_{19})$ to 13 at some real time t_{19}, $t_{19} > t_{18}$.

17) At some real time t_{20}, $t_{20} > t_{19}$, LP_3 sets its local clock $\overline{\sigma_3}(t_{20})$ to 7, writes this clock update to its FIFO, and processes an event at time 7. AP_3 sets $\sigma_3(t_{21})$ to 7 at some real time t_{21}, $t_{21} > t_{20}$.

18) At some real time t_{22}, $t_{22} > t_{21}$, LP_4 sets its local clock $\overline{\sigma_4}(t_{22})$ to 8, writes this clock update to its FIFO, and processes an event at time 8. AP_4 sets $\sigma_4(t_{23})$ to 8 at some real time t_{23}, $t_{23} > t_{22}$.

19) At some real time t_{24}, $t_{24} > t_{23}$, all global values now reflect the changes made to all local clocks: $\sigma'(t_{24}) = 7$, $\upsilon'(t_{24}) = 5$ and $\rho'(t_{24}) = \{5, s_{1\text{-}2}, 1\}$ (a deterministic selection). $GVT_c(t_{24}) = 5$.

20) At some real time t_{25}, $t_{25} > t_{24}$, LP_2 reads and processes the acknowledgment ρ' and updates $\upsilon_1(t_{25})$ to ∞.

21) At some real time t_{26}, $t_{26} > t_{25}$, all global values now reflect the change in LP_2: $\sigma'(t_{26}) = 7$, $\upsilon'(t_{26}) = 5$ and $\rho'(t_{26}) = \{5, s_{1\text{-}2}, 1\}$ (a deterministic selection). $GVT_c(t_{26}) = 5$. Furthermore, $GVT_c(t_{26}+\Delta t) = 5$ for all real times Δt.

We conclude, without proof, that unique timestamps are necessary for preventing livelock and guaranteeing progress of the reduction operations (i.e. to guarantee that GVT_c eventually increases as GVT_a increases). If non-unique timestamps are present in the system, we can augment timestamp values to create unique timestamp values with the following technique.

At the auxiliary processor level, append the sender ID to the message timestamp when messages are acknowledged in the reduction network. Furthermore, the logical process ID i from LP_i must be appended to *both* its smallest unreceived message time υ_i and its local clock σ_i, so that the ID serves as a tie-breaker for non-unique timestamps. When an LP executing on a host processor receives an event message, it does not see the sender ID in the timestamp field; this identifying ID is only seen at the AP level. The uniqueness of logical timestamps is only necessary for the message acknowledgment algorithm. Hence, there is a finer granularity of logical time with respect to the low-level algorithms executing on the auxiliary processors and not the LP's executing on the host processors, yet the ability for LP's to execute events with the same timestamp in any order is not sacrificed. If the PDES allows multiple messages to be sent to several LP's with the same logical timestamp, a receiver ID can be added to υ_i, and an additional logical process

ID can be appended to σ_i. This algorithm works with non-aggressive PDES synchronization protocols. Now we discuss the effects of rollback and forward processing.

In an aggressive PDES, an LP can process events with the *same* logical timestamp during forward processing and again after a rollback and hence, event messages can have duplicate timestamps. In this case, we suggest appending the unique sender-receiver sequence number to υ_i, and a suitable unused sequence number to σ_i. Another solution to this problem is to use an adapted version of TPA, such that the receiving AP will enter a *handshake mode* to break ties of two or more messages with timestamps equal to GVT_c. In this adapted TPA, an AP_r, the auxiliary processor for a receiving LP, will monitor the handshake acknowledgment reduced value τ' when it is acknowledging a message with time equal to GVT_c, and AP_r will stop submitting this acknowledgment in the reduction network when it receives a second phase acknowledgment, such that $\rho_r = \tau'$. AP_s, the auxiliary processor for a message sender, will submit a handshake acknowledgment τ_s to the reduction network when it reads and processes an acknowledgment for a message with timestamp equal to GVT_c (i.e, its υ_i is equal to GVT_c). Also unique timestamps must be guaranteed for each message-antimessage pair since both messages and antimessages must be acknowledged. We suggest using a single bit appended to the logical timestamp to distinguish between an event message and its corresponding antimessage. When GVT_c is used as an acknowledgment, it is the finer grain of logical time at the auxiliary processor level and in the reduction network.

4.4.3. Data Structures and Values Maintained by Each AP

The host processor requirements for SPA are the same as those for TPA. As with TPA, each auxiliary processor maintains two lists of messages: unacknowledged message list and outstanding message list. Each AP_i also maintains the *T*-values σ_i and υ_i. In addition to these data structures and values, each AP_i must maintain the following:

- $M_i(j)$— the current batch being acknowledged from AP_i's unacknowledged message list for sender LP_j. This is the contiguous batch of messages with the smallest timestamp from sender LP_j, such that the acknowledgment for the batch has not completed: it is a pointer into the unacknowledged message list *j*.
- ρ_i — current active acknowledgment {message time, message ID} from LP_i, one of the $M_i(j)$'s.
- T_{RR}— the total number of iterations of the auxiliary processor algorithm that each $M_i(j)$ will be acknowledged in the round robin acknowledgment. T_{RR} is a constant in SPA.
- T_i — a count of the number of iterations of the auxiliary processor loop for which a particular $M_i(j)$ has been the current active acknowledgment, ρ_i.

The $M_i(j)$'s represent message batches: {message time, message ID, batch size}. We use a functional notation to refer to a single component of the batch; for example, timestamp($M_i(j)$) refers only to the logical timestamp of $M_i(j)$ and sender(($M_i(j)$) refers to the sending LP (LP_j) which is part of the message ID.

4.4.4. General Description of Acknowledgment Algorithm

A round robin acknowledgment algorithm is executed by each LP. Each LP_i submits, in round robin fashion, each of its $M_i(j)$'s as the current acknowledgment ρ_i for an equal amount of time (T_{RR} units) to the reduction network. When LP_i has an $M_i(j)$ with timestamp equal to $GVT_c(t)$ at real time *t*, the *round robin acknowledgment mode*, or *round robin mode*, is suspended, and *GVT acknowledgment mode*, or *GVT mode*, is entered. In GVT mode ρ_i is set to the message acknowledgment with time equal to $GVT_c(t)$. GVT mode is critical to this algorithm and the progress of the simulation: the message with the timestamp at $GVT_c(t)$ *must* be acknowledged so that GVT_c advances. When AP_i, at real time $t+\Delta t$, detects that $GVT_c(t+\Delta t) > \text{timestamp}(\rho_i\,(t+\Delta t))$, $M_i(j)$ is updated, and round robin acknowledgments resume. The algorithm `RECEIVER` is only executed while in round robin mode. The algorithm mode is initialized to round robin and will only change to GVT mode when an $M_i(j)$ has a timestamp equal to the current GVT_c.

Each auxiliary processor executes the following algorithm:

```
AUX_PROC:   WHILE simulation is executing
            Read the PRN output;
            IF    global state has changed
            THEN  Write global state vector to HP interface;
            IF    GVT has changed
            THEN  Perform NEW_GVT;
            Perform SENDER;
            IF    mode = round robin
                  Perform RECEIVER;
            IF    FIFO is not empty;
            THEN  Get next entry from FIFO;
                  CASE (entry_type):
                        NEW_CLOCK: σ_i := new_clock_value;

                        SENT_MSG:   IF message_time < υ_i
                                    THEN υ_i := message_time;
                                    Add message to ordered
                                    outstanding message list;

                        RCVD_MSG:   Perform NEW_MSG;
            END WHILE
```

Figure 4.6 Auxiliary Processor Algorithm for Single Phase Acknowledgments.

The auxiliary processor algorithm remains the same for both the non-FIFO and FIFO acknowledgment algorithms. `NEW_GVT` is the procedure which is executed when GVT_c advances; its primary functions are to check for unacknowledged messages equal to the current GVT_c and to perform necessary garbage collection, i.e., the removal of messages from unacknowledged message lists.

4.4.5. Non-FIFO Case

The non-FIFO algorithms in Figure 4.7 make no assumption about the order that messages are received between a sender-receiver pair. Theoretically, a message with sequence number q could be received later than a message with sequence number $q+l$, $l > 0$.

```
SENDER:     IF    ρ′ has been sent to this LP
            THEN  IF    ρ′ is in any outstanding message list
                  THEN  Remove the acknowledged batch from outstanding
                              message list;
                        IF    timestamp (ρ′) = υ_i
                        THEN  υ_i := smallest timestamp in outstanding
                              message lists;

RECEIVER:   IF    T_i = 0                          -- timeslice end
            THEN  s := next sender;
                  T_i := T_RR;
                  ρ_i := M_i(s);
            ELSE  T_i := T_i - 1;

NEW_GVT:    IF    mode = GVT mode                  -- resume round robin
            THEN  mode := round robin;
                  T_i := T_RR;
            FOR   each sender r
                  Discard all messages, possibly including the batch
                        M_i(r), in unacknowledged message list with
                        timestamps < GVT;
                  M_i(r) := next minimum batch from sender r;
                  IF    ρ_i = old M_i(r)           -- continue round robin
                  THEN  IF    M_i(r) ≠ {∞, Φ, 0}
                        THEN  ρ_i := M_i(r);
                        ELSE  T_i := 0;            -- force timeslice end
                  IF    timestamp (M_i(r)) = GVT
                  THEN  mode := GVT mode;          -- suspend round robin
                        ρ_i := M_i(r);

NEW_MSG:    r := sender(new_msg);
            IF    (new_msg is contiguous with M_i(r))
                        AND (timestamp (new_msg) < timestamp (M_i(r)))
            THEN  Add new_msg to unacknowledged list and coalesce M_i(r)
                        with new_msg and adjacent contiguous batches;
                  IF    ρ_i = old M_i(r)
                  THEN  ρ_i := M_i(r);
            ELSE  IF    (new_msg is not contiguous with M_i(r)) AND
                        (timestamp (new_msg) < timestamp (M_i(r)))
                  THEN  Add new_msg to unacknowledged message list;
                        M_i(r) := batch including new_msg;
                        IF    ρ_i = old M_i(r)
                        THEN  ρ_i := M_i(r);
                  ELSE  Add new_msg to unacknowledged message list;
```

Figure 4.7 Acknowledgment Algorithms Assuming Non-FIFO Channels Between LP's.

The procedure SENDER is the portion of SPA in which LP_s monitors the output of the reduction network for message acknowledgments. When LP_s reads an acknowledgment from the reduction network for a message batch it sent and processes this acknowledgment, it removes this batch from its outstanding message list and updates υ_s accordingly. The procedure SENDER is performed each time an AP reads a new output state vector from the PRN. The procedure RECEIVER controls the counter for the round robin timeslice and cycles through the $M_i(j)$'s, making each the primary message acknowledgment ρ_i in turn, whenever AP_i is executing in round robin mode.

The procedure NEW_GVT is executed *only* when GVT_c has changed. In the first IF statement the AP will restart round robin mode if the mode is set to GVT mode; since GVT_c has advanced, this implies that the message with a timestamp equal to the previous GVT_c has been acknowledged. Next, the AP performs garbage collection on the unacknowledged message lists and updates the $M_i(j)$'s to reflect the changes in these lists. If a specific $M_i(j)$ changes and its previous value is currently being submitted as ρ_i to the reduction network, the *T*-value ρ_i is updated to reflect the change made to $M_i(j)$. If a specific $M_i(j)$ becomes null, indicating an empty unacknowledged list *j and* $M_i(j)$ is currently being submitted as ρ_i, the AP forces the timeslice to end and the next time the procedure RECEIVER is executed, ρ_i will be updated with the next $M_i(j)$ in round robin order. The last IF statement is the test for the equality of the current message batch $M_i(j)$ and GVT_c; if they are equal, the AP enters GVT mode, as described earlier, to force its smallest unacknowledged message to be acknowledged.

The procedure NEW_MSG is executed once each time an AP is notified by its HP that a new event message has been received. NEW_MSG adds the new message to the correct unacknowledged message list and updates the corresponding $M_i(j)$ if the timestamp of the new message is *less* than the timestamp of the current $M_i(j)$. If the batch $M_i(j)$ were updated

to contain the sequence number of the new message, but its timestamp was not smaller than the batch's, the batch $M_i(j)$ could be removed from the unacknowledged list with no guarantee that the acknowledgment with the additional sequence number was read and processed by the sender. To ensure that every message is acknowledged, the batch $M_i(j)$ is updated to include a new message *only* if the timestamp of the new message is smaller than the timestamp of the batch $M_i(j)$.

4.4.6. FIFO Case

The FIFO algorithms in Figure 4.8 assume messages are received by LP_r in the same order in which they are sent by LP_s. The maintenance of unacknowledged message lists is much easier in the FIFO case: each AP_i need only maintain two message tuples for each sender LP_k:

- $P_i(k)$— the current batch being acknowledged for LP_k.
- $B_i(k)$— the next batch to be acknowledged after the acknowledgment for $P_i(k)$ has been completed.

The $P_i(k)$'s replace the $M_i(j)$'s from the non-FIFO algorithm. Since messages arrive in order of increasing sequence numbers, an AP only has to store two batches as the entire unacknowledged message list: $P_i(k)$ is the acknowledgment being submitted in round robin fashion, and $B_i(k)$ will be the next batch acknowledged in round robin fashion. Therefore, the memory requirement for the unacknowledged lists will be bounded by a constant. This is likely to be a significant improvement in the amount of memory used at the AP level as compared to the amount of memory that we expect will be used in the non-FIFO case.

The procedures `SENDER` and `RECEIVER` are identical to those in the non-FIFO case. The only differences in the procedures `NEW_GVT` and `NEW_MSG` occur because the unacknowledged message list and $M_i(j)$'s are replaced by $P_i(k)$'s and $B_i(k)$'s.

```
SENDER:     IF    ρ′ has been sent to this LP
            THEN  IF    ρ′ is in any outstanding message list
                  THEN  Remove all outstanding messages up to and
                        including ρ′ message;
                        IF    timestamp (ρ′) = υ_i
                        THEN  υ_i := smallest timestamp in any
                              outstanding message list;

RECEIVER:   IF    T_i = 0                          -- timeslice is over
            THEN  s := next sender;
                  T_i := T_RR;
                  ρ_i := P_i(s);
            ELSE  T_i := T_i - 1;

NEW_GVT:    IF    mode = GVT mode                  -- resume round robin
            THEN  mode := round robin;
                  T_i := T_RR;
            FOR   each sender r
                  IF    GVT > timestamp (P_i(r))
                  THEN  P_i(r) := B_i(r);
                        B_i(r) := {∞, Φ, 0} ;
                  IF    ρ_i = old P_i(r)           -- continue round robin
                  THEN  IF    P_i(r) ≠ {∞, Φ, 0}
                        THEN  ρ_i := P_i(r);
                        ELSE  T_i := 0;            -- force timeslice end
                  IF    timestamp (P_i(r)) = GVT
                  THEN  mode := GVT mode;          -- suspend round robin
                        ρ_i := P_i(r);

NEW_MSG:    r := sender(new_msg);
            IF    (P_i(r) = {∞, Φ, 0} ) OR
                  (timestamp(new_msg) < timestamp(P_i(r)))
            THEN  P_i(r) := batch containing new_msg;
                  IF    ρ_i = old P_i(r)
                  THEN  ρ_i := P_i(r);
            ELSE  IF    timestamp (B_i(r)) > timestamp (new_msg)
                  THEN  timestamp (B_i(r)) := timestamp (new_msg);
                        size (B_i(r)) := size(B_i(r)) + 1;
                  ELSE  size(B_i(r)) := size(B_i(r)) + 1;
```

Figure 4.8 Acknowledgment Algorithms Assuming FIFO Channels Between LP's.

4.5. Proof of Correctness of SPA

A proof of correctness for an aggressive PDES synchronization protocol executing on the framework hardware was presented by Srinivasan and Reynolds [SRRE93]. The

correctness proof states that (1) $GVT_c(t)$ is always less than or equal to $GVT_a(t)$ at all real times *t*; and (2) $GVT_c(t)$ approaches $GVT_a(t)$, or that if the simulation is halted at any real time t_0, $GVT_c(t)$ will equal $GVT_a(t_0)$ in some finite time, $t \geq t_0$. We prove the correctness of SPA here.

There are two modes for acknowledging messages in SPA: round robin mode and GVT mode. The round robin mode coupled with lost state vectors on the output side of the reduction network makes no guarantee that *any* round robin acknowledgment sent through the PRN is ever read and processed by the AP of the LP sending the message. Therefore, we can only prove that messages are acknowledged in GVT mode, since there is no guarantee that any messages are acknowledged during round robin mode. We expect the round robin mode to enhance the efficiency of SPA. In Section 4.5.4., we prove that round robin mode maintains the correctness of SPA.

Our correctness proofs assume a processor pair consisting of a host processor and an auxiliary processor as described in Chapter 3. The event execution is performed on the host processors, GVT computation algorithms and SPA execute on auxiliary processors, and all globally reduced values are computed on state vectors in a reduction network.

4.5.1. Properties of the Hardware and Algorithms

Before proving the correctness of the single phase acknowledgment, we present some properties of the framework hardware and corresponding algorithms. Some of these properties appeared first in [SRRE93].

4.5.1.1. Properties of the Framework Hardware

Property P1 (*no loss*):
No communication from the HP to AP is lost.

Property P2 (*reduction operation*):

The PRN computes reductions on state vectors. Acknowledgment *T*-values are computed with a minimum operation that is a tagged selective operation.

4.5.1.2. Properties of the AP

Property P3 (*periodic read*):

Each auxiliary processor will read the output from the reduction network in a finite, bounded amount of time.

4.5.1.3. Properties of the AP Algorithm

Property P4 (*correctly set local clock*):

When LP_i completes an event, receives a message or antimessage, or rolls back, the *T*-value σ_i at AP_i is set correctly to reflect its local clock.

Property P5 (*correctly maintained unreceived message time*):

When LP_i sends an event message or antimessage, if υ_i is greater than the timestamp of the message, υ_i will be set to the timestamp of the message by AP_i. When AP_i reads an acknowledgment ρ' from the reduction network and processes this as an acknowledgment for a message LP_i has sent, that message is removed from one of LP_i's outstanding message lists and υ_i is set to the smallest among the timestamps of messages remaining in LP_i's outstanding message lists. At any time, υ_i is always equal to the smallest timestamp in LP_i's outstanding message lists.

Property P6 (*correctly updated unacknowledged message lists*):

When LP_1 receives an event message or antimessage, that message is added to an unacknowledged message list at AP_i.

Property P7 (*computed GVT*):

By definition, computed GVT at real time *t*, $GVT_c(t)$, is the minimum of all local clocks $\sigma'(t)$ and minimum of all unreceived message times $\upsilon'(t)$ and will always be set to either $\sigma'(t)$ or $\upsilon'(t)$.

4.5.1.4. Properties of SPA

Property P8 (*unique timestamps*):

All *T*-values that are inputs to the reduction network have unique logical timestamps. Therefore, given a set of inputs, there is only *one* possible computed output that can emerge from a tagged selective reduction in the reduction network.

Property P9 (*garbage collection*):

When AP_i reads and processes a new state vector at real time t, indicating that $GVT_c(t) > GVT_c(t-\Delta t)$, where $GVT_c(t-\Delta t)$ was the last GVT value processed by AP_i, it removes all messages with timestamps less than $GVT_c(t)$ from its unacknowledged message lists and updates the batches to be acknowledged in each list, including ρ_i, within finite real time.

Property P10 (*GVT mode acknowledgment*):

When AP_i determines that it has a message to acknowledge with timestamp equal to $GVT_c(t)$, it enters GVT mode and continues to acknowledge this message until some real time $t+\Delta t$ such that $GVT_c(t) < GVT_c(t+\Delta t)$. If AP_i enters GVT mode at real time t and at real time $t+\Delta t$, $GVT_c(t) < GVT_c(t+\Delta t)$, and there is no message in an unacknowledged message list for LP_i with timestamp equal to $GVT_c(t+\Delta t)$, then AP_i resumes round robin mode and acknowledges another message.

4.5.2. Overview of Lemma 4.1: $GVT_c(t)$ Is Monotonically Non-decreasing As a Function of Real Time *t*

Since $GVT_c(t)$ is a commitment horizon for the garbage collection of unacknowledged messages, it is critical that the function $GVT_c(t)$ is monotonically non-decreasing as a function of t and that it never exceeds $GVT_a(t)$. Note that if ever $GVT_c(t+\Delta t) < GVT_c(t)$, an AP may determine incorrectly that a message acknowledgment was read and processed by the sender and remove it from an unacknowledged message list. Hence, $GVT_c(t)$ would never advance, in particular it would never increase beyond the time of that message acknowledgment. Before proving the correctness of SPA, we must show that $GVT_c(t)$ is a non-decreasing function of real time t. This is the goal of Lemma 4.1.

Since $GVT_c(t)$ is the computed GVT that emerges from the reduction network, it follows that $GVT_c(t) = GVT_p(t-\Delta t)$ for some $\Delta t > 0$. This follows directly from definitions as depicted in Figure 4.5. If we show that $GVT_p(t)$ is monotonically non-decreasing for all times t when a reduction cycle is started, then $GVT_c(t)$ will be monotonically non-decreasing for all real times t. We show this.

Srinivasan and Reynolds [SRRE93] showed that $GVT_p(t)$ is strictly non-decreasing when a single processor, and not a host-auxiliary processor pair, was used to execute simulation events and interface with the reduction network. The asynchronous nature of the auxiliary processors and the FIFO's between the processors in a HP-AP pair make our proof more complex. We build on the proofs of Srinivasan and Reynolds [SRRE93] and use similar proof techniques. The methods for maintaining the local *T*-values σ_i and υ_i and the computation of both σ' and υ' do not change in this algorithm. The difference between TPA, proven correct in [SRRE93], and SPA, which we prove correct next, is the method of acknowledging messages.

In the following proof the only assumption made about acknowledgments is that there is a mechanism for the receiver of each message to notify the sender of the receipt of the message. There is no assumption that acknowledgments use a reduction network or any particular algorithms.

Lemma 4.1

> $GVT_c(t)$, which is the minimum of all local clocks, $\sigma'(t)$, and minimum of all unreceived message times $\upsilon'(t)$ at all real times *t*, is monotonically non-decreasing as a function of *t*; i.e., $GVT_c(t) \leq GVT_c(t+\Delta t)$, $\forall \Delta t \geq 0$.

Proof

> If there is a change in $GVT_p(t)$ from one reduction cycle ($GVT_p(t_0) = G$) to the next ($GVT_p(t_0+\delta) = \hat{G}$), where δ is the time it takes to complete a reduction cycle (refer to Figure 4.5), then we must show that this change is always nondecreasing. We refer to *T*-values contributing to $\hat{G}$ using the ˆ symbol and to those contributing to *G* without it. We consider the two cases — change to local clock or change to unreceived message time — in which $GVT_p(t)$ can be computed. For each case, there are two sub-cases. In other words, there are four ways to transition from *G* to $\hat{G}$.
>
> Case I. $\hat{G} = \hat{\sigma}_i$ for some *i*
>
> (*a*) AP_i processes a new local clock value (`NEW_CLOCK`) from the FIFO indicating that LP_i has finished processing an event and computed $\hat{\sigma}_i$. $\hat{\sigma}_i$ must be at least as large as σ_i since the events list is sorted in non-decreasing order. Therefore, $G \leq \sigma_i \leq \hat{\sigma}_i = \hat{G}$.

(*b*) AP_i processes a new local clock value (`NEW_CLOCK`) from the FIFO indicating that LP_i has received a straggler (an event message arriving in an LP_i's past) or antimessage from LP_j, causing a rollback. In this case, $\hat{\sigma}_i < \sigma_i$. Since both messages and antimessages are used to compute unreceived message times, we examine the possible scenarios:

(*i*) AP_j, the auxiliary processor for the sending LP_j, has processed the FIFO entry `SENT_MSG` for the message sent with timestamp $\hat{\sigma}_i$. Therefore, $G \leq \upsilon_j = \hat{\sigma}_i = \hat{G}$.

(*ii*) AP_j has not processed the FIFO entry `SENT_MSG`. In [SRRE93], it was shown that if a rollback chain (or possibly several rollback chains) are followed towards the root of the chain (or the root of the smallest rollback chain), there exists an AP_k such that $\mu_k \leq \hat{\sigma}_i$. (The LP at the root of the rollback chain is the LP that has rolled back its computation due to a straggler and not an antimessage. If LP_i receives a straggler and then sends an event message that is a straggler at LP_j, then there are two rollback chains, where LP_i and LP_j are both roots.) Therefore, $G \leq \mu_k \leq \hat{\sigma}_i = \hat{G}$.

Case II. $\hat{G} = \hat{\upsilon}_i$ for some *i*

(*a*) AP_i reads and processes an acknowledgment for the message with time υ_i, and therefore sets its new unreceived message time $\hat{\upsilon}_i$ to the smallest timestamp in its outstanding message list. By **P5** (correctly maintained unreceived message time), $\upsilon_i \leq \hat{\upsilon}_i$, and since $G \leq \upsilon_i$, $G \leq \hat{G}$.

(*b*) AP_i processes a FIFO entry `SENT_MSG`, such that the timestamp of the message sent $< \upsilon_i$. In other words, $\hat{\upsilon}_i < \upsilon_i$. However, since a message is sent only after executing an event and enqueueing a `LOCAL_CLOCK` entry, $\hat{\upsilon}_i = \sigma_i$. Since $G \leq \sigma_i$ by Case I., and $\hat{\upsilon}_i = \hat{G}$, $G \leq \hat{G}$.

Therefore, **Lemma 4.1** (monotonically non-decreasing $GVT_c(t)$) holds at all times, if there is a mechanism for LP's receiving messages to notify the LP's that sent them. ■

4.5.3. Overview of Theorem 4.1: GVT_c approaches GVT_a

Acknowledging messages is the key to the progress of the globally reduced unreceived message time, υ', and therefore $GVT_c(t)$. In SPA, $GVT_c(t)$ serves as the "handshake acknowledgment", such that a message acknowledgment is guaranteed to have been read and processed by the message sender when $GVT_c(t)$ is greater than the logical timestamp of the message.

Next we show that SPA acknowledges messages correctly and that if the simulation makes progress, GVT_c increases as messages are acknowledged, i.e. $GVT_c(t) < GVT_c(t+\Delta t)$ in some finite real time Δt, for all real times t as long as there are messages to be acknowledged or events to be processed. In Lemma 4.2 we show that a message acknowledgment with timestamp equal to $GVT_c(t)$ will eventually be *completed*. An acknowledgment is completed in SPA when AP_s, the sender of the message, reads ρ', removes the message from an outstanding message list, and updates υ_s accordingly. In Lemma 4.4 we show that $GVT_c(t)$ advances as messages are acknowledged. Finally, we show that $GVT_c(t)$ *approaches* $GVT_a(t)$ (Theorem 4.1). Theorem 4.1 is a liveness proof, to show that if the simulation is halted at any real time t, $GVT_c(t+\Delta t)$ will equal $GVT_a(t)$ in some finite real time Δt. We note that if the simulation is halted, some messages will remain unacknowledged since the progress of $GVT_c(t)$ is a commitment horizon for messages acknowledged.

Lemma 4.2

If $timestamp(\rho_r(t)) = GVT_c(t)$, then the acknowledgment of the unique message with timestamp ρ_r will complete in finite real time Δt.

Proof

By **P1** (no loss), each AP receives information about the receipt of each event message and antimessage from its corresponding HP. By **P6** (correctly updated unacknowledged message lists), each message will be incorporated into the unacknowledged message list of the message receiver.

We have assumed $GVT_c(t)$ is equal to the unique timestamp of an unacknowledged message, which means it also must be equal to the (same) timestamp of an outstanding message for some LP_s. Since $GVT_c(t)$ is the minimum of all outstanding messages and all local clocks by **P7** (computed GVT), it follows that $GVT_c(t) = \upsilon'(t)$. Therefore, $GVT_c(t) = \upsilon'(t) = \upsilon_s(t) = \rho_r(t)$.

By **P5** (correctly maintained unreceived message time), the message with timestamp $\upsilon_s(t)$ does not have a corresponding completed acknowledgment. By **P8** (unique timestamps) and **P5** (correctly maintained unreceived message time), $GVT_c(t) = \upsilon_s(t)$ until the acknowledgment is read and processed by AP_s and AP_s has updated υ_s.

Now we establish within finite time Δt, $\rho'(t+\Delta t) = \rho_r(t)$. By **P9** (garbage collection) and **P3** (periodic read), each AP_i, i=1…n, removes all messages with timestamps less than $GVT_c(t)$ from its unacknowledged message lists and updates its ρ_i, so, timestamp($\rho_r(t)$) < timestamp($\rho_i(t_1)$), i=1…n, $i \neq r$ by some real time t_1, $t_1 > t$. By **P2** (reduction operation), a new state vector will be computed by some real time t_2, $t_2 > t_1$, and $\rho'(t_2)$ will be set to $\rho_r(t)$. By **P10** (GVT mode acknowledgment) and **P8** (unique timestamps) $\rho'(t_2)$ retains its value until sometime after $GVT_c(t)$ advances.

By **P3** (periodic read) by some real time t_3, $t_3 > t_2$, AP_s will read and process the state vector containing $\rho'(t_2)$, the acknowledgment for its smallest outstanding message. By **P5** (correctly maintained unreceived message time), AP_s will update υ_s in finite real time, so that by real time t_4, $t_4 > t_3$, $\upsilon_s(t_4) > \upsilon_s(t)$.

Hence, if timestamp($\rho_r(t)$) = $GVT_c(t)$, the acknowledgment of ρ_r will complete in finite real time Δt, where $\Delta t = t_4 - t$. ■

Lemma 4.3

If $GVT_c(t)$ = timestamp($\rho'(t)$), then in some finite real time Δt, $GVT_c(t+\Delta t) > GVT_c(t)$.

Proof

Assume that $GVT_c(t)$ = timestamp($\rho'(t)$), indicating that the message with timestamp at $GVT_c(t)$ is being acknowledged. AP_s, the auxiliary processor for LP_s, such that s = sender($\rho'(t)$), will eventually process the acknowledgment by Lemma 4.1. AP_s will then increase its minimum unreceived message time υ_s, which was equal to $GVT_c(t)$.

By **P2** (reduction operation), a new state vector will be computed by some real time t_1, $t_1 > t$. $GVT_c(t_1) \neq GVT_c(t)$ by **P8** (unique timestamps) and the change to υ_s. Furthermore, by **Lemma 4.1** (monotonically non-decreasing GVT_c), $GVT_c(t_1) > GVT_c(t)$.

Therefore, $GVT_c(t+\Delta t) > GVT_c(t)$ in some finite real time Δt, where $\Delta t = t_1 - t$, if $GVT_c(t)$ = timestamp($\rho'(t)$). ■

Theorem 4.1

If the simulation is halted at real time t, $GVT_c(t+\Delta t) = GVT_a(t)$, in some finite real time Δt.

Proof

Assume that the simulation is halted at real time t, meaning $GVT_a(t)$ is fixed at real time t. At time t, all LP's stop processing events and retain the values of their $\bar{\sigma}_i$'s. With a reliable host communication network, all messages in transit will be received eventually. At real time t there exists $\bar{\sigma}_j(t)$, such that $\bar{\sigma}_j(t)$ is the smallest local clock in the system and because the simulation is halted, $GVT_a(t) = \bar{\sigma}_j(t)$.

By **P4** (correctly set local clock), by some real time t_1, $t_1 > t$, each AP will process all entries in the HP-AP FIFO, and $\sigma_i(t_1) = \bar{\sigma}_i(t)$, $\forall i$, i=1…n.

There are a finite number of messages to be acknowledged. By **Lemma 4.1** and **Lemma 4.2** for each $\upsilon_i < \bar{\sigma}_j$, i =1…n, the messages will be acknowledged, and the corresponding υ_i's will be set to greater values, and by these same lemmas, $GVT_c(t_2)$ will equal the value of $\bar{\sigma}_j(t)$ by some real time t_2, $t_2 > t_1$.

Hence, $GVT_c(t+\Delta t) = GVT_a(t)$, in some finite real time Δt, where $\Delta t = t_2 - t$. ■

4.5.4. Correctness of Round Robin Acknowledgments

Now that we have shown the correctness of the GVT mode acknowledgments in SPA, we must show that the round robin mode acknowledgments maintain the advancement of $GVT_c(t)$, and that the correctness of SPA is maintained during transitions from round robin mode to GVT mode and vice versa. In Lemma 4.4 we prove that all messages acknowledged during round robin mode are acknowledged properly. In Lemma 4.5 we show the *non-interference* of round robin acknowledgments. We use the term non-interference in a less rigorous way than Owicki and Gries [OwGr76]. By non-interference, we will show that each acknowledgment mode does not violate the correctness of the other. Our strategy is to examine the variables and data structures that are read and/or written while in round robin acknowledgment mode and to show that they will remain in correct states. Also we demonstrate that the transitions between the two acknowledgment modes do not violate the correctness of Lemma 4.2, Lemma 4.4, and Theorem 4.1.

Lemma 4.4

Message acknowledgments completed in round robin mode maintain the correctness of GVT mode acknowledgments.

Proof

During round robin acknowledgment mode, AP_r will write different $M_r(j)$'s to the T-value ρ_r. By **P6** (correctly updated unacknowledged message lists) each message in an unacknowledged list at AP_r at real time t, including the $M_r(j)$'s, has been received by LP_r.

By **P5** (correctly maintained unreceived message time) if LP_s, such that s = sender($\rho'(t)$), reads and processes acknowledgment $\rho'(t)$ during round robin mode at real time t, then LP_s updates $\upsilon_s(t)$ accordingly.

Therefore, any message acknowledgment that completes in round robin mode will maintain the correctness of GVT mode acknowledgments. ■

We note that any acknowledgment completed during round robin mode has a timestamp greater than or equal to $GVT_c(t)$ by **P9** (garbage collection). Furthermore, by this same property**,** an unacknowledged message is not removed from an unacknowledged message list unless $GVT_c(t)$ is greater than its timestamp. This is important so that ρ_i's with timestamps less than $GVT_c(t)$ do not prevent ρ_i's with timestamps greater than or equal to $GVT_c(t)$ from being completed. Next we prove that the transitions between the two modes do not affect the correctness.

Lemma 4.5

The transition from round robin acknowledgment mode to GVT acknowledgment mode is non-interfering.

Proof

Consider the transition from round robin mode to GVT mode.

Assume AP_r is operating in round robin acknowledgment mode at real time t.

By **P10** (GVT mode acknowledgment) AP_r will remain in round robin acknowledgment mode as long as no message in its unacknowledged message list has a timestamp equal to $GVT_c(t)$.

Assume at some real time t_1, $t_1 > t$, $GVT_c(t) < GVT_c(t_1)$. Assume that LP_s has the smallest outstanding message time in the system, $\upsilon'(t_1) = \upsilon_s(t_1) = GVT_c(t_1)$, and that LP_r was the receiver of this message. By **P3** (periodic read) and **P9** (garbage collection) AP_r will read a new state vector from the reduction network and process $GVT_c(t_1)$ by some real time t_2, $t_2 > t_1$. By **P10** (GVT mode acknowledgment) all round robin mode acknowledgments are halted by AP_r and GVT acknowledgment mode is entered, such that timestamp($\rho_r(t_2)$) = $GVT_c(t_1)$. By **Lemma 4.2** the acknowledgment will complete by some real time t_3, $t_3 > t_2$. By **Lemma 4.4**, $GVT_c(t_1) < GVT_c(t_4)$, by some real time t_4, $t_4 > t_3$, once GVT acknowledgment mode is entered.

Hence, the transition from round robin acknowledgment mode to GVT acknowledgment mode does not prevent the smallest message in the system from being acknowledged. Therefore, the transition to GVT mode is non-interfering. ■

Lemma 4.6

The transition from GVT acknowledgment mode to round robin acknowledgment mode is non-interfering.

Proof

Consider the transition from GVT acknowledgment mode to round robin acknowledgment mode.

Assume AP_r is operating in GVT acknowledgment mode at real time t.

Assume that at some real time t_1, $t_1 > t$, $GVT_c(t) < GVT_c(t_1)$. By **P3** (periodic read) and **P9** (garbage collection) AP_r will read a new state vector from the reduction network and will process $GVT_c(t_1)$ by some real time t_2, $t_2 > t_1$. By **P9** (garbage collection) AP_r will remove acknowledgments which have times less than $GVT_c(t_1)$, so that old acknowledgment values will not interfere with acknowledgments to be performed in the future.

Furthermore, by **P10** (GVT mode acknowledgment) AP_r resumes round robin mode when it determines it no longer needs to acknowledge the smallest message in the system. Hence, the transition from GVT acknowledgment mode to round robin acknowledgment mode does not prevent future messages from being acknowledged.

Therefore, the transition from GVT acknowledgment mode to round robin acknowledgment mode is non-interfering. ■

Theorem 4.2

The single phase acknowledgment is correct.

Proof

The correctness of the single phase acknowledgment follows directly from **Theorem 4.1**, **Lemma 4.4**, **Lemma 4.5**, and **Lemma 4.6**. ■

4.6. Improvements of the Acknowledgment Algorithms

The race condition described in Section 4.1.3. can be eliminated with additional communication between an AP and its corresponding HP. Once an AP reads the change to the next event time from the HP, processes this received message by adding it to the appropriate unacknowledged message list, and submits the corresponding next event time to the reduction network, resulting from a received message, it communicates this to the HP. Only at that time can the HP send the acknowledgment message through the host

communication network. This is essentially a handshake between the host processor and auxiliary processor in order to eliminate the race condition. Therefore, the race condition is eliminated by submitting the *T*-values in the receiving LP to the reduction network prior to sending an acknowledgment.

If this synchronization between the two processors is implemented, the host communication network can be used to acknowledge *some* of the messages in the system. A host processor can submit some acknowledgments to the reduction network and others to the host communication network. Furthermore, if an HP employs piggybacking of batched acknowledgments to event messages, no additional message traffic is generated in the host network. Piggybacking can only be used if a pair of LP's have a bidirectional communication, as in Figure 4.1. The performance of using both networks for message acknowledgments is a topic of future research.

4.7. Discussion

Each of the algorithms, TPA and SPA, presented here requires each LP to maintain two lists of outstanding messages; Reynolds's framework algorithms only required LP's to maintain a list of messages sent. The size of these data structures, i.e., the memory requirement, is largely dependent on the properties of an LP, the PDES synchronization protocol used, and the application. All lists are maintained on the auxiliary processor and not the host processor. We expect the size of the unacknowledged message lists in the non-FIFO SPA to grow faster than the unacknowledged message lists in TPA because garbage collection will only occur with the advance of GVT_c. In TPA messages are removed from the unacknowledged list when a handshake completes. We present performance results comparing these two algorithms with respect to memory requirements in Section 4.8.

All three of our proposed alternatives depend on a new global synchronization value ρ', which replaces the acknowledgment messages in Reynolds's framework algorithms. All algorithms eliminate the potential race condition in Reynolds's algorithm, although in the previous section we showed how to use the host network while eliminating this race condition. Both the two-phase acknowledgment and the single phase (round robin) acknowledgment guarantee the correctness of PDES synchronization algorithms even when output values from the PRN are overwritten prior to being read. In the next section we present our results on the performance of both TPA and SPA.

4.8. Performance Results

We have implemented both TPA and SPA on our four-processor prototype framework hardware [REPS93]. We discuss the prototype system and execution parameters prior to presenting our results.

4.8.1. Prototype Framework Hardware

The host system is a Sparc cluster: four Sparc 2 equivalent processors with Ethernet (TCP/IP) as the host communication network. The expected host communication latency time is approximately two milliseconds.

Each auxiliary processor is a 25 MHz Motorola 68020 microprocessor with 256 Kbytes of RAM. The host-auxiliary processor interface is implemented with a dual-ported RAM, where a Sparc 2 accesses the dual-ported RAM through a Sun SBus interface [SBUS90].

The parallel reduction network consists of three ALU's in a binary tree-shaped network. The minor cycle time is 150 nanoseconds. The pipelining in the reduction network is performed synchronously.

The prototype hardware limits state vectors to size eight; each of the eight components is a register pair, one 32-bit data register and one 32-bit tag register. For both acknowledgment algorithms, we have programmed the reduction network to operate on state vectors of size four. We discuss the implementation details next.

4.8.2. Implementation of Acknowledgment Algorithms

Both TPA and SPA were implemented. TPA was implemented exactly as discussed in Section 4.3. In lieu of guaranteeing unique timestamps in the implementation of SPA, we used TPA as a simple and efficient way to break a deadlock situation as discussed in Section 4.4.2. We assume the non-FIFO case for SPA, where messages at LP_r are not necessarily received in the order they were sent from LP_s.

Time Warp was selected as the parallel simulation synchronization protocol. The communication topology of the LP's was a fully connected graph, i.e., each LP sent a message with equal probability to one of the three other LP's. Prior to gathering results we varied the global virtual time between 1000 and 20,000 to verify that the statistics would be gathered during stable conditions in the simulation. In each result reported, the termination condition was that global virtual time exceeded 20,000.

In SPA, we performed a sensitivity analysis to the variable T_{RR}, the number of iterations that a round robin acknowledgment is submitted by AP_i to the PRN as ρ_i. We observed little sensitivity to this variable. It is an open question whether larger simulations are more sensitive to this value. For SPA we selected $T_{RR} = 25$ since this value gave slightly better timings.

Each point reported on a graph indicates the average of eight executions of a simulation with the same event time, same delay to save state, and the same number of internal events generated per output event (event resulting in a message send). We gathered

results on the size of the outstanding and unacknowledged message lists from each of the four auxiliary processors twice. We varied the event delay, the delay to save state, and the number of internal events generated per output event. Our results follow.

4.8.3. Results of Experiments

We gathered the following statistics for each run:

- mean length of the unacknowledged message lists at an AP
- mean length of the outstanding message lists at an AP
- maximum sizes of batches of acknowledgments
- wallclock time to execute the simulation
- number of messages (event messages and antimessages) received by an LP

We estimated the mean time between received messages by dividing the wallclock time of the simulation by the number of messages received at an LP. The mean time between received messages reflects that load on the auxiliary processor, so we use this time as the independent variable to present our results graphically. In all graphs the results of TPA studies are represented with circles on the curves, and the results of SPA studies are represented with inverted triangles on the curves.

The graphs in Figures 4.9 and 4.10 show that the execution times of the simulations are essentially the same for both acknowledgment algorithms. In Figure 4.9 the number of internal events between output events (those that generate an event message) was uniform randomly distributed between 0 and 10. In Figure 4.10 the number of internal events between output events was uniform randomly distributed at 2. The general shape of the curves is linear, as expected. As the mean time between messages decreases, the execution time of the simulation decreases. Note that as the mean time between messages decreases (causing the auxiliary processor load to increase), the execution time becomes asymptotic to the cost of doing message transmission and processing. At this point the host

communication network is saturated (message arrival rate ≥ network message processing rate).

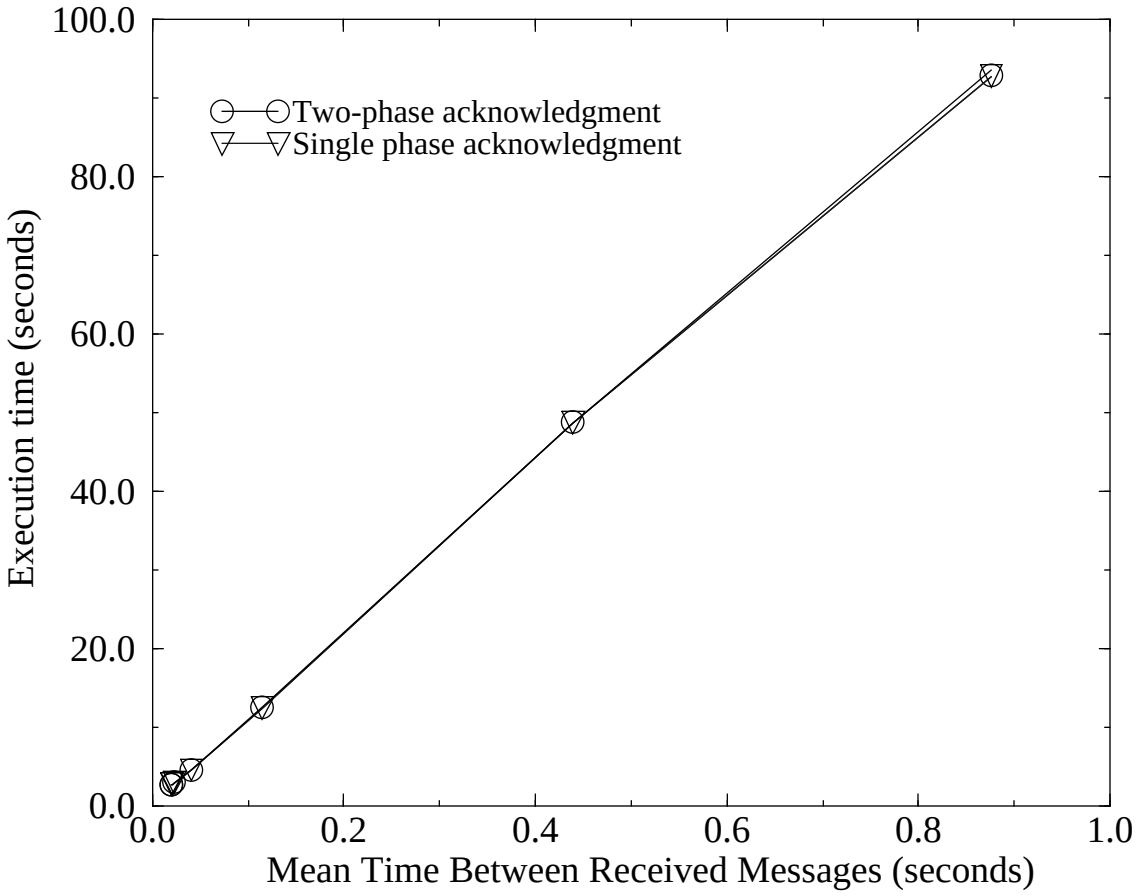


Figure 4.9 Effect of Load on Execution Time, where Number of Internal Events Is 10.

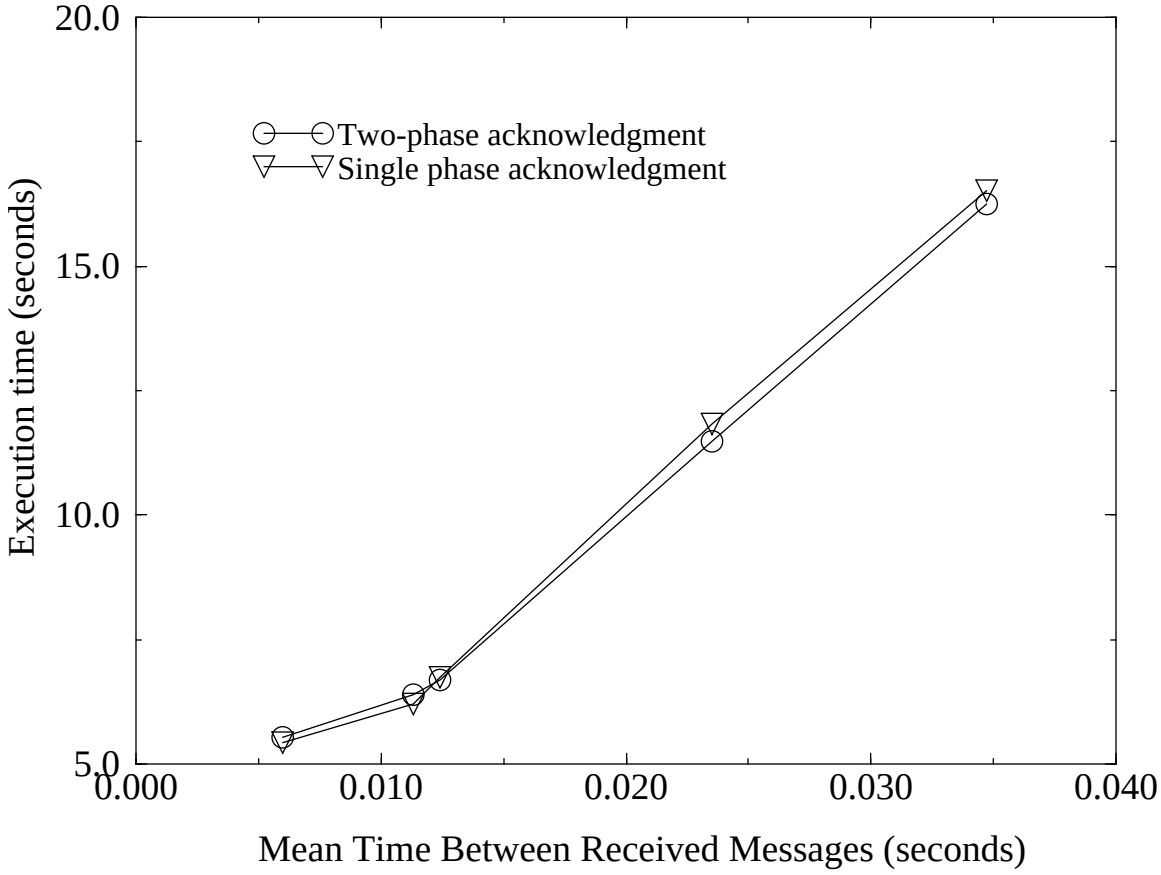


Figure 4.10 Effect of Load on Execution Time, where Number of Internal Events Is 2.

The graphs in Figures 4.11 and 4.12 show the effect of the auxiliary processor load on both the size of the unacknowledged message list and the size of the outstanding message list. The size of the unacknowledged message list in TPA is essentially zero at all times. This suggests that when AP_i receives an entry for a newly received message at its host processor, that message will be submitted to the reduction network as ρ_i immediately. For a given LP in TPA, the size of its outstanding message list is larger than its unacknowledged list because a message remains in the outstanding message list from the

time a message is sent through the host communication network until that message is received and acknowledged.

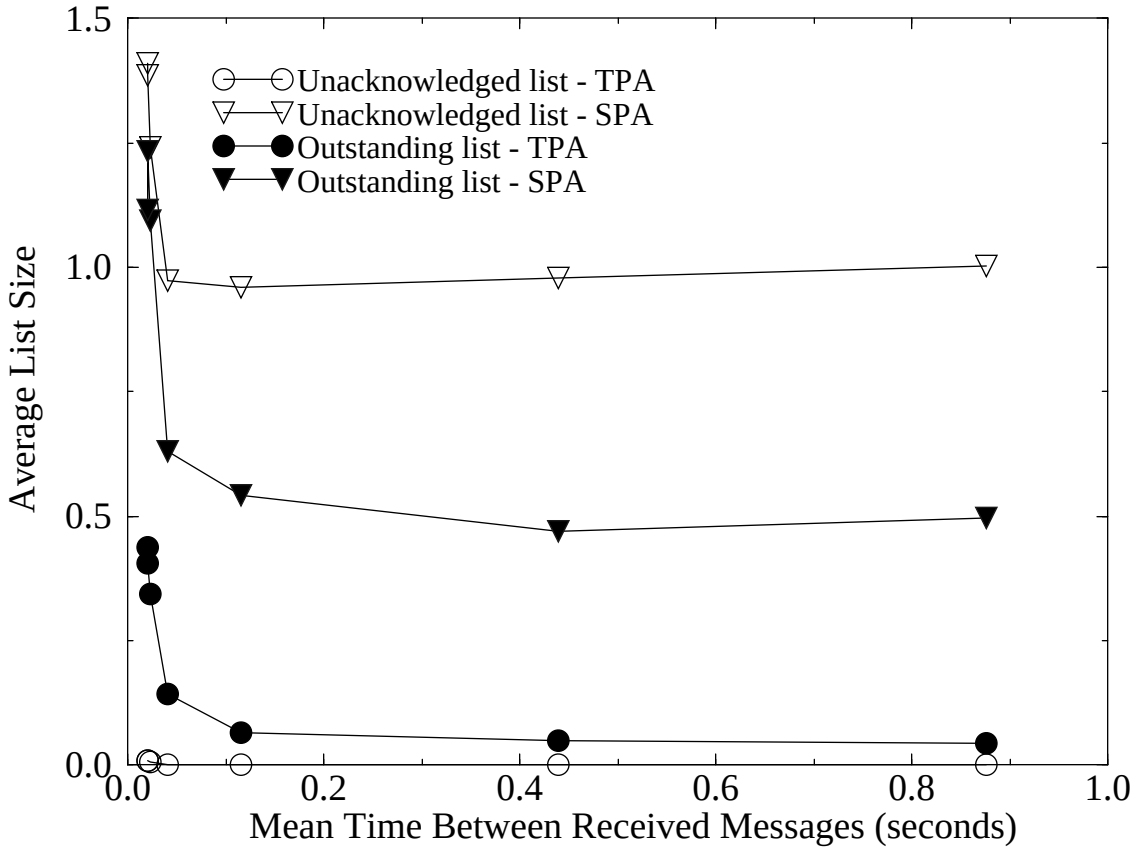


Figure 4.11 Effect of Load on Sizes of Lists, where Number of Internal Events Is 10.

In SPA the unacknowledged message list is smaller than the outstanding message list. A message is removed from the outstanding message list when an acknowledgment is received in the reduction network. A message is not removed from the unacknowledged message list until global virtual time has increased to a time that is greater than the message's timestamp. In all cases, the total sizes of both the unacknowledged message list and the outstanding message list in SPA are significantly greater than the sizes of the corresponding lists in TPA. We explore the effects of load on the maximum batch size next.

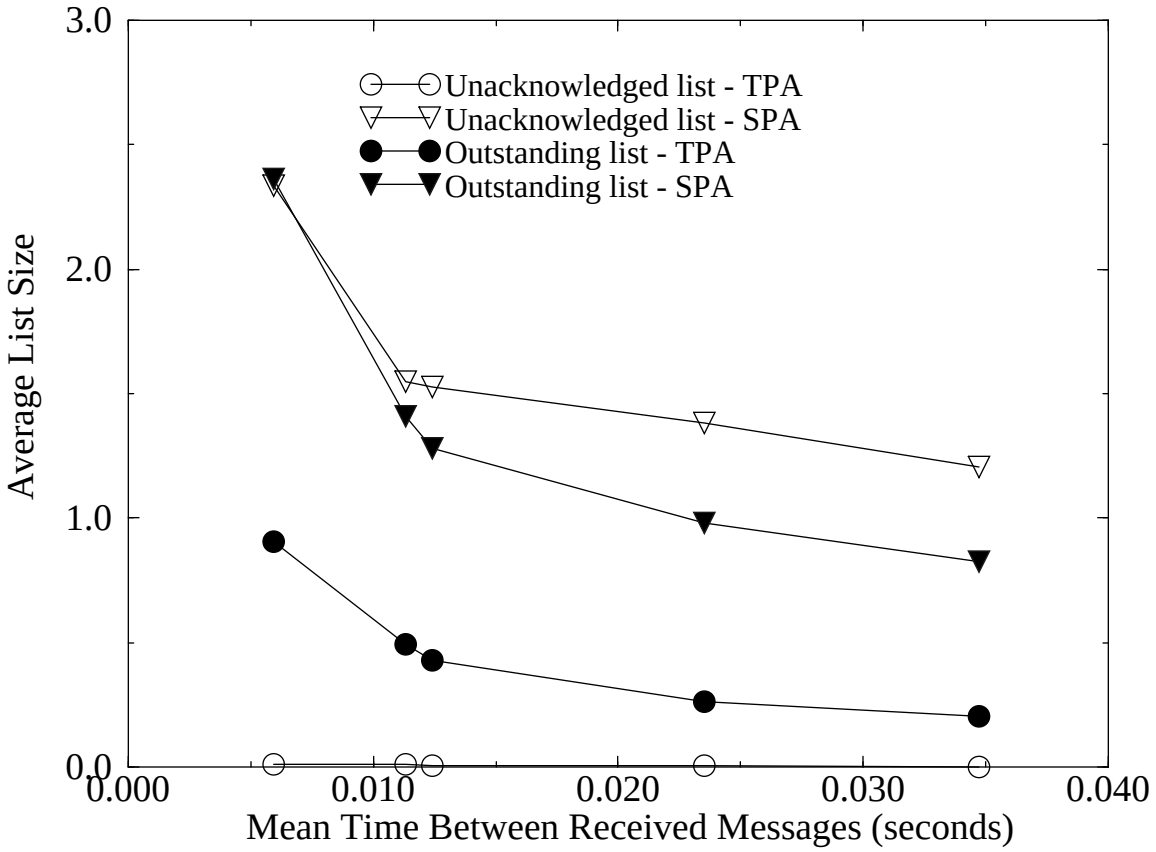


Figure 4.12 Effect of Load on Sizes of Lists, where Number of Internal Events Is 2.

The graphs in Figures 4.13 and 4.14 show the effect of load on the maximum batch size of an acknowledgment in the reduction network. It comes as no surprise that the batch sizes are larger in SPA than in TPA. In TPA, the maximum size of batches is less than three in all cases and less than two in the case when the number of internal events is uniform randomly distributed from 0-10 (Figure 4.13). SPA, however, allows a message to be coalesced with a batched acknowledgment in progress, as long as the timestamp of the message is less than that of the acknowledgment in progress. This allows larger batches to

form. In most cases the sizes of the maximum batches are three times greater in SPA than in TPA.

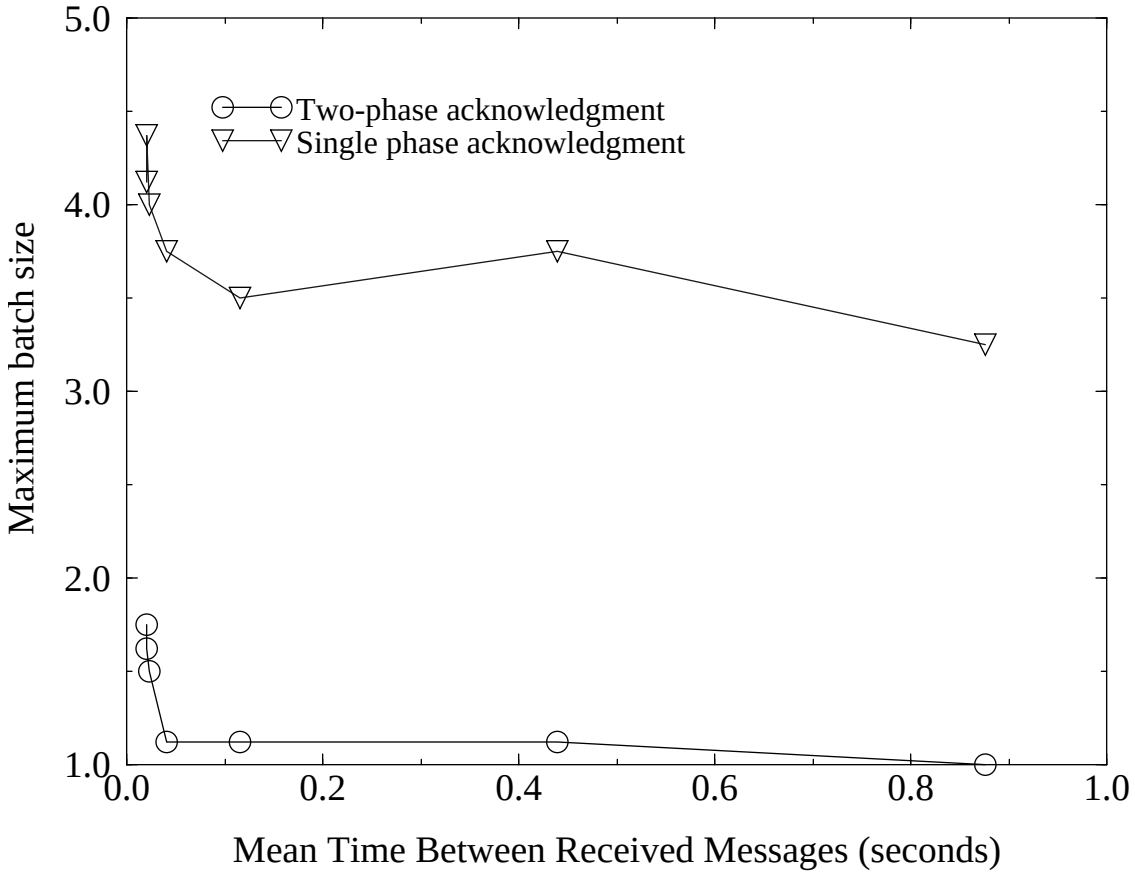


Figure 4.13 Effect of Load on Batch Size, where Number of Internal Events Is 10.

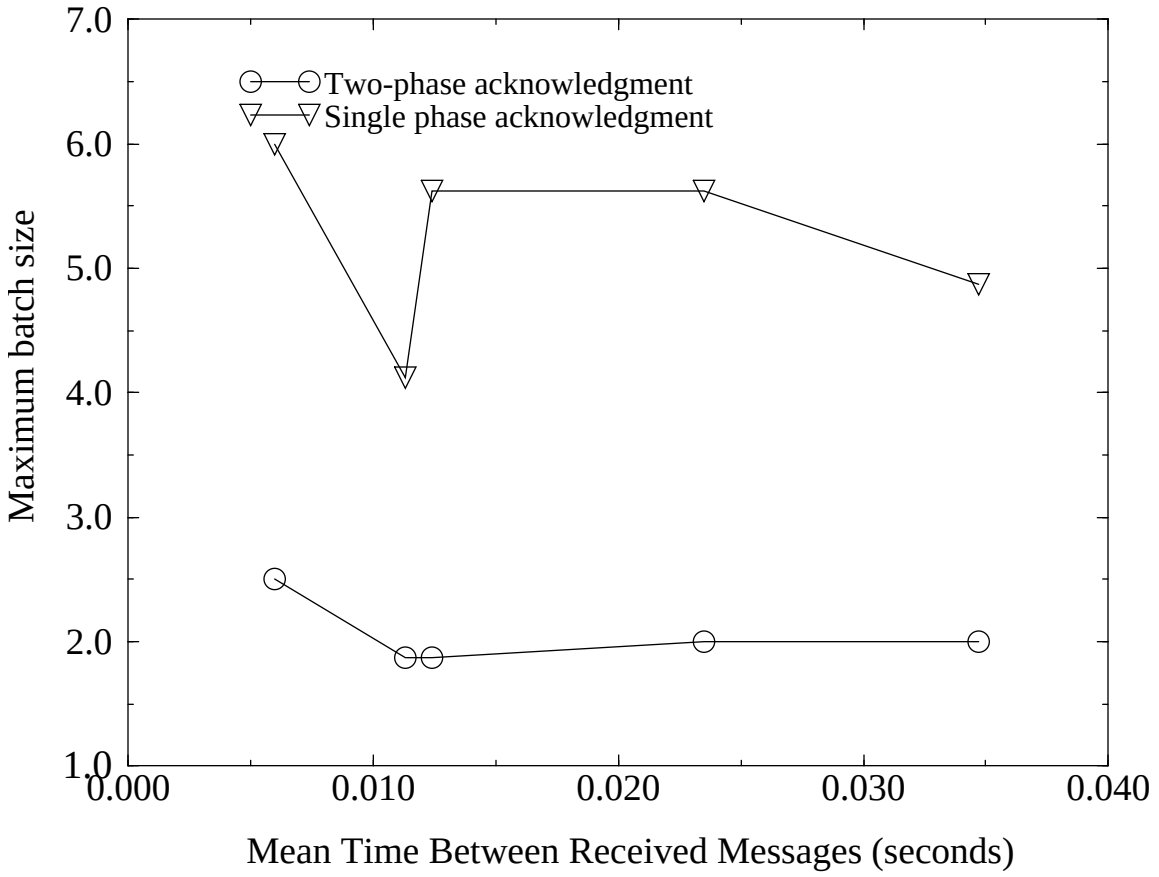


Figure 4.14 Effect of Load on Batch Size, where Number of Internal Events Is 2.

We conclude, based on our performance results, that TPA gives the same performance as SPA, using less memory on the auxiliary processor. It is an open question how each acknowledgment algorithm will perform in a larger system. The simulation results in [SRIN92], however, indicate that TPA is scalable to up to 32 processors with essentially no growth in the time required to acknowledge messages.

4.9. Summary and Conclusions

In this chapter we have presented several algorithmic variations on acknowledging event messages in a reduction network. We have made several contributions in this area.

First, we have demonstrated the feasibility of performing message acknowledgments in the reduction network. The algorithms are instrumental in particular to the computation of global virtual time in aggressive PDES synchronization protocols and minimum outstanding message times in other PDES synchronization protocols in the reduction-based framework developed in Chapter 3.

Second, we have presented batching acknowledgments as a method of acknowledging multiple messages in a single reduction, and this improvement has proven to be robust and stable. With the batching of acknowledgments, the framework hardware is able to efficiently support smaller event granules. Throughout this chapter we have discussed the fundamental issues involved with each alternative. We have included discussions about implementation details and correctness issues.

Third, we have developed two algorithms which correctly acknowledge messages in a reduction network where output state vector loss is a property of the hardware: a two-phase acknowledgment and a single phase acknowledgment. Furthermore, the algorithms are correct when the reductions are being computed asynchronously with the execution of the simulation. A correctness proof for TPA was presented in [SRRE93], and correctness proofs for SPA were presented here.

Finally, we have implemented both the two-phase acknowledgment and the single phase acknowledgment on our prototype framework hardware. We compared the two with respect to execution time of the simulation and sizes of the message lists on auxiliary

processor. We conclude that TPA performs as well as SPA, and that memory utilization on the auxiliary processors is better.

The simulation performance studies presented in [SRIN92] and performance studies presented in Section 4.8. are encouraging. They show significant potential for reduction-based acknowledgment algorithms. These experimental results, however, assume that reductions are computed globally. In the next chapter we introduce target-specific reductions to more accurately depict the state of a PDES. We believe the performance of the message acknowledgment algorithms in this chapter can be improved greatly since target-specific acknowledgments allow many acknowledgments to occur concurrently. We expect the time lag between the sending of a message and its acknowledgment to be reduced significantly.

5 The Cost of Doing Target-specific Reductions

We have established that target-specific reductions can be critical to the performance of both aggressive and non-aggressive parallel simulations [PARE93]. In this chapter we present the best known theoretical results on the sequential computation of target-specific binary, associative operations. Our theoretical contributions include the establishment of specific time complexity and space complexity trade-offs. These complexities are important to the computation of reduced values in a target-specific reduction network because they can be used to estimate the cost of computing target-specific reductions in parallel.

We begin this chapter by motivating the need for the computation of target-specific reductions in parallel discrete event simulations. We demonstrate the applicability of target-specific reductions to a wide range of PDES synchronization protocols in Section 5.1. In Section 5.2. we discuss characteristics of binary, associative operations that have an effect on the computation of target-specific reductions. In Section 5.3. we present a problem formulation and relate its graph theoretical representation to an equivalent set theoretic one. In Sections 5.4 and 5.5 we present sequential solutions to this problem. These solutions differ in time and space complexities and the obvious trade-offs between the two. Intuitively, the best sequential solution to this problem has time complexity $O(n^2)$; however, the results in this chapter show that a sub-quadratic solution is attainable. In Section 5.6. we discuss the implementation of a parallel target-specific reduction network.

5.1. Target-specific Reductions in Parallel Simulations

The dissemination of global state information in the form of globally reduced values means that each of the LP's in a parallel simulation receives information that is derived from the whole group's inputs. In a *target-specific* reduction, each of the LP's is an individual *target*, such that it receives a particular reduced value computed from a subset of the LP's on which it is *logically dependent*. In a system of *n* LP's, *n* target-specific reductions must be computed for each operation. Each of the necessary reduction operations in a PDES may have different targets. Thus, from the perspective of a given LP, its target-specific inputs and outputs depend on the operation being performed. Also, different LP's can have different sources of inputs and different targets, as we explore next.

5.1.1. Target-specific Reductions in Conservative PDES's

A conservative PDES synchronization protocol based on global state information can introduce artificial dependencies that may not exist, causing potentially parallel events to execute sequentially. For example, assume the communication graph of LP's in Figure 5.1. LP_1 is not dependent on any LP's and should be able to advance its simulation clock without blocking. LP_2 should receive synchronization information only from LP_1, LP_5 should only receive synchronization information from LP_1 and LP_3, etc. In this case, target-specific next event times and target-specific unreceived message times for LP_i are computed from those LP's that can have an impact on its performance, that is, all of LP_i's predecessors in the directed graph representing the communication topology of the PDES.

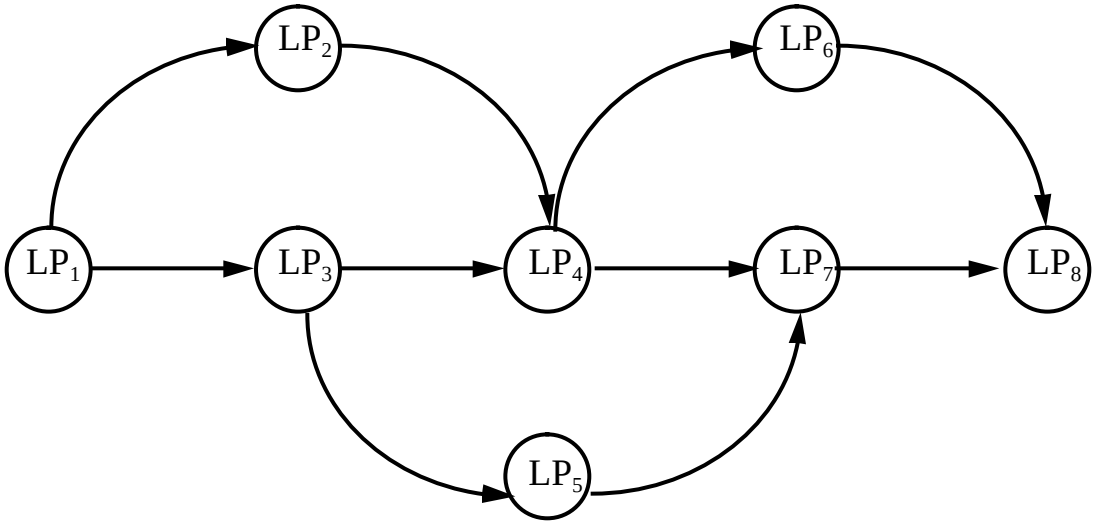


Figure 5.1 An Example PDES Communication Topology.

Many PDES's exhibit static communication properties: that is, the number of LP's and the communication topology are known a priori. Furthermore, many have partially static topologies, where the sets of *potential* predecessors to each LP are known a priori. A conservative PDES with these properties shows significant runtime speedup if the rapid dissemination of target-specific synchronization information is possible. (See Chapter 6.) By providing the dissemination of target-specific next event times, target-specific unreceived message times, and target-specific lookahead values to a conservative PDES, decisions about safe processing will be based on more accurate information. These target-specific values are computed from an LP's predecessors as determined by the transitive closure of the static communication graph. By providing information specific to each LP's requirements, the potential for increased parallelism and a resulting speedup is apparent. We report on speedup potential in conservative PDES in Chapter 6.

5.1.2. Target-specific Reductions in Optimistic PDES's

We introduce a new value, *target-specific virtual time (TSVT)* to be computed for LP's in an optimistic PDES. Target-specific virtual time is a relative value: $TSVT_i$ is the minimum logical timestamp to which LP_i, $i = 1, 2, \ldots, n$, can roll back, and it is computed from input values based on the transitive closure of the communication graph. If there are n LP's in a simulation, then n separate TSVT's must be computed. The TSVT for two separate LP's will only be computed in the same way if both predecessor sets are identical.

Definition. Target-specific virtual time for LP_i, $TSVT_i,(t)$ at real time t, is the minimum of the virtual times in (1) the logical clocks of predecessors of LP_i based on the transitive closure of the communication graph at time t, and (2) all messages that have been sent by LP_i's predecessors but have not yet been processed by real time t.

Figure 5.2 shows smallest unreceived message times (υ_i's), local clocks (σ_i's), and computed $TSVT_i$'s at an instance of real time in a parallel simulation. The nodes in the communication graph are shaded to show the nodes which have the same TSVT.

Since $TSVT_i$ is customized for each LP_i, it more accurately reflects the state information on which to base event processing decisions. It is more accurate than GVT. Therefore, fossil collection can be done with more accurate state knowledge. This can lead to better utilization of state saving memory. (See Chapter 6 for performance results on the reduction in state space.) If the state space were limited, as it is in Fujimoto's high-speed rollback chip [FUTG92], this can be a significant benefit.

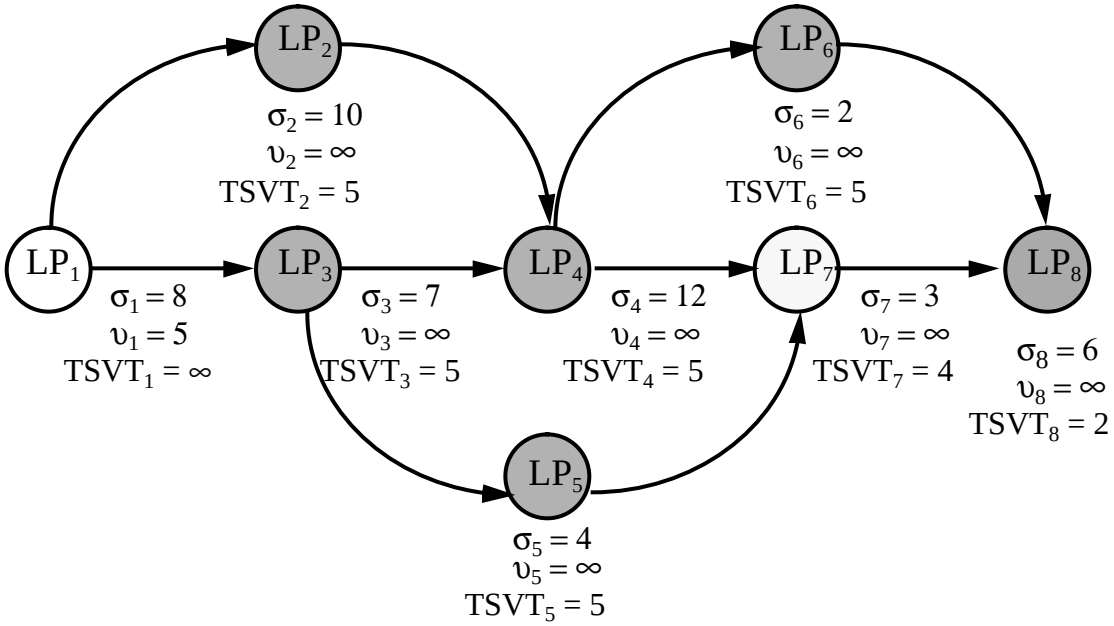


Figure 5.2 An Instance of an Optimistic PDES.

The efficient computation of target-specific virtual times, i.e., in a high-speed reduction network, can provide near-perfect state information at a low cost. This can be important, for example, to the cancelback protocol of [JEFF90], a memory management protocol for Time Warp, executing on a shared memory multiprocessor. Performance studies [DAFU93] have shown that the global computation of GVT on the Kendall Square Research Machine (KSR) [KEND92] in support of the cancelback protocol has a high cost.

5.1.3. Target-specific Acknowledgment of Messages in PDES's

In Chapter 4 we discussed how globally computed minimum operations support message acknowledgments in a PDES. All reduced values and message acknowledgment algorithms can be modified for target-specific message acknowledgments. Target-specific acknowledgments have the same target as messages: a target-specific acknowledgment of a message for a given LP is computed using the inputs from only *immediate* predecessors

to that LP, those LP's which send it messages. For example, in Figure 5.1, the only LP which is an immediate predecessor to LP_5 is LP_3. Consequently, a target-specific handshake acknowledgment is computed using the inputs from only *immediate successors*. In the same figure, LP_4's immediate successor's are LP_6 and LP_7. Based on simulations presented in Chapter 6, we have strong reason to believe that the performance of all acknowledgment algorithms will improve if global reductions are replaced with target-specific reductions because the acknowledgments are done on a per LP basis.

5.1.4. Other Target-specific Reductions in PDES's

We expect the computation and dissemination of target-specific state information to benefit other PDES synchronization protocols as well. For example, target-specific ceiling or fault values can support windowing synchronization protocols. New windowing protocols are likely to arise with the rapid and efficient computation of target-specific reductions.

Finally, we expect the dissemination of near-perfect state information to support adaptive PDES synchronization protocols, those that combine the strengths of both aggressive and non-aggressive protocols while limiting the weaknesses. This is a topic of current research [SRIN93].

If target-specific synchronization information is available to LP's in a PDES, all LP's receive more accurate state information and can process events accordingly. The final result is a framework for PDES that efficiently supports a wide range of PDES's.

5.1.5. Target-Specific Reductions in Other Parallel Computing Applications

The benefits of the computation and dissemination of target-specific reduced values are not limited to parallel discrete event simulations. We expect target-specific reductions to enhance a range of parallel computing problems: load balancing [KIRK92], iterative

numerical computations, and parallel programming problems that require the computation of binary, associative operations across *irregular* communication patterns. The impact of the efficient computation target-specific reductions in parallel computations is a topic of future research.

5.2. Problem Characteristics

All binary, associate operations (minimum, maximum, addition, logical OR, etc.) do *not* have the same characteristics. We discuss differences which are pertinent to the computation of target-specific reductions.

Both addition and multiplication have *inverse operations*; subtraction is the inverse operation to addition and division is the inverse operation to multiplication. Some binary associative operations (minimum, maximum, logical OR, and logical AND, for example) are *persistent*; in other words, an operand can be included in the operation one or more times without changing the result of the operation. The operations of addition and multiplication and the computation of logical XOR are not persistent.

A final characteristic of the binary, associative operations minimum and maximum is that they are *comparison-based* operations. This suggests that sorting algorithms may be instrumental in algorithms which compute comparison-based target-specific reductions.

The theoretical results we present in the next sections assume that the target-specific binary, associative operations are comparison-based. Therefore, we narrow the class of reductions to include only those which are comparison-based, i.e., minimum and maximum operations. We are not concerned with this limitation because most of the computations required in a parallel discrete event simulation are comparison-based: smallest outstanding message time, minimum logical clock, and message acknowledgments all require the computation of minimums.

5.3. Target-Specific Reduction Problem Definition

Several applications, including parallel discrete event simulation, require the computation of *n* minimum (or maximum) values of any *n* subsets of *n* dynamically changing numbers. In PDES, we view this problem as a graph theoretical one, since a communication graph, as in Figure 5.1, shows the relationship of LP's. We redefine this problem as a set theoretical one in order to describe our solution algorithms.

Problem Definition. Given a set of *n* numbers and *n* subsets of those *n* numbers, the target-specific minimum values are computed as *n* minimum values, one minimum value per set.

Each of the *n* numbers are input values to reduction operations (input values from an LP to a reduction), each subset represents the set of input values to one target-specific reduction (a target-specific reduction for one LP is based on inputs from its predecessors in the directed graph), and the computed minimums are the results of the *n* target-specific reductions (a computed output for each LP).

5.3.1. Upper Bound of the Target-Specific Minimum Value Problem

Clearly, an upper bound on this problem is $O(n^2)$. A simple $O(n^2)$ algorithm to compute the *n* target-specific minimum values examines each set, and for each set, it compares the elements in that set to find the minimum value. Since there are *n* sets with at most *n* elements each, then at most n^2 values need to be compared. Hence, the upper bound time complexity is $O(n^2)$. Space complexity is also $O(n^2)$ since it takes $O(n^2)$ space to store the *n* sets. We note that a large number of graphs do not have this worst case complexity. For example, if either the total number of arcs is $O(n \log n)$ or the maximum number of input arcs is limited to $O(\log n)$ per node, then the time complexity to compute *n* target-specific values is $O(n \log n)$. The $O(n^2)$ is simply worst case. Our goal was to find an

O(n log n) time solution to this particular problem. Before presenting our current solutions, we give a detailed problem discussion.

5.3.2. An Equivalent Problem

In order to visualize the problem, consider the set of *n* numbers to be a set of *bucket references* $\{b_1, b_2, \ldots, b_n\}$, such that each bucket contains a value. This adds a level of indirect referencing to the *n* inputs; for example, the input value from LP_1 is always contained in bucket b_1. There are *n* subsets of the bucket set, labeled S_1 through S_n. Each subset is essentially a set of references to the buckets. An instance of this problem is shown pictorially in Figure 5.3. To solve this problem the minimum bucket value in each set must be computed. The solution is a vector of *n* numbers, where each number is a value contained in a bucket.

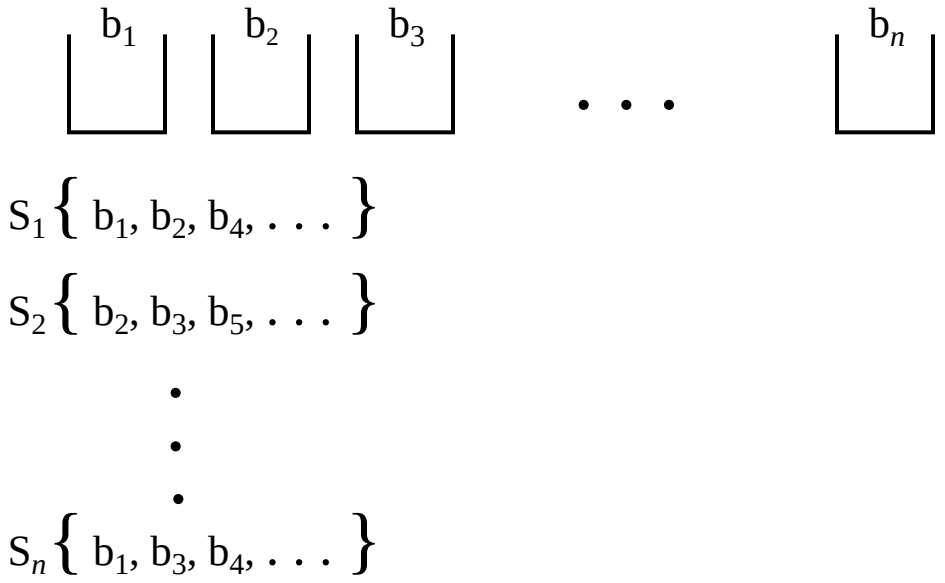


Figure 5.3 An Instance of the Minimum Value in All Subsets Problem.

Therefore, an equivalent problem to the target-specific reduction problem can be defined as follows:

Minimum Value in All Subsets (MVAS) Problem. Given a set B of buckets, $B = \{b_1, b_2, \ldots, b_n\}$, and n subsets of these buckets, $S_1, S_2, \ldots, S_n$, find the solution vector $M = \{m_1, m_2, \ldots, m_n\}$, such that m_i is the minimum value across all buckets in set S_i.

The contents, or bucket references, of the subsets are known prior to the computation of the minimums. Bucket references are static for each graph topology. Therefore, we allow preprocessing to be done to the subsets prior to the computation of the minimums. The preprocessing is *not* part of the execution time of the sequential algorithm. In the solutions in Section 5.4.and Section 5.5., preprocessing reduces the ultimate time complexity of an algorithm to solve the MVAS problem. In the next section we present a solution to the target-specific reduction problem with the best time complexity we have found.

5.4. A *O(n log n)* Time Solution

We present the necessary preprocessing and an *O(n log n)* time algorithm for the MVAS problem.

5.4.1. Preprocessing

Our first observation is that there are *n!* possible permutations, or sorted orders, of the *n* buckets. For each permutation there is an associated solution vector to the MVAS problem. This suggests that there are at most *n!* possible solutions.

During preprocessing, each of the *n!* solution vectors are stored in a table in memory, such that a solution vector can be addressed by its permutation. We assume that a permutation can be treated as an *n*-word address to the associated solution vector. We envision a memory hierarchy, as depicted in Figure 5.4, such that each word in the *n*-word address refers to a different memory bank. The time complexity of the algorithm will take at least $O(n \cdot n!)$ since each memory location must be initialized. This preprocessing will

not affect the runtime of the algorithm. In Section 5.4.4. we show that the memory can be reduced to $O(2^n)$; however, for now we assume the entire permutation table to simplify our description of this solution.

5.4.2. General Algorithm

Assume that an instance of the MVAS problem is viewed as in Figure 5.3. In other words, the subsets are references to the buckets. The first step of the solution is to sort the buckets in nondecreasing order using a computationally efficient sorting algorithm.

The second step is to use the sorted permutation as an address to find the solution vector of bucket references in the table. Figure 5.4 shows how a memory hierarchy can support an *n*-word address. Once the *n*-word address is referenced, a pre-stored solution vector of bucket references is located.

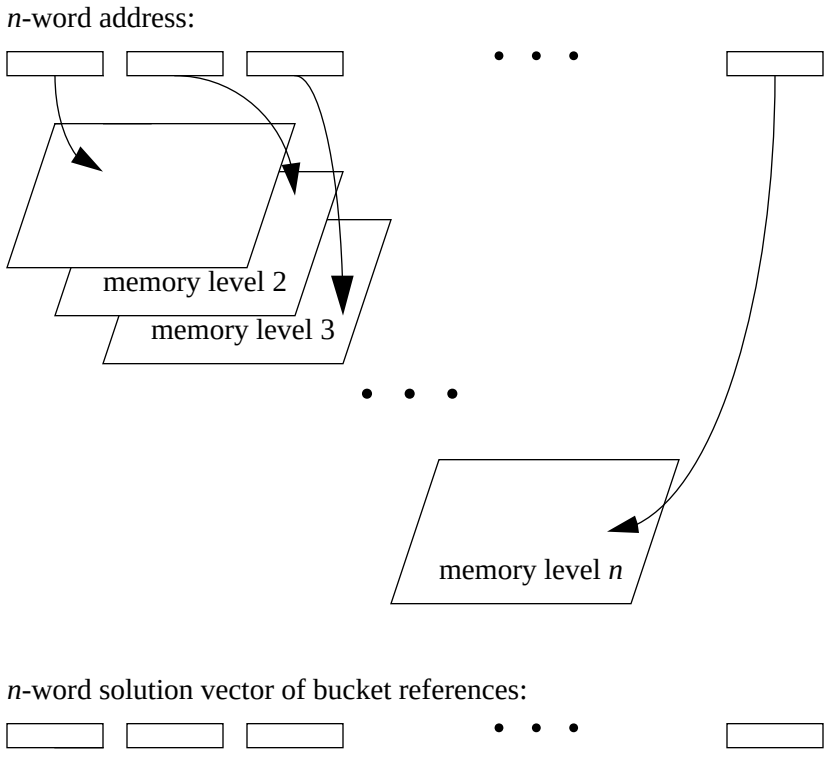


Figure 5.4 Memory Requirements of an O(*n log n*) Solution to the MVAS Problem.

The third step of the algorithm is the use of the bucket reference vector to assign actual values from the buckets to the vector of minimums. The result of this step of the algorithm is a vector of n values, each is the minimum value in the respective set.

5.4.3. Time Complexity Analysis

We analyze each of the three steps in the algorithm. The time to sort the n buckets is $O(n \log n)$. The time to address the solution vector of bucket references is $O(n)$, since it takes a unit time to read each word in the n-word address. Finally, the time to assign a value to each set is $O(n)$ since the contents of a bucket can be read in unit time, and there are n bucket references in the solution vector. Therefore, the time complexity of the algorithm is dominated by the sort, and the total time complexity to solve the MVAS problem is $O(n \log n)$.

5.4.4. Space Complexity Analysis

We analyze the total space needed to solve the problem using both word complexity and bit complexity. The table of possible solution vectors has $n!$ entries. Each entry is an n-word solution vector. So, the space complexity is $O(n! \cdot n)$ words. Each word in memory has width of $\log n$ bits. Therefore, the total space complexity in bits is $O(n! \cdot n \log n)$. We recognize that this is a super-exponential amount of space and unacceptable.

We now show how to reduce the space requirements of the algorithm, employing well known divide and conquer techniques. We show that the space complexity can be reduced to an exponential space complexity instead of a super-exponential space complexity and the time complexity will remain $O(n \log n)$.

Assume that both the sets and the buckets are partitioned into n/k groups of k subsets by n/k groups of k buckets, where $1 \leq k \leq n$. This gives us $(n/k)^2$ subproblems to solve using the same algorithm which was used to solve the large problem. There are $k!$ permutations

of *k* buckets. During the preprocessing, a table of the solution vectors of bucket references of the *k* subsets is created for each of the *k!* permutations for one subproblem. Figure 5.5 shows how the problem is partitioned. We assume that each partial solution vector of bucket references can be addressed by its permutation.

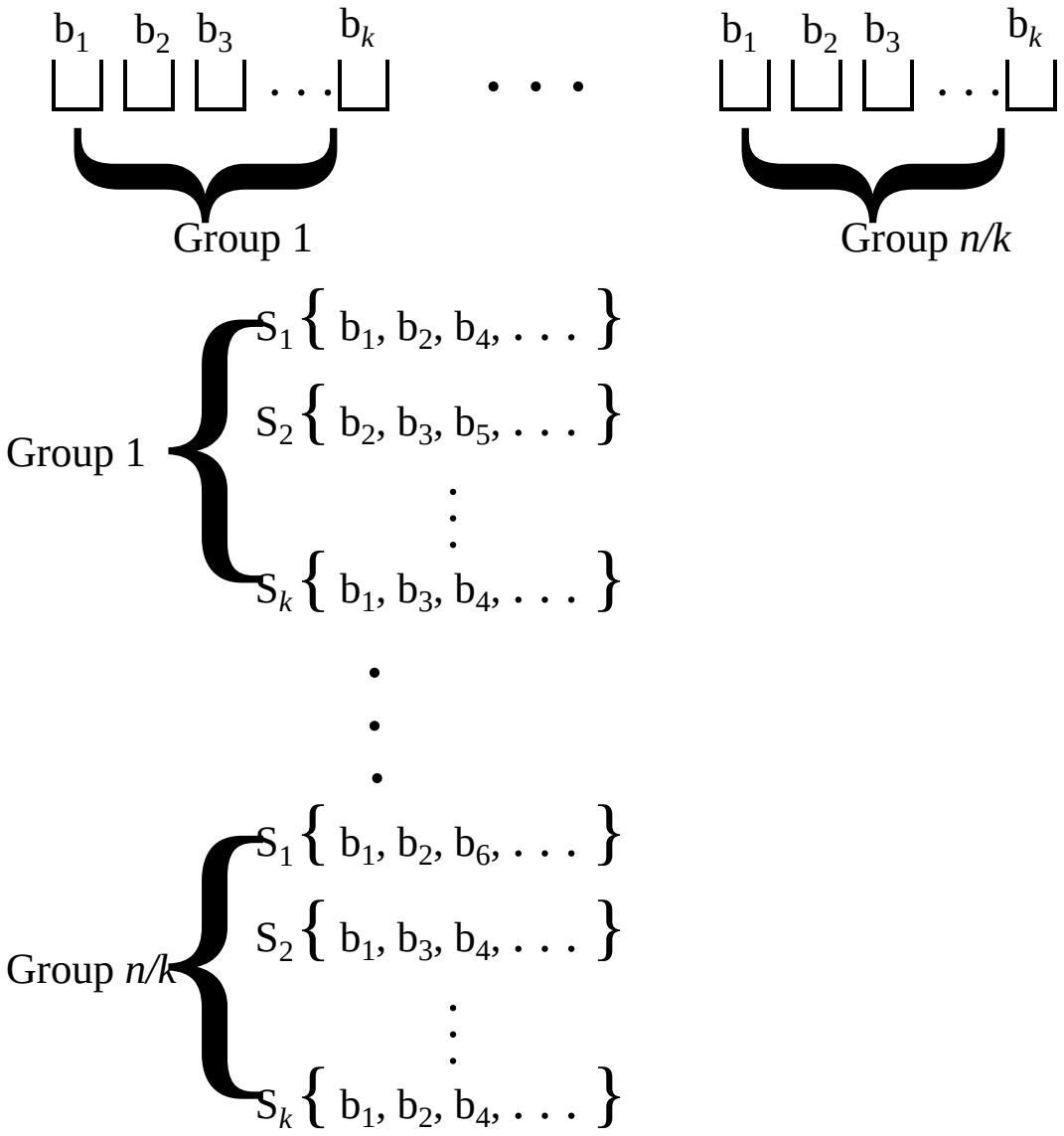


Figure 5.5 Divide-and-Conquer Partitioning of MVAS Problem.

There are four steps in the divide-and-conquer algorithm, where the first three steps are similar to the larger problem solution and the final step resolves the solution of the smaller subproblems to get a solution to the entire problem. The first step in the execution of the algorithm is to sort each of the n/k groups of k buckets. Second, for each subproblem of k subsets and k buckets, the permutation of the sorted k subsets is used as a k-word address to access a solution vector of bucket references for the k subsets. Third, a *partial* minimum value is assigned to each of the k subsets for that subproblem; this value is the minimum value for that subset across k buckets. The final step in the algorithm is to combine the partial minimum values of the subproblems to give a solution to the large problem. So, for each of the n subsets, the minimum of its n/k partial minimums is computed, one for each group of buckets.

It is possible to keep the time complexity of the divide-and-conquer solution at $O(n \log n)$ if k is selected carefully. We explain this now. The time to sort one group of k buckets is $O(k \log k)$. Therefore, the time to sort n/k groups is $n/k \cdot k \log k$ or $O(n \log k)$. The time to address the solution vector of bucket references is $O(k)$ for one subproblem, since it takes unit time to read each word in the k-word address. Since there are $(n/k)^2$ subproblems, the total time to address the solution vectors is $O(n^2/k)$. Similarly, the time complexity to assign minimum values to each *of* k sets is $O(k)$ for one subproblem since the contents of a bucket can be read in unit time; the time complexity to do the assignments for all $(n/k)^2$ subproblems is $O(n^2/k)$. Finally, the time to compute the minimums of n/k partial values for all n subsets will be $O(n^2/k)$. Therefore, the time complexity of the algorithm is dominated by $O(n^2/k)$, so the total time complexity to solve the MVAS problem with a divide-and-conquer algorithm is $O(n^2/k)$.

We analyze the total space needed to solve the problem using bit complexity. For each subproblem of k buckets and k subsets, the table of possible solution vectors has $k!$

entries. Each entry is an *k*-word solution vector. Each word in memory has a width of *log k* bits. There are $(n/k)^2$ subproblems. Therefore, the total space complexity to solve all subproblems is $O(k! \cdot (n^2/k) \cdot \log k)$ bits.

The selection of *k* is critical to the time and space complexities of the algorithm. If we assume that $k = n/\log n$, there are *log n* groups of $n/\log n$ buckets by $n/\log n$ subsets. The time complexity to solve the problem is $O(n \log n)$. This does not change.

The space complexity to solve the problem is $O(n \log n \cdot \log(n/\log n) \cdot (n/\log n)!)$ bits. The dominating component of this will obviously be $(n/\log n)!$. Since it is well known that $n! \leq n^n$, and both have the same order complexity, we make the observation that $(n/\log n)! \leq n^{n/\log n}$. Furthermore, $n^{n/\log n} = (2^{\log n})^{n/\log n} = 2^n$. Hence, the space complexity is $O(2^n)$ bits, a reduction from super-exponential space to exponential space.

We are optimistic about this result because of the $O(n \log n)$ time complexity though we are aware of the practical considerations of the exponential space requirements. We next present a family of solutions to this problem. Several members of the family reduce the space complexity to polynomial space with cost of the time complexity increasing. We note, however, that the time complexity remains sub-quadratic.

5.5. A Family of Solutions to the Target-specific Dissemination Problem

We assume the same set theoretical MVAS problem for this family of solutions. In order to facilitate our algorithm discussion, we view the sets as sets of pointers to the buckets as in Figure 5.6.

The first step of the solution is to once again sort the buckets in nondecreasing order of the bucket contents. During the execution of the sort, the pointers are dragged along as the elements are put in place.

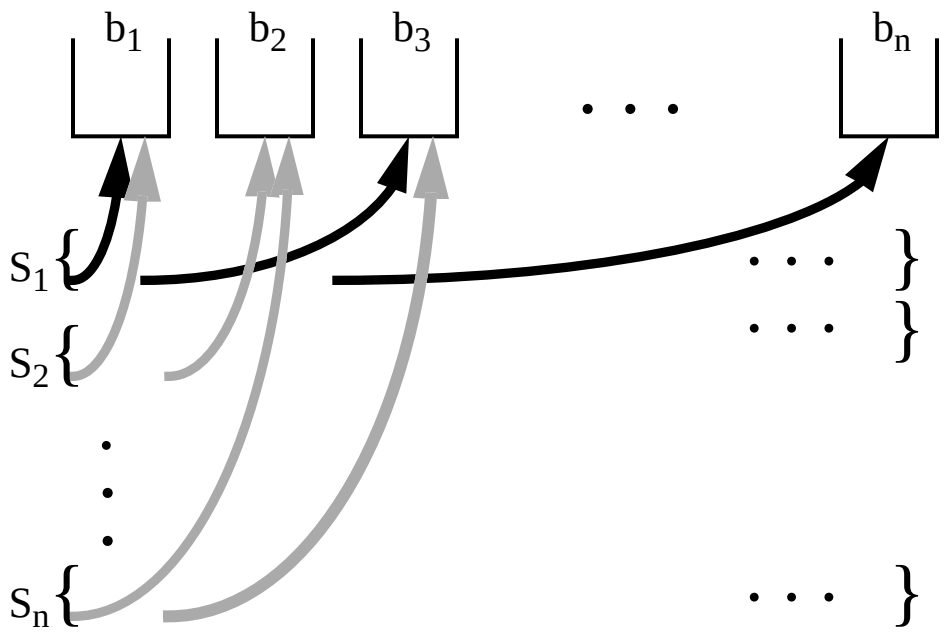


Figure 5.6 An Instance of the Minimum Value in All Subsets Problem Assuming Pointers.

After the sort has completed, the second step is to select the minimum value in each set S_i, $1 \leq i \leq n$, by finding the set S_i's pointer which points to the smallest bucket value in the sorted list. The algorithm begins with the first bucket, that bucket with the smallest value, and assigns the bucket value to the minimum values of all sets with a pointer into the bucket. Once a value has been assigned to m_i, all pointers from the set, i.e., all pointers of the set's color, are removed from consideration. For example, assume the buckets in Figure 5.6 have been sorted. The minimum value m_1 of set S_1 will be the value in bucket b_1. Figure 5.7 depicts the removal of set S_1 once the value of m_1 has been resolved. The buckets are followed in increasing order and minimum values are assigned in this way until all minimum values have been assigned values.

We describe the algorithm, associated data structures, and necessary preprocessing to accomplish this solution in sub-quadratic time in the following section.

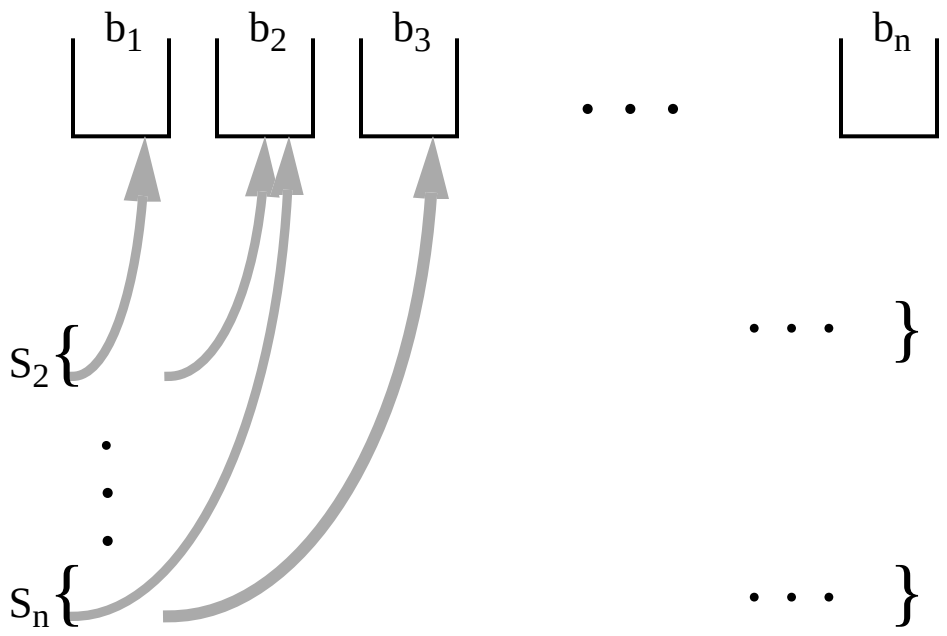


Figure 5.7 After Minimum Value Assigned to Set S_1.

5.5.1. Solution Algorithm

We begin by discussing the necessary preprocessing and data structures in our solution algorithm. Recall that the contents of each subset $S_1, \ldots, S_n$ are known *a priori,* so this information can be used to initialize a data structure we call a *hierarch*y or *lattice*, as shown in Figure 5.8. The lattice is a partially ordered set containing the relationship between the bucket contents and the subset references. Each node of the lattice represents a set of the subsets, such that the first level of the lattice contains one node representing all *n* sets, the second level contains *n* nodes representing all subsets of size *(n-1)* sets, and so on until the *(n+1)*st level contains an empty set. Each node has two vectors of size *n* associated with it. One is a vector of pointers to the next level of the lattice. This pointer array *P*, shown below each node in Figure 5.8, contains all pointers from that node to the next level in the lattice; a node will point to all nodes on the next level which are subsets of

it. For example, the node containing the set $\{S_1, S_2, S_3\}$ will point to the nodes $\{S_1, S_2\}$, $\{S_1, S_3\}$, and $\{S_2,S_3\}$. The pointer in vector location i points to the subset of the next smaller size which does *not* contain element i; thus, $P[2]$ of node $\{S_1, S_2, S_3\}$ contains the pointer to node $\{S_1, S_3\}$. If a node does not contain the subset S_i, then $P[i]$ in that node will be a null pointer. The configuration of the pointer vector P will *always* be the same for each problem instance of the same size.

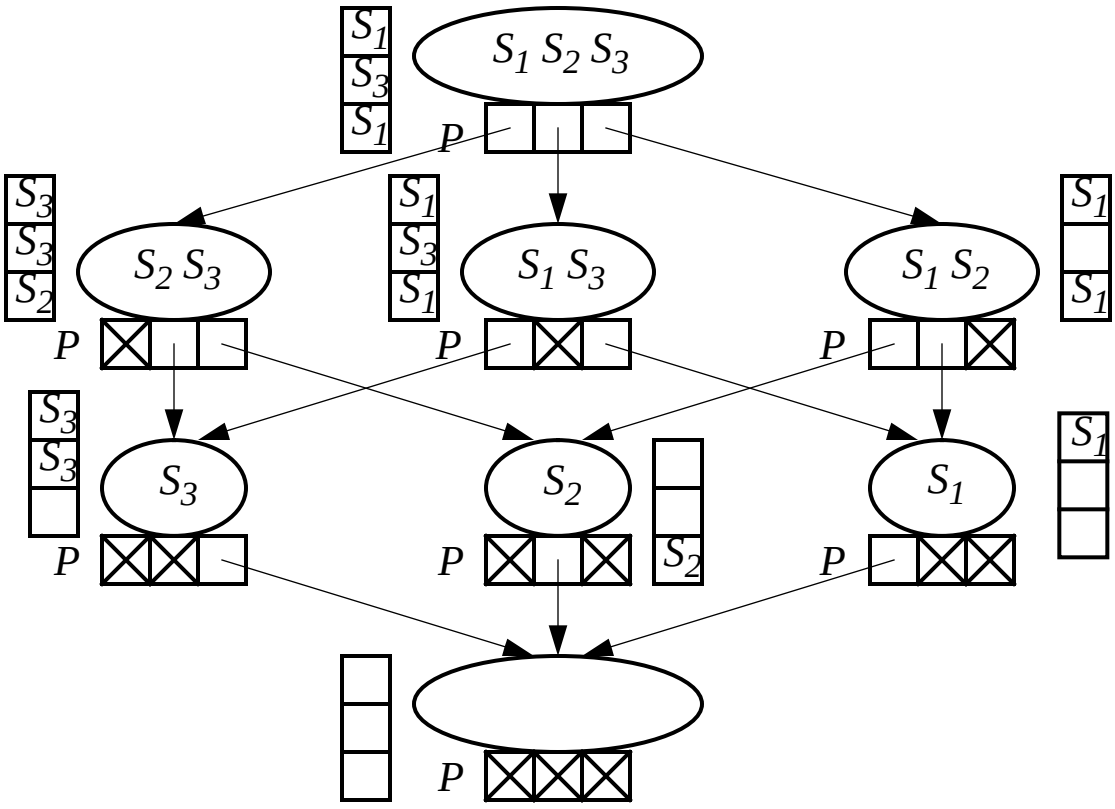


Figure 5.8 Lattice Used to Store Preprocessed Subset Information.

The second vector, A, is the solution vector, or vector which assigns bucket references to sets. This array is indexed by the bucket number. We have assumed the following three subsets of three total buckets: $S_1 = \{b_1, b_3\}$, $S_2 = \{b_3\}$, and $S_3 = \{b_1, b_2\}$ for the assignment of values to the A vector in Figure 5.8. Once the bucket values are sorted, the permutation of buckets is used to traverse the lattice and find the solution. For example,

in Figure 5.8, if b_2 is the sorted minimum, then S_3 is assigned the minimum value of b_2, and P[3] is followed to get to the next node in the lattice. The bucket reference b_2 will continue to be the bucket reference into A until all sets with that minimum have been assigned its value. Once this happens and an empty node in the A vector is found, then the next bucket reference in the sorted permutation is used to index the A array.

5.5.2. Space Complexity Analysis

We analyze the space requirements with a bit analysis. The total number of nodes in the lattice is 2^n. For each node, there are 2 vectors: P and A, each with n elements. Each pointer in P has n bits so that 2^n unique nodes can be referenced. Each element in the A vector has $\log n$ bits to reference n buckets. Therefore, the total number of bits is $2^n(n^2 + n \log n)$, or $O(n^2 2^n)$ bits.

5.5.3. Time Complexity Analysis

The time analysis will take into consideration that a word of size $\log n$ can be read in constant time. There are two parts to the algorithm: the sort of the bucket values and the assignment of minimum values to each set.

The sort will take $O(n \log n)$ time. Then the lattice will be used to assign minimum values to each set. There are n levels in the lattice. At each level, at least one A vector element will be read and one pointer P will be read. Each A vector element can be read in constant time. Each pointer, however, will take $n/\log n$ time to read because of its size. There are n levels of the lattice and at most n elements of vector A will be read at only one level. Therefore, the assignment process will take $O(n^2/\log n)$ time. Hence, the total time complexity for this algorithm is $O(n^2/\log n)$.

5.5.4. A Family of Solutions

Next we explore divide-and-conquer techniques to reduce the space complexity of the above problem. Assume that the sets are divided into n/k groups of k sets, where $1 \leq k \leq n$. Then separately solve the problem for each of the groups.

There will be n/k lattices, one for each group of sets. The lattices will be initialized as in Section 5.5.1. Once the sort of the n buckets is completed, the assignment of minimum values will take place by following the levels in every lattice. The algorithm is complete when each lattice has been used to assign values to sets.

We discuss the space requirements in terms of bit complexity. The total number of nodes in each lattice is 2^k. For each node, there are two arrays. The pointer array P will have k elements, each of size k bits. The bucket array A will have n elements, one for each bucket, each with $\log k$ bits to refer to one of k sets. So, the total space requirement for each node is $2^k(k^2 + n \log k)$. Since there are n/k nodes, the total space needed is $O(2^k(nk + (n^2 \log k)/k))$ bits.

Again, we consider the time to read values with size greater than $\log n$. There are two parts to the algorithm: the sort of the bucket values and the assignment of minimum values to each set. It will take $O(n \log n)$ time to sort n buckets; this doesn't change from the original algorithm. The time to traverse each lattice and assign values to k sets will be $n + k(k/\log n)$, where n is the time to read possibly n buckets, and $k(k/\log n)$ is the time to read all pointers to traverse the lattice. Since there are n/k lattices, the time complexity to assign values to all n sets will be $O(n^2/k + nk/\log n)$.

We discuss the selection of k by presenting a table of results. Recall that k can be between *1* and n. Given this we have a family of solutions, depending on the value of k:

Table 5.1 Family of Solutions.

k	Time Complexity	Space Complexity (in bits)
1	$O(n^2)$	$O(n^2)$
$\log n$	$O(n^2/\log n)$	$O(n^3(\log\log n)/\log n)$
$\log^2 n$	$O(n^2/\log^2 n)$	$O(n^4(\log\log^2 n)/\log^2 n)$
$n^{1/2}$	$O(n^{3/2})$	$O((2 \wedge (n^{1/2})) \cdot (n^{3/2}))$
$(n \log n)^{1/2}$	$O(n^2/((n \log n)^{1/2}))$	$O(n^2(\log((n \log n)^{1/2}))/(n \log n)^{1/2})$
n	$O(n^2/\log n)$	$O(2^n n^2)$

If $k = (n \log n)^{1/2}$, the time complexity is minimized at $O(n^2/((n \log n)^{1/2}))$. If $k = \log n$ or $k = \log^2 n$, then the time complexity is sub-quadratic, and the space complexity is polynomial in the number of bits.

5.6. A Physical Realization of a Target-specific Reduction Network

One way to compute target-specific reductions for all possible communication topologies in $O(\log n)$ time is to construct a reduction network by essentially duplicating a binary tree global reduction network n times, such that each of the n reduction results is sent to an LP executing on a different processor and a processor only contributes an input to a reduction if its LP is a predecessor to the LP receiving the result. The drawback to this approach is that $O(n^2)$ components (elements in the reduction network) are required. We use this construction to show an implementation of a target-specific reduction network in Figure 5.9. Each of the n inputs is broadcasted n times in a binary tree of broadcast switches, and then the n^2 leaves of the trees are inputs to the n binary trees of ALU's which

compute the reductions for each of the *n* outputs. The leaves of the broadcast trees are preprogrammed to determine if its input must be sent to the corresponding reduction computation; if an input is not sent to a reduction computation, the identity element for that reduction is sent instead. Whether there exists a reduction network which computes target-specific reductions for all communication topologies in *O(log n)* time with less than $O(n^2)$ components is still an open research question. The complexity of the network in Figure 5.9 is equivalent to Akl's best solution to compute the multiple criteria *n*-processor BSR, as discussed in Section 2.2.3.

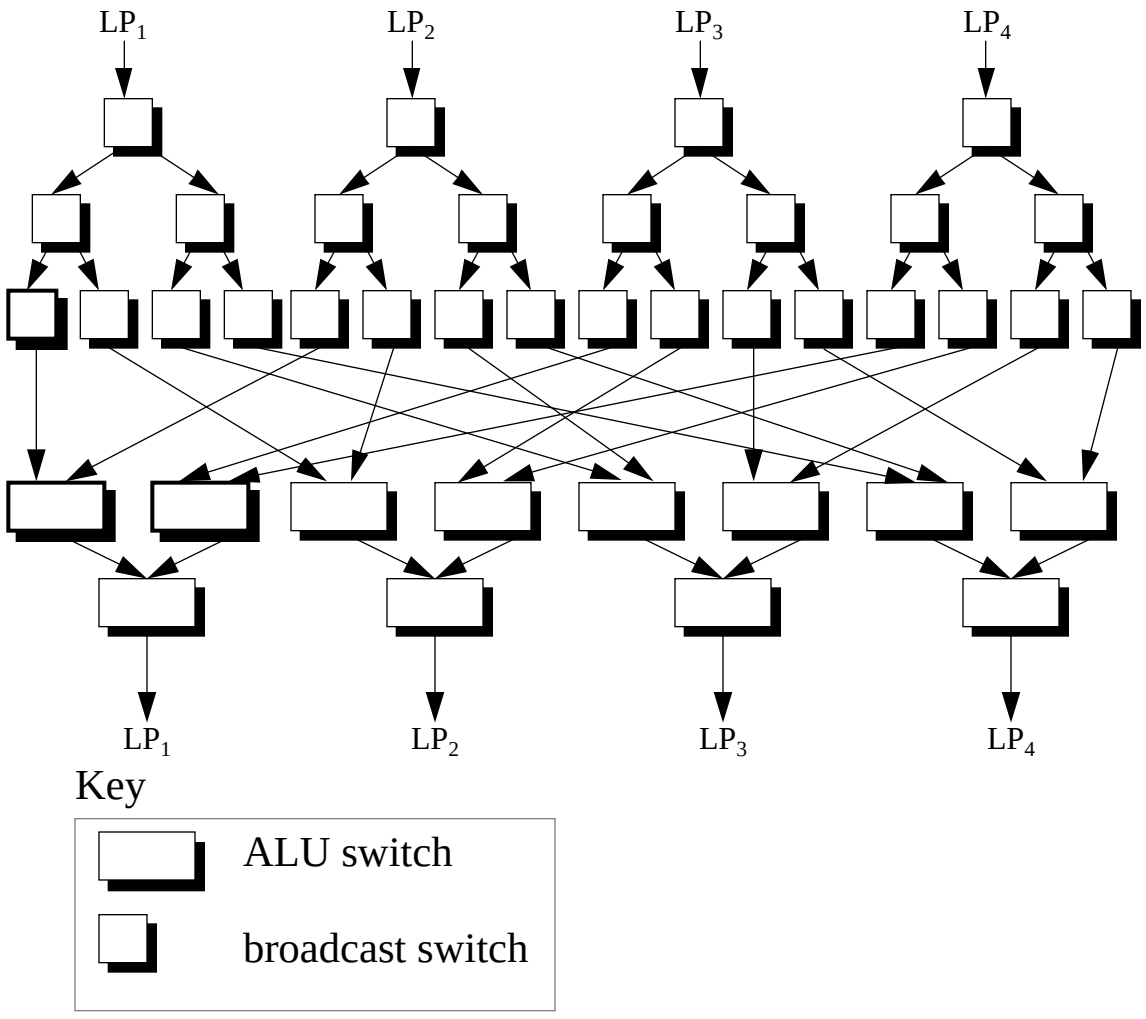


Figure 5.9A Target-specific Parallel Reduction Network.

5.7. Summary and Conclusions

In this chapter we have presented theoretical results on the cost of computing and disseminating target-specific reductions. We have two contributions in this area and several observations. First, we have demonstrated the applicability of target-specific reductions to a wide range of PDES synchronization protocols. In the next chapter we will present simulation results that quantify the benefits of target-specific reductions over global reductions to both conservative and optimistic protocols.

Second, we have shown two sequential algorithms which solve the target-specific dissemination problem in the general case. The algorithms show a trade-off between the time and space complexity. The first algorithm has a $O(n \log n)$ time complexity with the associated cost a $O(2^n)$ space complexity in bits. The second solution is actually a family of solutions, where the best time/space complexity combination of solutions is $O(n^2/\log^2 n)$ time complexity and $O(n^4(\log \log^2 n)/\log^2 n)$ space complexity in bits. These are the best known time/space complexity results for this problem. They are encouraging because of the sub-quadratic time complexity of the associated algorithms.

Also, we can make some observations regarding the computation of target-specific values in parallel. We have presented a design of parallel target-specific reduction network which computes n target-specific reductions in $O(\log n)$ time with $O(n^2)$ components. This is a small contribution but it is worthy of note at this time because the product of the time and space complexity is $O(n^2 \log n)$, which is less than the product of the time and space complexities of any of our sequential solutions. It also serves as a proof of concept. It is also an interesting observation that this parallel target-specific reduction network computes *all* binary, associative operations and is not limited to the computation of minimum and maximum values.

In the next chapter we present simulation results which demonstrate the need for hardware to compute target-specific reductions in support of parallel simulations.

6 Performance of Global versus Target-specific Reductions

In this chapter we present results of simulations that strongly suggest the need for a next-generation reduction network to compute and disseminate results of target-specific reductions to support both aggressive and non-aggressive parallel discrete event simulations. Many of these simulation results were first presented in [PARE93]. As established in previous chapters, target-specific reductions allow an LP to receive synchronization information only from those logical processes which may have a direct or indirect impact on its performance.

To determine the performance gains of a reduction network which computes target-specific reductions, we have simulated the two conditions: several PDES's operating on top of a reduction network which computes globally reduced values and several PDES's operating on top of a reduction network which computes target-specific reduced values across subsets of LP's (assuming one LP per processor). The goal of these simulations was to demonstrate the utility and benefits of target-specific reduction networks.

We have concluded that target-specific synchronization information offers significant benefits to conservative PDES's. In a conservative parallel simulation, target-specific synchronization information reduces the finishing time of the simulation. This result is intuitive: near-perfect state information – information that comes close to the true state when the near-perfect information is received – eliminates artificial dependencies and enables more parallelism in a PDES.

We have also concluded that the dissemination of target-specific synchronization information is beneficial to Time Warp-like PDES's. The dissemination of target-specific critical state information allows GVT to be computed on a local basis — target-specific virtual times (TSVT's). Since a TSVT is based only on information relevant to the target LP, thus, in a sense, making it more accurate than GVT, fossils (state information that precedes a TSVT) will be reduced.

In the first section we describe the algorithms used in the simulations. In Section 6.2. we give a brief description of the hardware model used in the simulations. In Section 6.3. we describe our simulation assumptions. In Section 6.4. we present the results of our simulations. Finally, in Section 6.5. we discuss the implications of our results.

6.1. Simulation Algorithms

We simulated two different PDES's, a conservative parallel simulation and an optimistic parallel simulation. Each PDES was simulated on top of the two hardware configurations, each with the high-level configuration in Figure 3.2. The only difference between the two configurations is the type of reductions computed in the reduction network.

6.1.1. Conservative Simulation Algorithms

The conservative PDES is based on the synchronization algorithms in [REYN92] (See Section 3.1.1.). LP's maintain a next event time η_i and a smallest unreceived message time υ_i; two globally reduced minimum operations are performed on these inputs, giving η' and υ', respectively. The LP with the smallest next event time, such that it is not larger than the global minimum unreceived message time, can safely execute its event. Message acknowledgments are performed with the two-phase protocol (See Section 4.3.) in the

reduction network in order to correctly maintain the smallest unreceived message time. All acknowledgments are batched in order to use the reduction network more efficiently.

When the conservative PDES is simulated on top of the target-specific reduction network, two target-specific minimum operations are computed for each LP: a target-specific minimum next time η'_i and a target-specific smallest unreceived message time υ'_i. Each LP receives reduced information only from those LP's that can have an impact on its performance, and LP's will receive more accurate state information. As described in Chapter 5, target-specific primary acknowledgments have the same target as messages, and consequently, target-specific handshake acknowledgments are computed using the inputs from only immediate successors. Thus, from the perspective of a given LP, its target-specific inputs and outputs depend on the operation being performed, and different LP's will have different sources of inputs and different targets.

6.1.2. Optimistic Simulation Algorithms

The optimistic PDES implemented is a Time Warp simulation [JEFF85]. (See Section 2.1.3. for details on Time Warp simulations.) In our simulations, antimessages are cancelled aggressively, such that all events sent in the LP's future are cancelled at the time of rollback. Furthermore, our simulations *batch antimessages* to each receiving LP so that the total amount of message traffic in both the host communication network and our synchronization network is reduced. A batch antimessage allows multiple antimessages to be sent in one physical message, reducing the total number of outstanding messages in the host network. Furthermore, the number of messages which must be acknowledged in the reduction network is also reduced. GVT is computed in the reduction network as the smallest time in the system: the minimum of the smallest logical clock time σ' and minimum unreceived message time υ'. (See Section 3.1.2.) The optimistic parallel

simulations also use two-phase acknowledgments of both messages and antimessages in the reduction network in order to correctly maintain GVT.

When the optimistic PDES is implemented on top of the target-specific reduction network, a target-specific virtual time (TSVT) is computed for all LP's. (See Section 5.1.2.) Since $TSVT_i$ is customized for each LP_i, $i = 1, 2, \ldots, n$, it more accurately reflects the state information on which to base event processing decisions. Furthermore, fossil collection should be done with more accurate information. This supports a better utilization of state saving memory.

6.2. Hardware Computation Model

A hardware computation model can be found in Figure 3.2. The host system is a closely coupled network of high speed processors with its own network for interprocess communication. Each host processor (HP) is paired with a dedicated auxiliary processor (AP) which performs all synchronization activity and interfaces to the high speed reduction network. The PDES synchronization protocol and all event processing occurs on the host processors. Interfaces between a host processor and its corresponding auxiliary processor and between an auxiliary processor and the reduction network are designed to permit the correct execution of a PDES while allowing the host processors to operate asynchronously with the auxiliary processors and reduction network. (See Chapter 3 for details.)

For our simulations we assume that all reductions in the reduction network are computed in *O(log n)* time where *n* is the number of processors in the host system. In a reduction which computes globally reduced values, a binary tree-shaped reduction network, as , can compute reductions with this time complexity. Furthermore, this same network requires *O(n)* components.

A reduction network which computes and disseminates target-specific reductions in *O(log n)* time is certainly feasible. (See Chapter 5.)

6.3. Simulation Assumptions

We have made certain assumptions with respect to our simulations. We believe them to be realistic with respect to current technology and parallel simulations in general. Assumptions about the reduction network and all interfaces are based on the prototype design of our four-processor global reduction network [REPS93].

- Each logical process LP_i in the parallel simulation executes on a dedicated physical host processor HP_i.
- There are two times represented in this simulation. Logical time refers to the logical time of the PDES being simulated. Wallclock time refers to the simulated time of physical events, where a physical event can be a message sent through the host communication network, an event being executed, or a reduction operation being performed.
- The communication topology of the PDES is based on an input graph to the simulator. An LP sends an event message with equal probability to any of its immediate successors. Each event generates one new event message. Internal events are generated by an LP so that the workload of all processors is the same.
- LP's send event messages and antimessages through the host communication network. The wallclock time to send a message is based on a probability distribution and the distance between two processors in the physical system. The physical distance or number of hops between processors is an input to the simulations. All simulations presented in this chapter assume that the distance between any two processors is the same.
- The reduction networks take 150 ns. per stage. It takes 90 ns. per 16-bit read or write in the auxiliary processor. It takes 100 ns. to read values from the interface to the PRN. Likewise it takes 100 ns. for values to propagate from the PRN to the level of the interface readable by the AP.
- The wallclock time that it takes for a host processor to read values from the host processor - auxiliary processor interface is 95 nanoseconds for a 32-bit read.
- The AP executes all acknowledgment processes. Batched acknowledgments, as described in Chapter 4, are implemented.
- An HP will poll its interface to its AP at regular intervals, with no delay. This suggests that GVT computed in the reduction network is as accurate as possible.

- The target-specific reduction network can compute reductions with the same speed as the global PRN.
- The wallclock time to execute an event is a parameter consisting of a distribution and a mean. The distribution can be exponential or uniform random.
- The logical time to execute an event is also a parameter consisting of a distribution and a mean. The distribution can be either exponential or uniform random.
- An antimessage will cause an event to be interrupted.
- Antimessages are cancelled aggressively in batches if possible. A batch antimessage is a group of antimessages sent through the host network in one physical message.
- Fossil collection is uniform randomly distributed to be performed after every 2-5 events.
- State saving is performed after each event.
- There is no cost associated with state saving or fossil collection. There is an unbounded amount of state saving memory. This is a reasonable assumption, assuming the rollback chip of Fujimoto [FUTG92].
- States are counted each time fossil collection occurs. A state is either a snapshot of the current state or a message in the output message list, which is used to determine where to send antimessages.

The topologies of the graphs we studied were varied to reflect certain degrees of fan-in, fan-out and combinations of the two. We chose to work with relatively small (4 and 8-node) graphs in order to guarantee reasonable execution times for the large set of experiments we ran. We expect to see trends are we increase the size of graphs.

6.4. Simulation Results

The following parameters were used in every simulation:

- The wallclock time to execute an event was uniform randomly distributed between 0 and 2 milliseconds with a mean of 1 millisecond.
- The simulation time to execute an event was uniform randomly distributed between 0 and 20 units with a mean of 10 units.
- The wallclock time to send a message takes at least 100 microseconds. In 85% of all messages sent, the message will arrive in 100 microseconds. In 6% of all messages sent the message will arrive in 200 microseconds. In 4% of all messages sent the message will arrive in 300 microseconds. In 3% of all messages sent the message will arrive in 400 microseconds. Finally, in 2% of all

messages sent the message will arrive in 500 microseconds. We believe that these times are representative of current technology and message traffic patterns.

6.4.1. Topology of Four Logical Processes

A linear topology such as in Figure 6.1 was used in the simulations with four LP's. These simulations were run until GVT was greater than 15,000, i.e. the termination condition was GVT exceeding 15,000 units. (Approximately 12,000 to 20,000 total events were executed.)

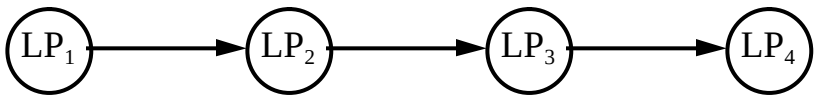


Figure 6.1 Linear Topology With Four LP's.

For each simulation, we report our results using bar graphs. The left bar graph in each figure shows the simulation results of the conservative PDES: one bar indicates the execution time of a conservative PDES operating on top of a reduction network which computes globally reduced values and the second bar indicates the execution time of the same conservative PDES operating on top of a reduction network which computes target-specific reduced values. Likewise, the bar graph for the optimistic simulations show the difference in the total amount of state space in the simulation, when the same simulation has globally reduced values and target-specific reduced values.

Figure 6.2 shows the simulation results of this four LP linear topology. As expected, the target-specific dissemination of synchronization information reduced the finishing time of the conservative PDES by a factor greater than 3. We note that the finishing times of the optimistic PDES's were not affected by the dissemination of target-specific reductions.

(There are some areas where computed TSVT might be beneficial to reducing the execution time of the simulation. This is to be explored.) The total amount of state space in the simulation is reduced by a factor of 3.6. The total amount of state space is computed for all LP's, in this case for four LP's, and not on a per LP basis. If the state space were limited, as it is in Fujimoto's high-speed rollback chip [FUTG92], this can be a significant savings.

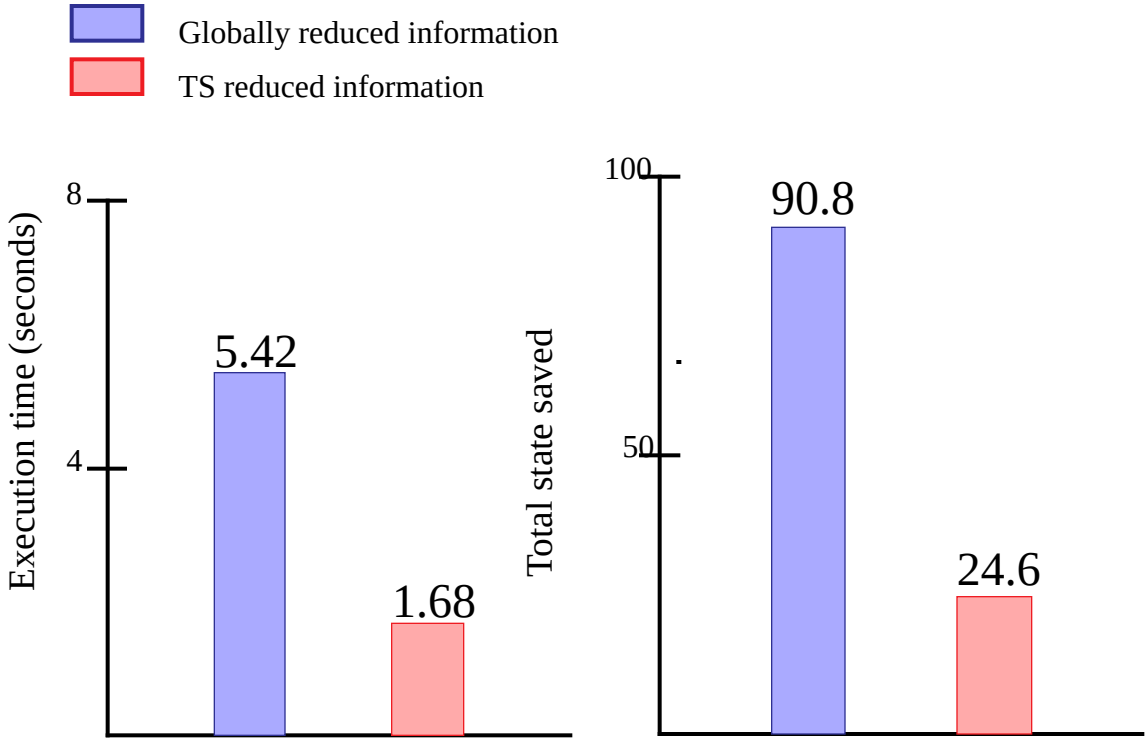


Figure 6.2 Results of Linear Topology with Four LP's.

6.4.2. Topologies of Eight Logical Processes

For the simulations of size eight, we used more interesting topologies. All topologies are acyclic graphs because a cyclic subgraph can be reduced to a single node requiring the same target-specific information that each of its components requires. Figure 6.3 depicts a communication topology which we refer to as a fan-out graph. A fan-out graph has one LP that is a predecessor of every other LP in the system; this LP, LP_1 in Figure 6.3,

is the single source. LP_i, $i = 1, 2, \ldots, n$, needs target-specific information from all LP's on the path from the source to it.

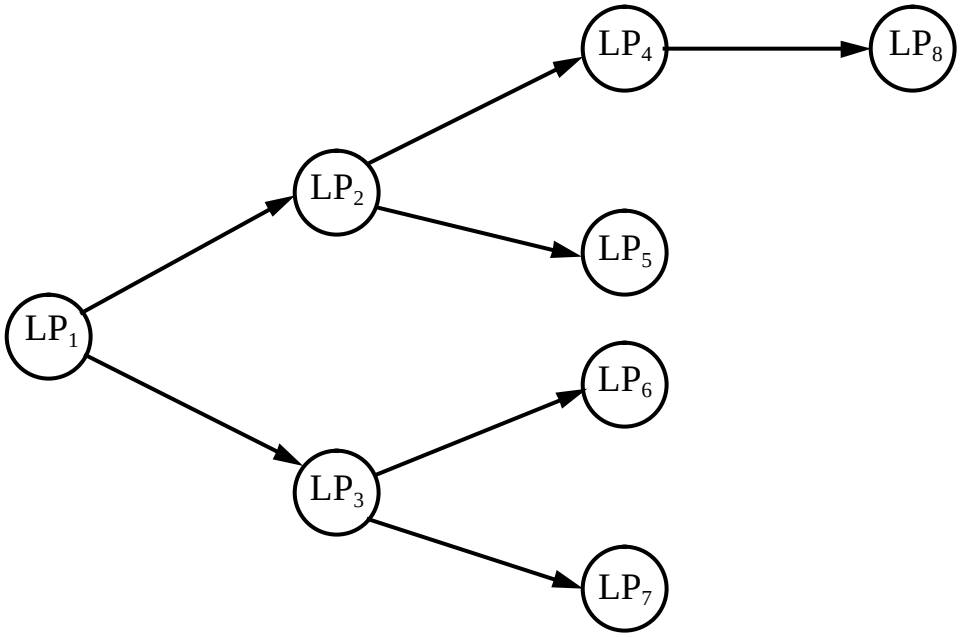


Figure 6.3 A Fan-out Topology With Eight LP's.

Figure 6.4 depicts a communication topology which we refer to as a fan-in graph. A fan-in graph has a single sink LP, in this case LP_8, such that every other LP in the system is a predecessor to that sink. The sink needs global synchronization information.

Finally, Figure 6.5, Figure 6.6, and Figure 6.7 illustrate both fan-in properties and fan-out properties in a communication topology for a PDES with eight LP's. Each of these topologies has a different dissemination pattern for target-specific information.

Figure 6.5 is a topology where no two LP's have the same immediate predecessor set. We believe this type of graph represents a class of graphs for which it may be difficult to provide target-specific information in a general interconnection network.

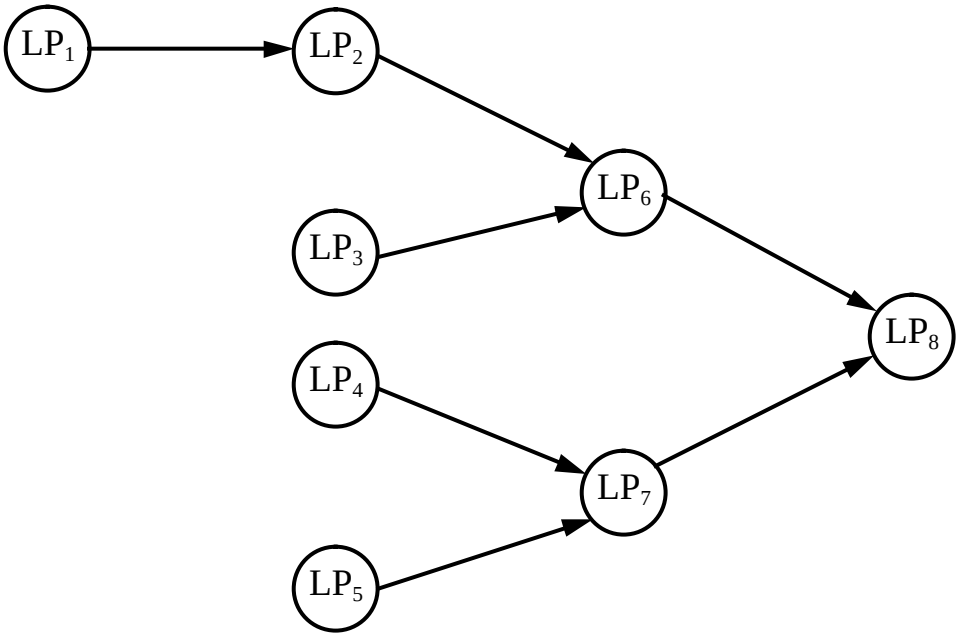


Figure 6.4 A Fan-In Topology With Eight LP's.

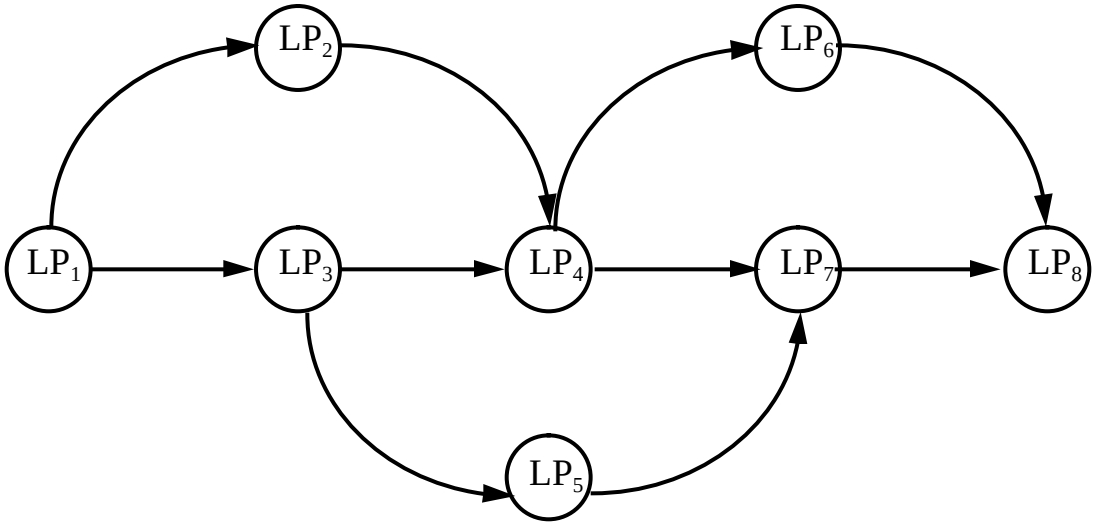


Figure 6.5 A Fan-in/ Fan-out Topology With Eight LP's.

Figure 6.6 shows a very regular graph with a single source and a single sink. Three independent paths exist from the source node LP_1 to the sink LP_8.

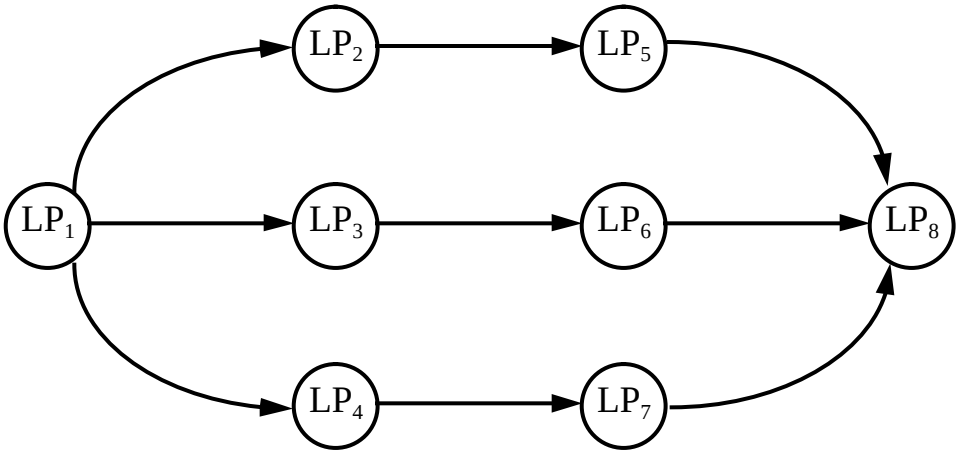


Figure 6.6 A Fan-in/ Fan-out Topology With Eight LP's.

Figure 6.7 adds two additional communication channels to Figure 6.6. Hence, LP_6 is now dependent on both LP_2 and LP_4. The additional dependencies will change the necessary target-specific reductions which will be computed to support this topology.

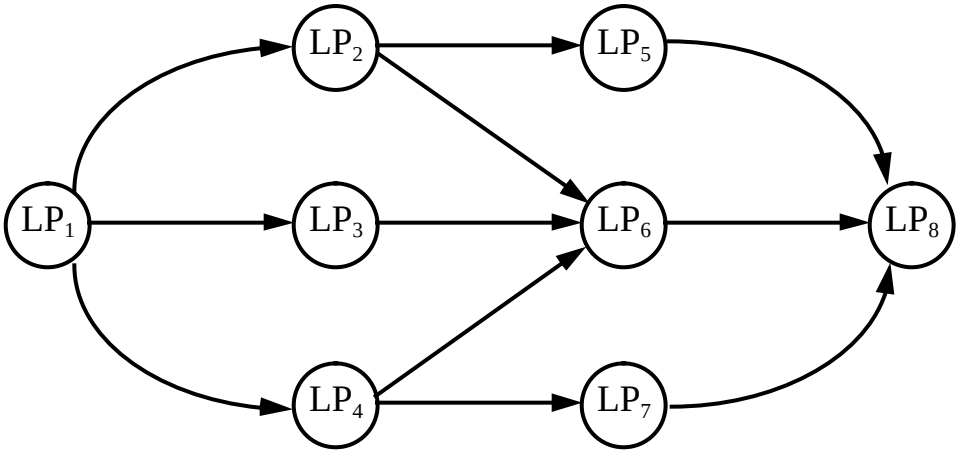


Figure 6.7 A Fan-in/ Fan-out Topology With Eight LP's.

6.4.3. Results of Simulations with Eight LP's

All of the simulations which were run for eight LP's executed until GVT exceeded 20,000. In other words, the termination condition for the simulations of eight LP's was that GVT was greater than 20,000 units.

Figure 6.8 is a bar graph showing the results of the PDES fan-out communication topology in Figure 6.3. We note that the finishing times of the optimistic simulations were essentially the same. The benefit of providing hardware support for target-specific virtual time is that the amount of saved state over time decreases. In the optimistic simulations with eight LP's in this fan-out topology, the total state space required was cut by a factor of 4.5. As expected, the finishing time of the conservative PDES was reduced significantly,

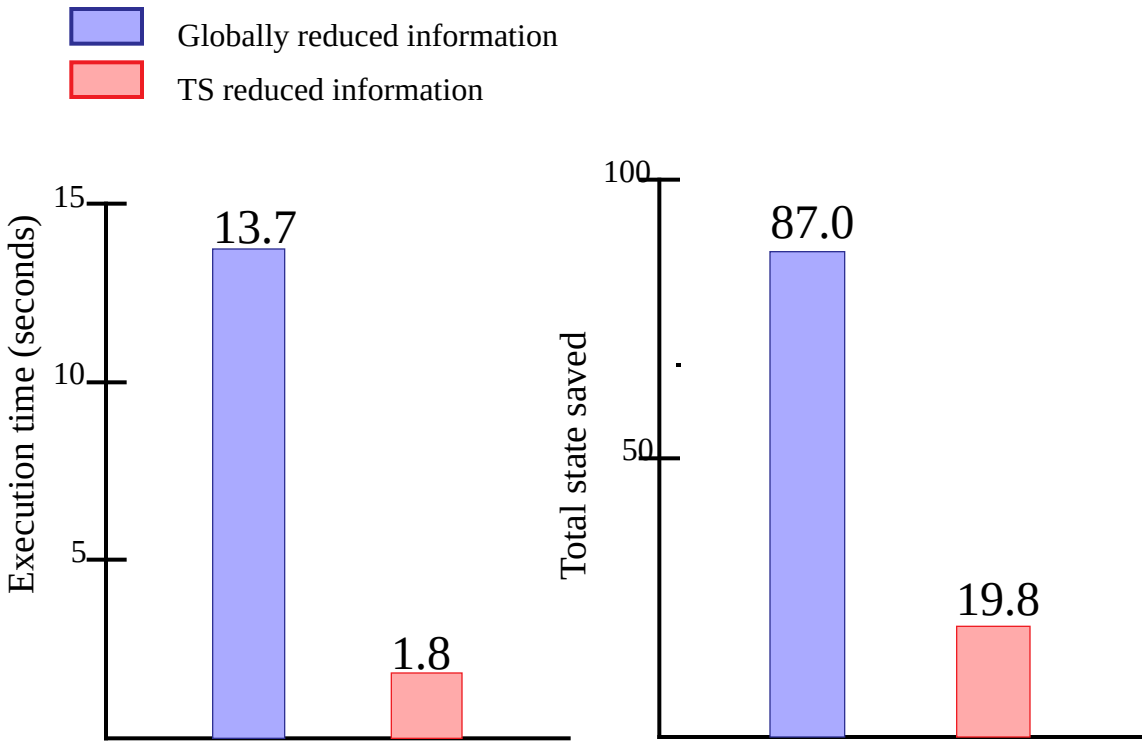


Figure 6.8 Results of Fan-out Topology With Eight LP's.

approximately 7.5 times, with the benefit of target-specific synchronization information on which to base processing decisions.

Figure 6.9 shows the results of the PDES fan-in communication topology in Figure 6.4. The finishing time for the conservative PDES was reduced by a factor of 5 when target-specific state information was computed and disseminated.

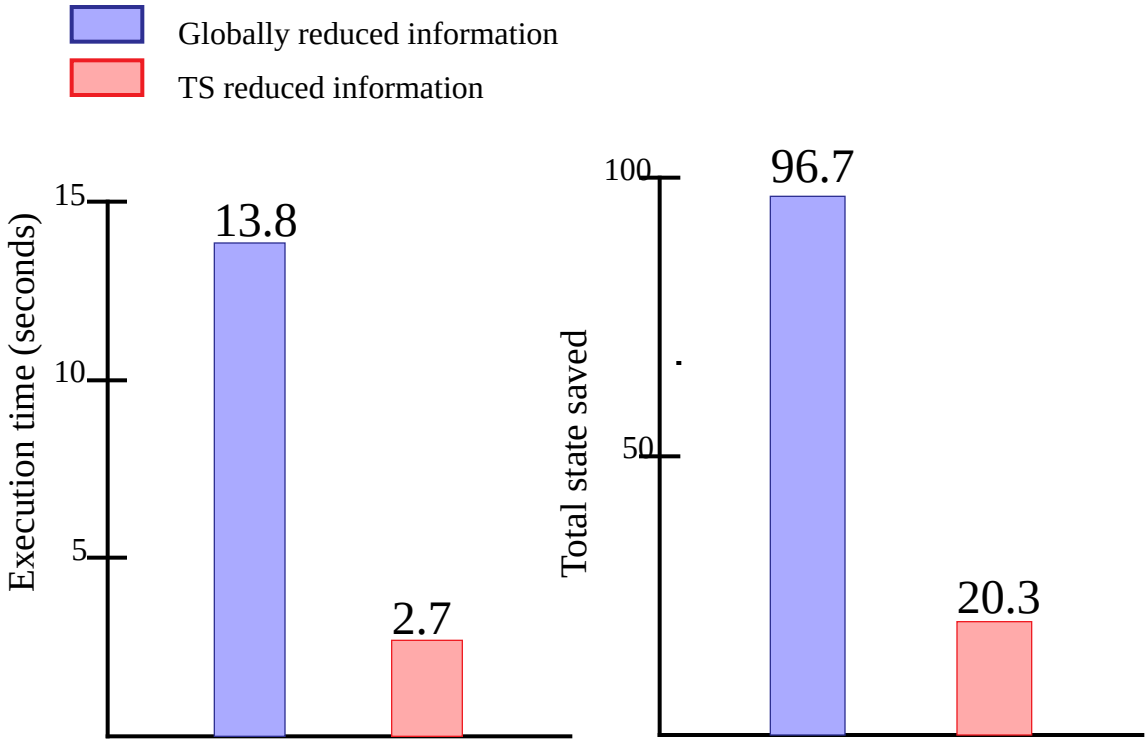


Figure 6.9 Results of Fan-in Topology With Eight LP's.

Due to the communication topology, fewer average states are saved in the optimistic fan-in topology (Figure 6.9) than in the optimistic fan-out topology (Figure 6.8). This is because there are more source LP's, and source LP's do not need to save state at all. However, the total amount of memory needed to save state in the simulations of the fan-in topology was still reduced by a factor of 4.75 when target-specific reductions were computed and disseminated.

Figure 6.10 is a bar graph showing the results of the first combination graph, i.e., with both fan-in and fan-out properties. In this conservative PDES with eight LP's the finishing time was reduced by a factor of 5. The dissemination of TSVT had a much greater effect on this topology than either the fan-in graph (Figure 6.9) or the fan-out graph (Figure 6.8); the amount of state space needed was reduced by a factor of over 6.

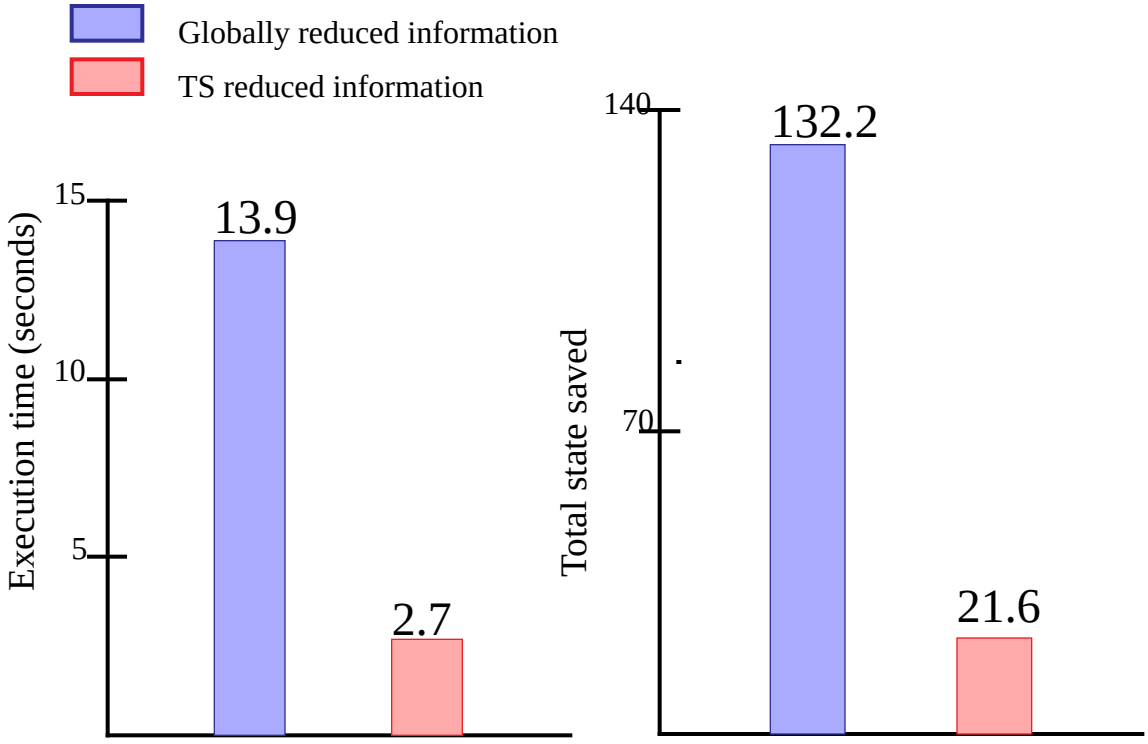


Figure 6.10 Results of Topology With Eight LP's in Figure 6.5.

Figure 6.11 is a bar graph showing simulation results for the topology in Figure 6.6. In the conservative parallel simulation, the finishing time was reduced by a factor of 5.7, the effect is slightly greater than that reported in Figure 6.10. The total amount of state space needed in the optimistic parallel simulation was reduced by a factor of 15. This is a significant reduction to the total average state space required to run the simulation.

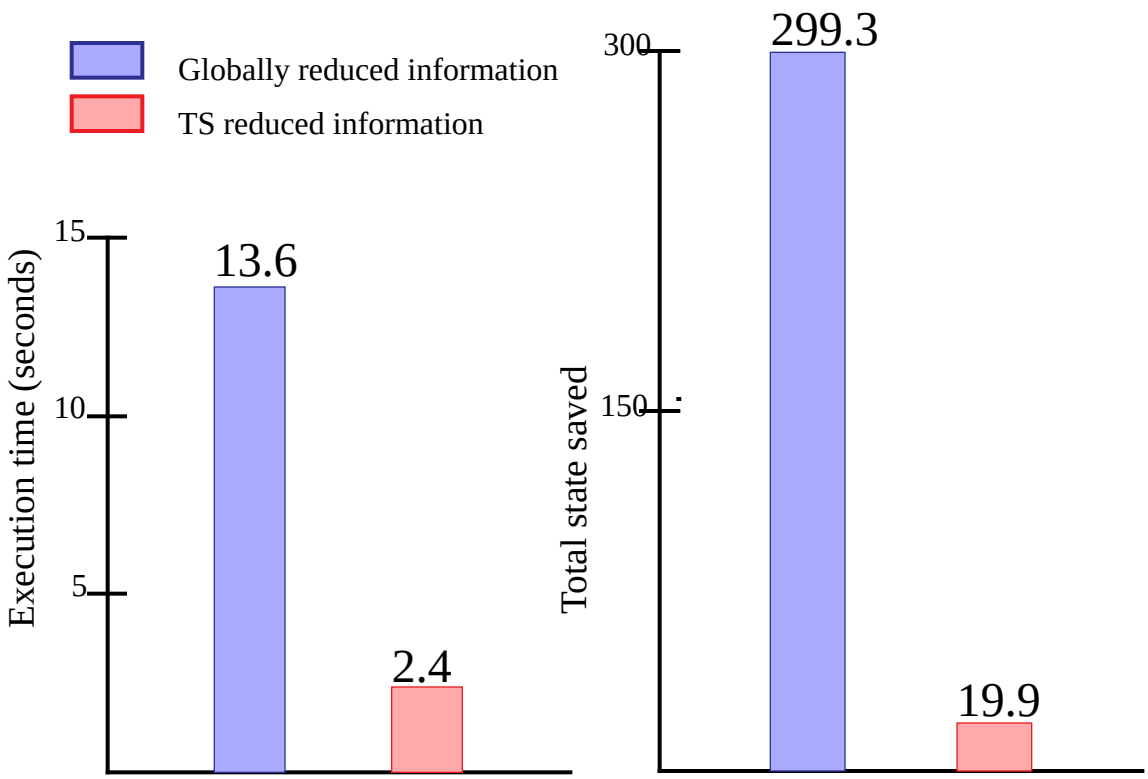


Figure 6.11 Results of Topology With Eight LP's in Figure 6.6.

Finally, Figure 6.12 is a bar graph of the results of the Figure 6.7, the augmented topology of Figure 6.6 with additional directed arcs into LP_6. The reduction to the execution time in the conservative PDES is a factor of 5; this is essentially the same as the results without the additional arcs. On the other hand, the results of the optimistic simulations for the same graph differ. In the optimistic PDES, the reduction to the state space is a factor of 7.7. Notice that the total average states saved in the PDES in Figure 6.7 sitting on top of the target-specific hardware and that in the PDES in Figure 6.6 using the

target-specific hardware differ by a factor of 2. The additional dependencies require more state to be saved, in the average case, by LP_6.

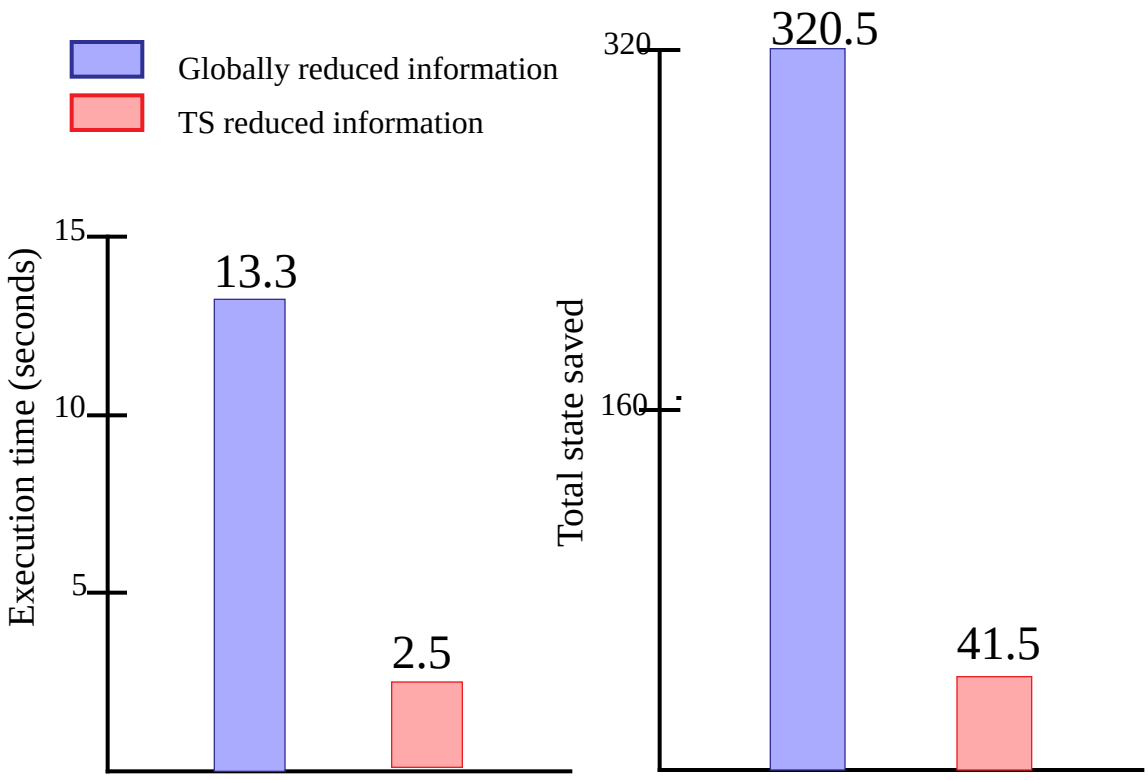


Figure 6.12 Results of Topology of Eight LP's in Figure 6.7.

6.4.4. Results of Simulations with Sixteen LP's

We simulated conservative PDES of size sixteen LP's operating on top of both a reduction network computing globally reduced values and a reduction network computing target-specific reductions. We did not have the resources to run an optimistic PDES for topologies of larger size than eight.

We simulated four different topologies of 16 LP's: a linear topology, a fan-out topology (similar to Figure 6.3), a fan-in topology (similar to Figure 6.4), and a topology with both fan-in and fan-out properties (similar to Figure 6.5). Our results follow.

Figure 6.13 shows the results of the linear topology of sixteen LP's. The total execution time of the conservative PDES is reduced by more than a factor of eight when target-specific reductions, instead of global reductions, are computed and disseminated in support of the simulation.

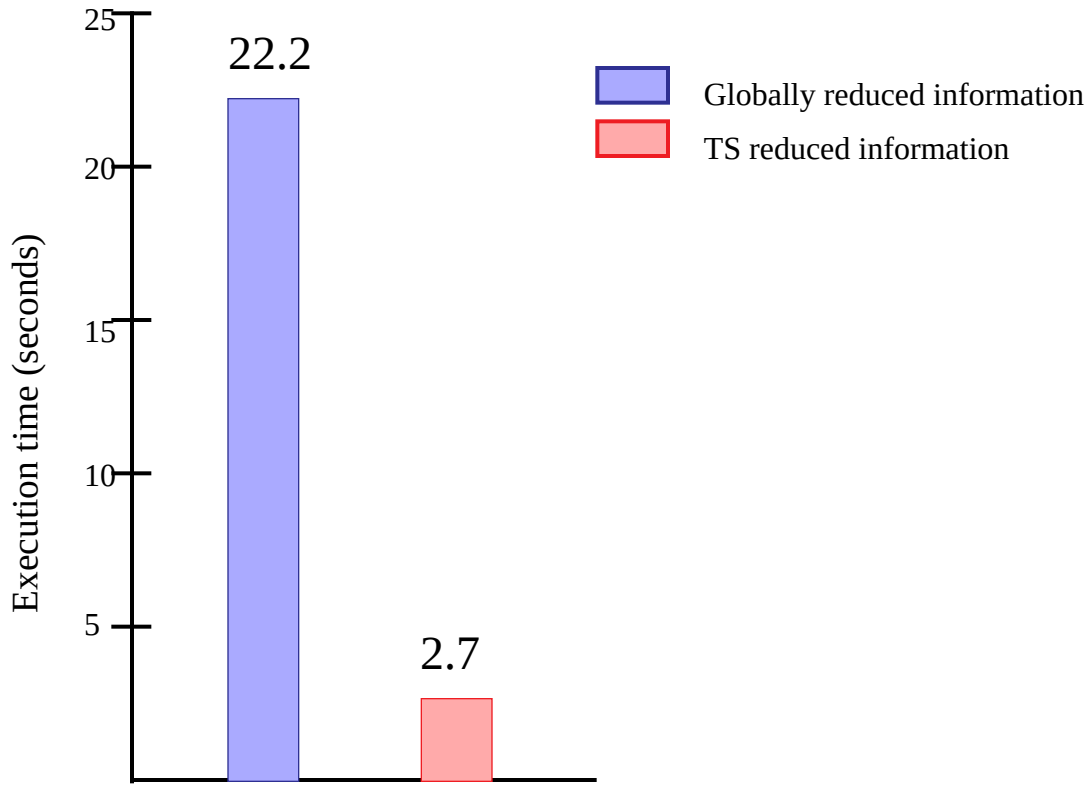


Figure 6.13 Results of Linear Topology of Sixteen LP's.

Figure 6.14 shows the results of a fan-out topology of sixteen LP's. The total execution time of the simulation is reduced by a factor of over 10.5. We observe a greater improvement in the fan-out topology with sixteen LP's than with eight LP's. This suggests

that target-specific reductions could have more benefits to a conservative PDES as the number of LP's increases.

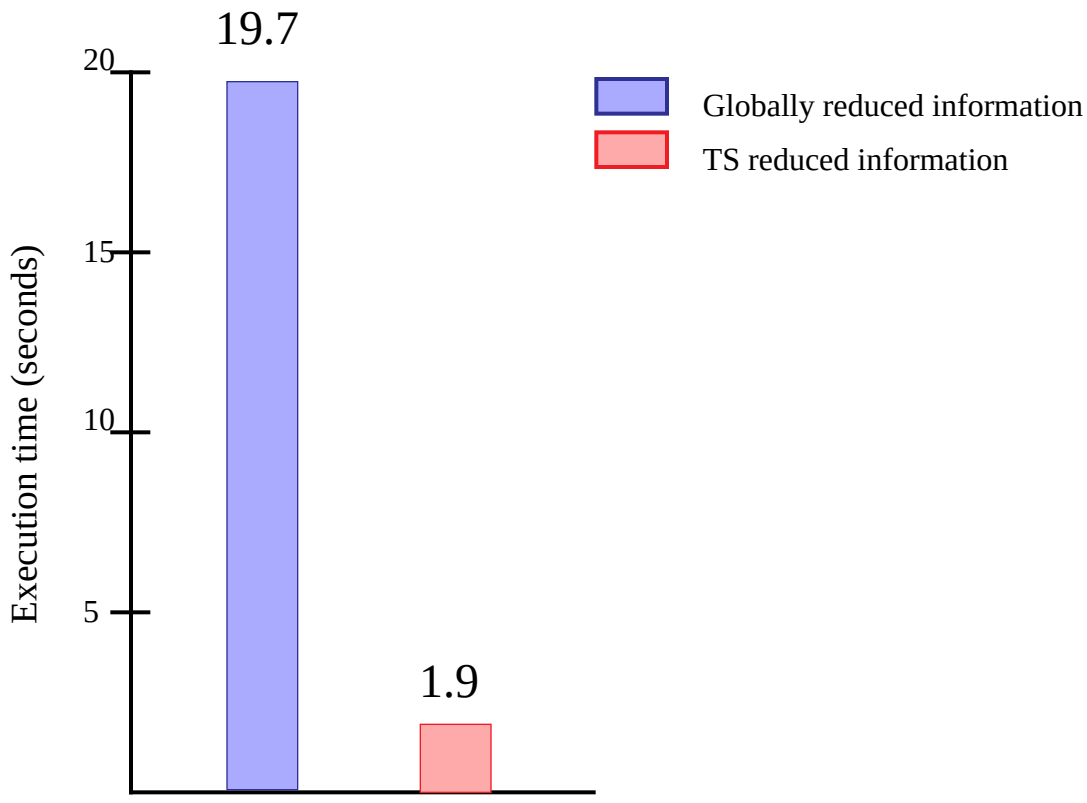


Figure 6.14 Results of Fan-out Topology of Sixteen LP's.

Figure 6.15 shows the results of a fan-in topology of sixteen LP's. In this case there is a reduction in the finishing time of the conservative PDES by a factor of eight. Again this is an increase in the performance from the eight LP fan-in topology.

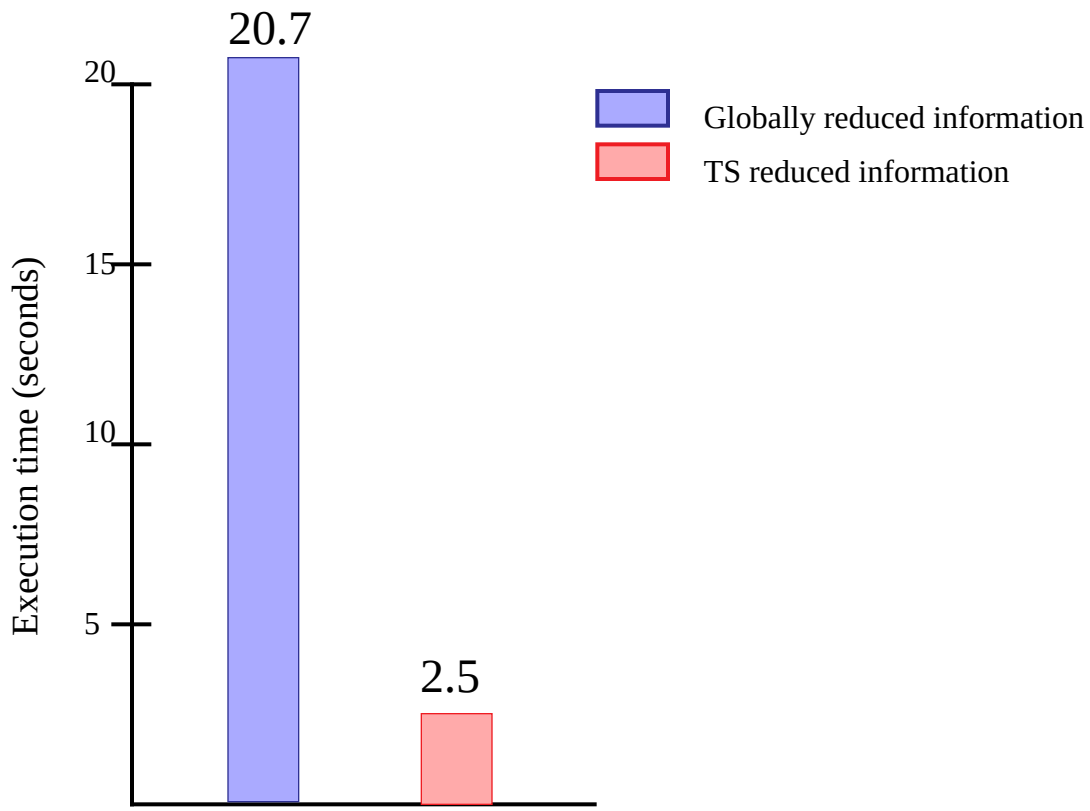


Figure 6.15 Results of Fan-in Topology of Sixteen LP's.

The final topology of sixteen LP's that we simulated of sixteen LP's that has very similar communication properties to Figure 6.5. This communication topology is characterized by the property that no two LP's have the same immediate predecessor set.

In other words, each LP has a unique set on which target-specific reductions are computed. The results of the simulations with this topology can be found in Figure 6.16.

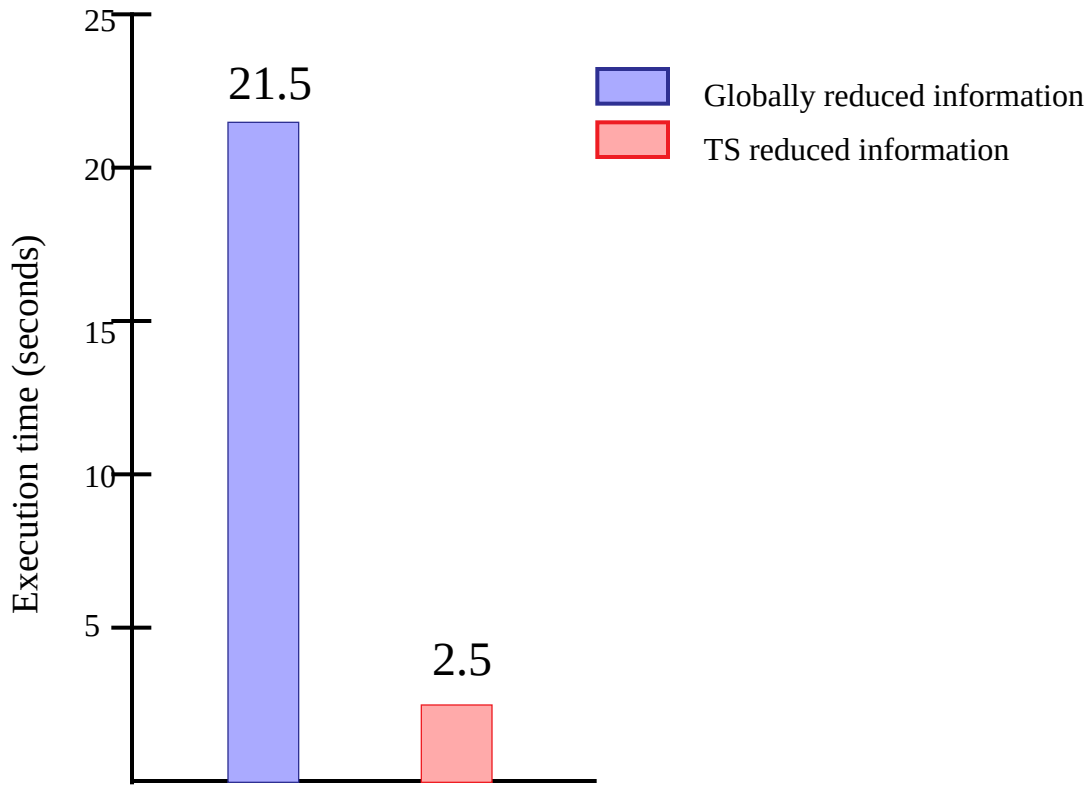


Figure 6.16 Results of Fan-in/ Fan-out Topology of Sixteen LP's.

6.4.5. Results of Simulations with Thirty-two LP's

We simulated conservative PDES of size thirty-two LP's operating on top of both a reduction network computing globally reduced values and a reduction network computing target-specific reductions.

We simulated the same four different topologies of 32 LP's as we did with 16 LP's: a linear topology, a fan-out topology (similar to Figure 6.3), a fan-in topology (similar to Figure 6.4), and a topology with both fan-in and fan-out properties (similar to Figure 6.5).

We find again that the execution time of a conservative PDES with 32 LP's is greatly reduced with the availability of target-specific reductions.

In Figure 6.13 we present the results of 32 LP's in a linear topology executing a conservative PDES on top of both a global reduction network and a target-specific reduction network. The speedup measured is a factor of 11.5. This is a greater performance improvement than the sixteen LP linear topology.

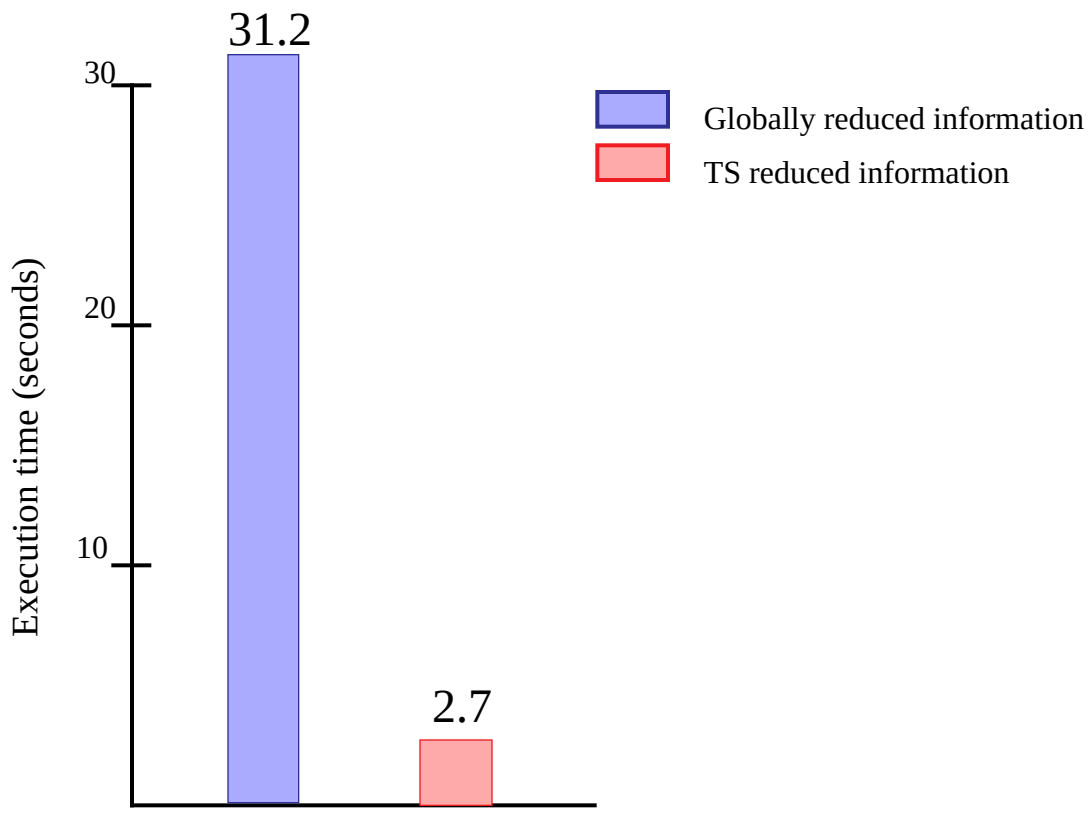


Figure 6.17 Results of Linear Topology of 32 LP's.

Figure 6.14 shows a bar graph for the conservative PDES's of a 32 LP fan-out topology executing on top of the two reduction networks. In this case the execution time of the simulation is reduced by a factor of 15.5 when the target-specific next event times and

unreceived message times are computed. The effect of the target-specific information is once again more significant in the 32 LP fan-out topology than the 16 LP fan-out topology.

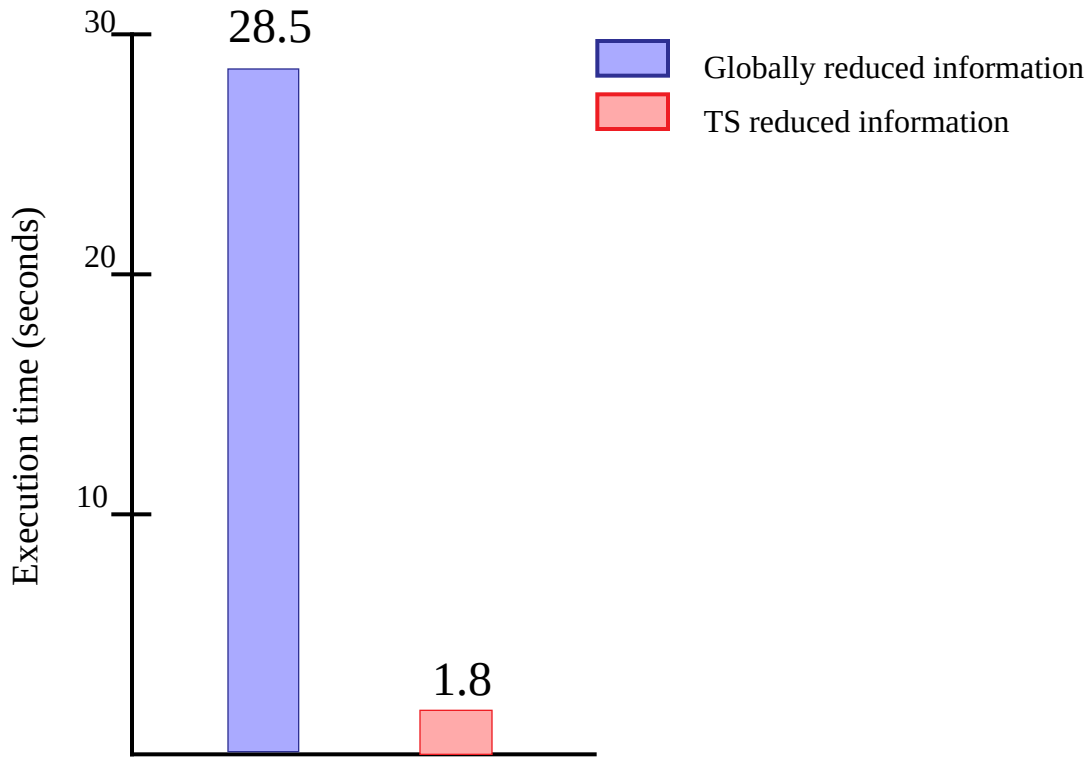


Figure 6.18 Results of Fan-out Topology of Sixteen LP's.

Figure 6.15 shows the results of a fan-in topology of 32 LP's. The execution time of the conservative PDES is reduced by a factor of approximately twelve. Recall that the improvement in execution time was a factor of eight for the fan-in of 16 LP's.

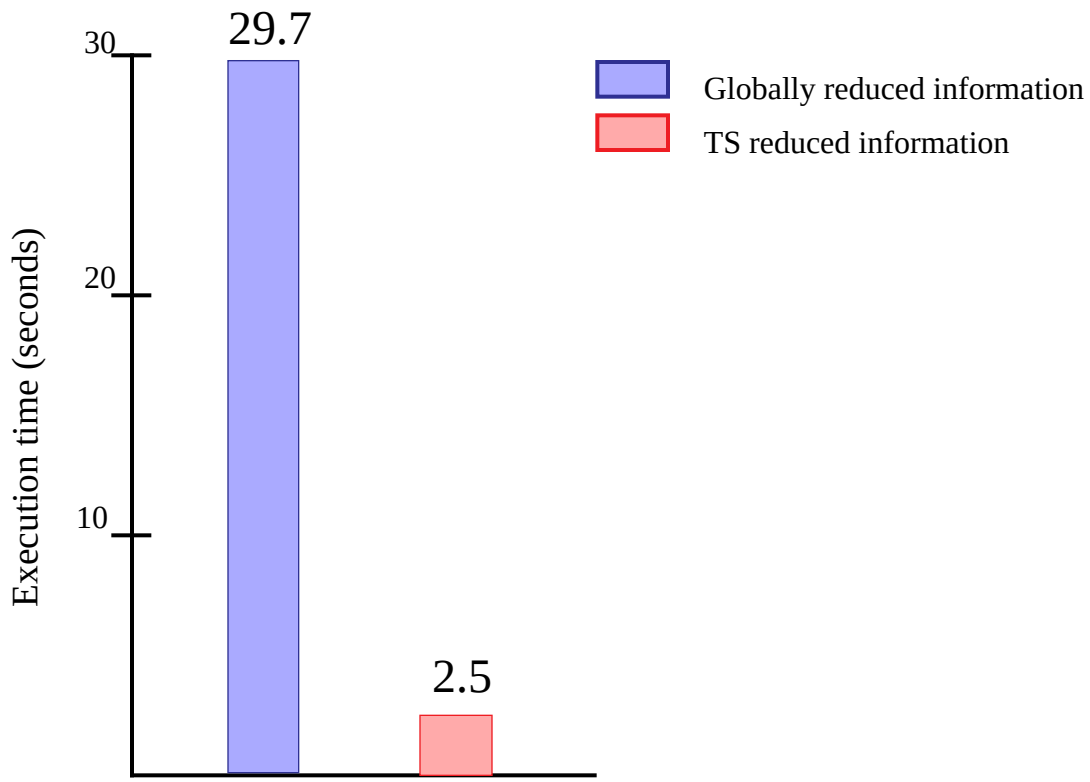


Figure 6.19 Results of Fan-in Topology of 32 LP's.

Finally in Figure 6.16 we see the results of a 32 LP communication topology that combines fan-in and fan-out properties such that no two LP's have the same predecessor set. In this case the execution time of a conservative PDES executing on top of the reduction network that computes target-specific reduced values is reduced by a factor of over twelve.

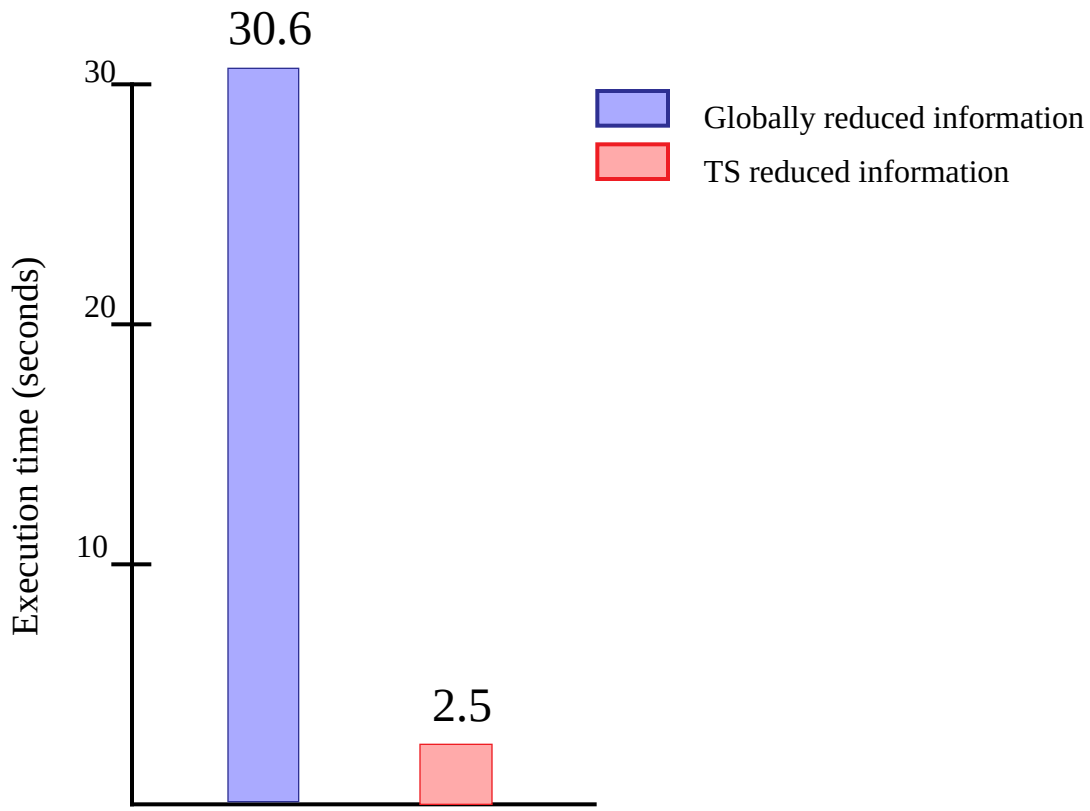


Figure 6.20 Results of Fan-in/ Fan-out Topology of 32 LP's.

6.5. Summary and Conclusions

We have presented a short study of the effects of target-specific synchronization information on both optimistic and conservative PDES synchronization protocols. These empirical results demonstrate the utility of target-specific reductions to both conservative and optimistic PDES protocols. First, the finishing time of conservative PDES's was reduced substantially, in all cases, since the target-specific information about event times of predecessors eliminated artificial dependencies among LP's. Second, the amount of state space in optimistic PDES's was reduced when TSVT was used in fossil collections. The efficient computation of TSVT will support the rollback chip [FUTG92] and the cancelback

protocol [DAFU93], as discussed in Section 5.1.2. It is a topic of future research to investigate both larger systems and a broader class of protocols.

We have concluded that target-specific synchronization information offers significant benefits to conservative PDES's. In a conservative parallel simulation, target-specific synchronization information reduces the finishing time of the simulation. This result is intuitive: near-perfect state information – information that comes close to the true state when the near-perfect information is received – eliminates artificial dependencies and provides more parallelism in a PDES.

We have also concluded that the dissemination of target-specific synchronization information is beneficial to Time Warp-like PDES's. The dissemination of target-specific critical state information allows GVT to be computed on a local basis. We have introduced a local GVT, *target-specific virtual time* or $TSVT_i$, as the smallest possible time to which LP_i can roll back. Any fossils with timestamps earlier than $TSVT_i$ can be collected by LP_i, and $TSVT_i$ is a more accurate commitment horizon for LP_i than GVT. Since TSVT is based only on information relevant to the target LP, thus, in a sense, making it more accurate than GVT, fossils (state information that precedes a TSVT) will be reduced.

In [SRIN93], it was proposed that effective PDES protocols will be those that do adaptive aggressive processing, characterized by controlled aggressiveness, where the benefits of aggressiveness are maximized and its costs are minimized. The impact on state saving is clearly evident. We believe that target-specific virtual time and other target-specific information that can be derived through reduction network techniques will be beneficial to aggressive adaptive algorithms.

7 Conclusions

The results presented in this dissertation include the successful achievement of several of our objections and significant progress toward the achievement of others. We summarize our contributions and discuss avenues of future research.

7.1. Summary of Work

The framework for parallel discrete event simulations [REYN91] was advanced in this thesis, is a novel and efficient combination of both hardware and software to rapidly compute and disseminate reduced values in support of a spectrum of PDES synchronization protocols. It is the first significant research in the computation and dissemination of reduced values to support parallel simulations. As hardware technology progresses and parallel simulation protocols advance, this framework will remain fundamental.

The framework has three major components: 1) small sets of reduced values that describe the state of a parallel simulation, 2) hardware, consisting of a reduction network and general-purpose auxiliary processors, to rapidly compute and disseminate these values, and 3) algorithms that execute on the auxiliary processors so that the reduced values are computed correctly in the reduction network and the parallel simulation executes correctly on the host processors. In this dissertation we have completed research in each of these three areas. We summarize the work in each area now.

As discussed in Section 3.1. and Section 5.1., a set of reduced values to support a parallel simulation can consist of both globally reduced values and target-specific reduced values. We have demonstrated the applicability of both to parallel simulation

synchronization protocols. We have introduced target-specific reductions as an integral part of the next design of the framework hardware. We have made substantial progress in determining the cost of computing target-specific reductions and in developing a scalable design for a parallel target-specific reduction network.

As presented in Chapter 3, we have developed the framework hardware design at three levels: an abstract computation model, a functional design, and a detailed design. We have met our goals, presented in Chapter 1, for the design at each level of the framework hardware:

- Speed — The hardware is designed to compute and disseminate global synchronization information very rapidly (on the order of hundreds of nanoseconds per reduction operation).
- Scalability — The processing time of the hardware to compute global reductions increases logarithmically with the number of processors while the number of components in the hardware increases linearly. The processing time of the parallel target-specific reduction network shown in Figure 5.9 increases logarithmically. It is a topic of future research to determine if we can reduce the complexity of the number of components.
- Adaptability — The design of the interface to the host computing system (See Section 3.5.3.) isolates the design of the rest of the framework hardware from the host computing system. Our prototype system [REPS93] assumes a Sun SBus interface to a Sparc cluster (a network of Sparc-1e's); this design is easily adapted to other host systems.
- Generality — The framework hardware contains programmable ALU's which allow it to be used to support a wide variety of applications. Furthermore, the hardware design allows the selection of two to eight different reduction operations to be computed in support of the application.
- Low cost — A prototype system for four processors has been built for twenty thousand dollars. We expect a production system to cost much less.

We believe these qualities will enable the hardware to support synchronization in general parallel computations as well as parallel simulations.

We have developed algorithms which use a reduction network to acknowledge event messages in support of the computation of a reduced value for the minimum

unreceived message time in the system. This reduced value is one of two reductions necessary to compute global virtual time in an aggressive PDES. The event message acknowledgment algorithms are correct when reductions are being computed asynchronously with the execution of the simulation and with the assumption of state vector loss on the output side of the reduction network. The algorithms guarantee that every message is acknowledged and that the computed global virtual time tracks the actual global virtual time.

7.2. Contributions

This dissertation makes contributions in six important areas for using a reduction-based framework for parallel discrete event simulations. First we have demonstrated the applicability of both global reductions and target-specific reductions to a wide range of parallel discrete event simulations. The characterization of PDES synchronization protocols utilizing reduced values is a new approach to making current parallel simulation synchronization protocols efficient and to developing new synchronization protocols. Protocols that employ the efficient computation of reductions have the potential to be very efficient. Our framework is the first to demonstrate that reduced values can be computed with near-zero overhead to the simulation.

Second, we have provided sound correctness criteria for this framework. The correctness criteria define the computation and dissemination of multiple reduced values in a PDES, where LP's are executing asynchronously, and the computation of reduced values proceeds asynchronously with the execution of the simulation.

Third, we have made significant contributions to the hardware component of the framework at three levels. Each level adheres to the established correctness criteria. At the highest level, we have developed a computation model that decouples the synchronization

processing, i.e., the computation of reduced values, from the execution of logical processes, i.e., event processing and event message transmission and receipt. This abstract model is realizable by many hardware implementations. At the functional level, we have developed a hardware description which employs separate processors in a processor pair for the execution of a logical process and the execution of the synchronization algorithms to support the PDES protocol. Our functional design also employs separate networks for event message transmission and reduction operations. This design offers minimal interaction between the two processors in the processor pair and enhances the efficiency of the simulation. Finally, we have described a detailed design of each component in the functional design. The detailed design includes interfaces between the host and auxiliary processors and between the auxiliary processors and the reduction network. These interfaces preserve the correctness criteria and at the same time minimize the contention at the interface.

Fourth, we have presented several algorithms for acknowledging messages in a reduction network in support of computing global virtual time (and target-specific virtual time) as reductions. In particular, two of the proposed algorithms correctly acknowledge messages in a reduction network where output state vector loss is a property of the hardware: a two phase acknowledgment and a single phase acknowledgment. These two algorithms are correct when the reductions are computed asynchronously with the execution of the simulation. The correctness of the single phase acknowledgment was proven in this thesis. Our presentation of acknowledgment algorithms included discussions and observations on the performance of the simulations executing in conjunction with each algorithm. We have developed the batched acknowledgment enhancement as a method of acknowledging several messages in a single reduced value. Simulations [SRIN92] have shown that batched acknowledgments allow our framework hardware to support smaller

granules of both event message processing times and host network communication latencies. We have implemented both the two-phase acknowledgment (TPA) and the single phase acknowledgment (SPA) on our four-node prototype framework hardware. TPA performs as well as SPA with respect to the execution time of the simulation. Furthermore, the combined sizes of the message lists on any given auxiliary processor is significantly less for the two-phase acknowledgment. We therefore advocate using TPA to acknowledge messages in a PDES implemented on our framework hardware.

Fifth, we have derived and presented best known results for computing and disseminating target-specific reduced values. We have presented two sequential algorithms which solve the target-specific dissemination problem in the general case. The algorithms demonstrate a trade-off between time and space complexity. Both algorithms are encouraging because they show an attainable sub-quadratic time complexity. We have made progress in determining the cost of computing target-specific reductions in parallel. We intend to continue on this course to develop efficient algorithms and networks to compute target-specific reductions or approximations of target-specific reductions.

Finally, we have demonstrated the utility of target-specific reductions to both conservative and optimistic parallel simulation protocols. Target-specific reductions provide near-perfect state information to parallel simulation protocols. As a result, the finishing times of conservative protocols can be reduced greatly by allowing more safe events to be executed concurrently. Similarly, the total average state space of optimistic protocols is reduced significantly. The reduction of state space is encouraging for both hardware and software memory management support for Time Warp, such as the rollback chip and the cancelback protocol. These conclusions were drawn empirically using simulations. We believe that the efficient dissemination of near-perfect state information in the form of target-specific reductions will be beneficial to adaptive aggressive parallel

simulation synchronization protocols, where aggressive processing can be throttled without the risk of the simulation deadlocking.

7.3. Future Research

Although we have developed a PDES framework and demonstrated its utility and feasibility, work remains.

We have implemented two different message acknowledgment algorithms on our prototype hardware. Based on performance results, we have concluded that the two-phase acknowledgment is better. This conclusion is not drawn from a large simulation, however. Simulations [SRIN92] have demonstrated the scalability of the two-phase acknowledgment to simulations of size 32. It is open question how the two message acknowledgment algorithms will compare when the number of processors is increased.

The empirical results presented in Chapter 6 are encouraging results for small simulations. Whether these results are scalable to large and interesting PDES communication topologies is still an open question. We believe that the benefits of target-specific reductions will continue on larger graphs, especially on large graphs that are sparse, i.e., a large number of LP's each with a small number of immediate predecessors.

Probably the most important research to continue is our work with the target-specific reduction problem. Our current best case parallel target-specific reduction network has time complexity of $O(\log n)$ at the cost of $O(n^2)$ switches in the network. We believe that this network complexity is not scalable to thousands of processors. Our goal is to find a scalable and efficient network topology to compute and disseminate target-specific reduced values.

In lieu of a scalable and efficient solution to computing and disseminating perfect target-specific reductions, there is the computation and dissemination of *approximate*

target-specific reductions in parallel simulations. Approaches to exploring this include the generation of a large set of random directed acyclic graphs (DAG's) which represent PDES communication topologies. One could then perform several topological sorts on each DAG, and analyze the DAG with respect to the dissemination of both parallel prefix and interval parallel prefix target-specific reductions. An analysis would include a sensitivity analysis to different topological sorts.

Once the analysis is complete, one could simulate parallel simulations with perfect target-specific information versus approximate target-specific information (computed with both parallel prefix and interval parallel prefix) and report on these results. Both parallel prefix and interval parallel prefix can be computed with *O(n)* components in *O(log n)* time. This empirical result would enable the comparison of the dissemination of approximate target-specific reductions at a low cost with the dissemination of perfect target-specific reductions at a higher cost. If the dissemination of approximate target-specific reductions proves promising, a parallel prefix network or other prefix networks are low-cost approximate target-specific reduction networks. Furthermore, current network designs can be used to implement approximate target-specific reductions. (See Chapter 2.)

7.4. Concluding Remarks

In this thesis we have demonstrated the importance of the efficient computation of reduction operations to parallel discrete event simulations. We have shown how the efficient computation and dissemination of reduction operations enhances existing parallel discrete event simulation synchronization protocols. The computation of target-specific reductions within our framework allows new parallel discrete event simulation protocols which are characterized by adaptive aggressive event processing to be developed. It is our belief that the future direction of parallel discrete event simulation synchronization protocols is the development of protocols that combine properties of both aggressive and

non-aggressive protocols. Our framework clearly determines the direction of the development of such protocols because it reduces the associated costs of deadlock detection, state saving, and synchronization. In sum, the work presented here has forever changed the course of research in parallel simulation protocols in a very favorable direction.

Bibliography

[ABRI91] Abrams, M. and Richardson, D., "Implementing a Global Termination Condition and Collecting Output Measures in Parallel Simulation", *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, Anaheim, California, pp. 86-91, (January 1991).

[AJKS83] Ajtai, M., Komlos, J. and Szemeredi, E., "An O(n(log(n)) Sorting Network", *Proceedings of the 15th Annual Symposium on Theory of Computing*, Boston, Massachusetts, pp. 1-9, (1983).

[AKST94] Akl, S. G. and Stojmenovic, I., "Multiple Criteria BSR: An Implementation and Applications to Computational Geometry Problems", *Proceedings of HICSS*, January 1994.

[AYAN89] Ayani, R., "A Parallel Simulation Scheme Based on Distances Between Objects", *Proceedings of the SCS Multiconference on Distributed Simulation*, Tampa, Florida, pp. 113-118, (March 1989).

[BATC68] Batcher, K. E., "Sorting Networks and Their Applications", *Proceedings of the AFIPS 1968 Joint Computing Conference*, Atlantic City, New Jersey, pp. 307-314, (April 1968).

[BELL90] Bellenot, S., "Global Virtual Time Algorithms", *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, California, pp. 122-127, (January 1990).

[BERR86] Berry, O. "Performance Evaluation of the Time Warp Distributed Simulation Mechanism", PhD Thesis, University of Southern California, Los Angeles, California, May 1986.

[BLEL89] Blelloch, G. E., "Scans as Primitive Parallel Operations", *IEEE Transactions on Computers*, Vol. 38, No. 11, pp.1526-1538, (November 1989).

[BLEL90] Blelloch, G. E., "Prefix Sums and Their Applications", CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1990.

[BROW93] Brown, M. S., "The Hardware Design and Implementation of a Parallel Reduction Network", Master's Thesis, School of Engineering and Applied Science, University of Virginia, Charlottesville, Virginia, 1993.

[BRYA77] Bryant, R. E., “Simulation of Packet Communications Architecture Computer Systems”, MIT-LCS-TR-188, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1977.

[BURO90] Buzzell, C. A. and Robb, M. J., “Modular VME Rollback Hardware for Time Warp”, *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, California, pp. 153-156, (January 1990).

[CHIE94] Chien, A., Personal Communication, May 1, 1994.

[CHLA85] Chandy, K. M. and Lamport, L., “Distributed Snapshots: Determining Global States of Distributed Systems”, *ACM Transactions on Computer Systems*, Vol. 3, No. 1, pp.63-75, (February 1985).

[CHMI79] Chandy, K. M. and Misra, J., “Distributed Simulation: A Case Study in Design and Verification of Distributed Programs”, *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 5, pp. 440-452, (September 1979).

[CHMI81] Chandy, K. M. and Misra, J., “Asynchronous Distributed Simulation via a Sequence of Parallel Computations”, *Communications of the ACM*, Vol. 24, No. 11, pp. 198-206, (April 1981).

[CHMI87] Chandy, K. M. and Misra, J., “Conditional Knowledge as a Basis for Distributed Simulation”, Technical Report 5251:TR:87, Computer Science Department, California Institute of Technology, Pasadena, California, 1987.

[CHSH89] Chandy, K. M. and Sherman, R., “The Conditional Event Approach to Distributed Simulation”, *Proceedings of the SCS Multiconference on Distributed Simulation*, Tampa, Florida, pp. 93-99, (March 1989).

[COKE91] Concepcion, A. I. and Kelly, S. G., “Computing Global Virtual Time Using the Multi-Level Token Passing Algorithm”, *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, Anaheim, California, pp. 63-68, (January 1991).

[DAFU93] Das, S. R. and Fujimoto, R. M., “A Performance Study of the Cancelback Protocol for Time Warp”, *Proceedings of the 1993 Workshop on Parallel and Distributed Simulation*, San Diego, California, pp. 135-142, (May 1993).

[DEGY91] DeBenedictis, E. and Ghosh, S., “A Novel Algorithm for Discrete-Event Simulation”, *IEEE Computer*, Vol. 24, No. 6, pp. 21-33, (June 1991).

[DICK93] Dickens, P. M.,“Analysis of the Aggressive Global Windowing Algorithm”, PhD Thesis, School of Engineering and Applied Science, University of Virginia, Charlottesville, Virginia, January 1993.

[DIRE92] Dickens, P. M. and Reynolds Jr., P. F., “State Saving and Rollback Costs for an Aggressive Global Windowing Algorithm”, Computer Science Report

No. TR-92-18, Department of Computer Science, University of Virginia, Charlottesville, Virginia, June 1992.

[FEKL92] Felderman, R. and Kleinrock, L., "Two Processor Time Warp Analysis: Capturing the Effects of Message Queueing and Rollback/State Saving Costs", Technical Report 920035, Computer Science Department, University of California at Los Angeles, Los Angeles, California, 1992.

[FEKL92b] Felderman, R. and Kleinrock, L., "Two Processor Conservative Simulation Analysis", *Proceedings of the 1992 Western Simulation MultiConference on Parallel and Distributed Simulation*, Newport Beach, California, pp. 169-177, (January 1992).

[FIGP91] Filoque, J. M., Gautrin, E. and Pottier, B., "Efficient Global Computations on a Processors Network with Programmable Logic", Report 1374, Institut National de Recherche en Informatique et en Anutomatique, France, January 1991.

[FOJO88] Fox, G., Johnson, M., Lyzenga, G., *et. al.*, *Solving Problems on Concurrent Processors, Volume 1*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1988.

[FRWW84] Franklin, M. A., Wann, D. F. and Wong, K. F., "Parallel Machines and Algorithms for Discrete-Event Simulation", *Proceedings of the 1984 International Conference on Parallel Processing*, pp. 449-458, (August 1984).

[FUJI87] Fujimoto, R. M., "Performance Measurements of Distributed Simulation Strategies", Technical Report No. UUCS-87-026a, Computer Science Department, University of Utah, Salt Lake City, Utah, November 1987.

[FUJI88] Fujimoto, R. M., "Lookahead in Parallel Discrete Event Simulation", *Proceedings of the 1988 International Conference on Parallel Processing*, University Park, Pennsylvania, pp. 34-41, (August 1988).

[FUJI89] Fujimoto, R. M., "The Virtual Time Machine", *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, Santa Fe, New Mexico, pp. 199-208, (June 1989).

[FUJI89b] Fujimoto, R. M., "Time Warp on a Shared Memory Multiprocessor", *Proceedings of the 1989 International Conference on Parallel Processing*, University Park, Pennsylvania, pp. 242-249, (August 1989).

[FUJI90] Fujimoto, R. M., "Parallel Discrete Event Simulation", *Communications of the ACM*, Vol. 33, No. 10, pp. 30-53, (October 1990).

[FUJI90b] Fujimoto, R. M., "Performance of Time Warp Under Synthetic Workloads", *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, California, pp. 23-28, (January 1990).

[FUTG92] Fujimoto, R. M., Tsai, J. J. and Gopalakrishnan, G.C., "Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp", *IEEE Transactions on Computers*, Vol. 41, No. 1, pp. 68-82, (January 1992).

[GAFN88] Gafni, A., "Rollback Mechanisms for Optimistic Distributed Simulation Systems", *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, California, pp. 61-67, (February 1988).

[GILM88] Gilmer, J. B., "An Assessment of 'Time Warp' Parallel Discrete Event Simulation Algorithm Performance", *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, California, pp. 45-49, (February 1988).

[GIRY88] Gibbons, A. and Rytter, W., *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge, Great Britain, 1988.

[HOSH85] Hoshino, T., *PAX Computer: High-Speed Parallel Processing and Scientific Computing*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1985.

[INTE89] Intel Corporation, *iPSC2 Programmer's Reference Manual*, Intel Scientific Computers, Beaverton, Oregon, October 1989.

[INTE93] Intel Corporation, *Paragon Users's Guide*, Intel Supercomputer Systems Division, Beaverton, Oregon, October 1993.

[IVER62] Iverson, K. E., *A Programming Language*, Wiley, New York, New York, 1962.

[JEBH85] Jefferson, D., Beckman, B., Hughes, S., *et. al.*, "Implementation of Time Warp on the Caltech Hypercube", *Proceedings of the Conference on Distributed Simulation*, San Diego, California, (January 1985).

[JEFF85] Jefferson, D. R., "Virtual Time", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, pp. 404-425, (July 1985).

[JEFF90] Jefferson, D. R., "Virtual Time II: Storage Management in Distributed Simulation", *Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing*, Quebec City, Quebec, Canada, pp. 75-89, (August 1990).

[JESO85] Jefferson, D. and Sowizral, H., "Fast Concurrent Simulation Using the Time Warp Mechanism", *Proceedings of the Conference on Distributed Simulation*, San Diego, California, pp. 63-69, (January 1985).

[JOSC79] Jordan, H. F., Scalabrin, M. and Calvert, W., "A Comparison of Three Types of Multiprocessor Algorithms", *Proceedings of the 1979 International Conference on Parallel Processing*, pp. 231-238, (August 1979).

[KEND92] Kendall Square Research Corporation, *KSR Parallel Programming*, Kendall Square Research Corporation, Waltham, Massachusetts, 1992.

[KIRK92] Kirks, D. J., "A New Approach to Load Sharing", A Research Proposal, Department of Computer Science, University of Virginia, Charlottesville, Virginia, September 1992.

[LAMP79] Lamport, L., "How to Make a Multiprocessor Computer That Correctly Executes Multiprocessor Programs", *IEEE Transactions on Computers*, Vol. C-28, No. 9, pp.690-691, (September 1979).

[LEAD92] Leiserson, C. E., Abuhamdeh, Z. S., Douglas, D. C., *et. al.*, "The Network Architecture of the Connection Machine CM-5", *Proceedings of the Symposium on Parallel and Distributed Algorithms '92*, San Diego, California, (June 1992).

[LIAK93] Lindon, L. F. and Akl, S. G., "An Optimal Implementation of Broadcasting with Selective Reduction", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 3, pp. 256- 269, (March 1993).

[LILA89] Lin, Y. B. and Lazowska, E. D., "Exploiting Lookahead in a Parallel Simulation", Technical Report 89-10-06, Department of Computer Science, University of Washington, Seattle, Washington, October 1989.

[LILA89b] Lin, Y. B. and Lazowska, E. D., "Determining the Global Virtual Time in a Distributed Simulation", Technical Report 90-01-02, Department of Computer Science, University of Washington, Seattle, Washington, December 1989.

[LILA89c] Lin, Y. B. and Lazowska, E. D., "Optimality Considerations for "Time Warp" Parallel Simulation", Technical Report 89-07-05, Department of Computer Science, University of Washington, Seattle, Washington, July 1989.

[LILA89d] Lin, Y. B. and Lazowska, E. D., "A Study of Time Warp Rollback Mechanisms", Technical Report 89-09-07, Department of Computer Science, University of Washington, Seattle, Washington, November 1989.

[LILA89e] Lin, Y. B. and Lazowska, E. D., "The Optimal Checkpoint Interval in Time Warp Parallel Simulation", Technical Report 89-09-04, Department of Computer Science, University of Washington, Seattle, Washington, 1989.

[LILA90] Lin, Y. B. and Lazowska, E. D., "Optimality Considerations for Time Warp Parallel Simulation", *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, California, pp. 29-34, (January 1990).

[LILA90b] Lin, Y. B. and Lazowska, E. D., "Reducing the State Saving Overhead for Time Warp Parallel Simulation", Technical Report 90-02-03, Department of Computer Science, University of Washington, Seattle, Washington, 1990.

[LIMA85] Livny, M. and Manber, U. “Distributed Computation Via Active Messages”, *IEEE Transactions on Computers*, Vol. C-34, No. 12, pp.1185-1190, (December 1985).

[LITR90] Liu, L. Z. and Tropper, C., “Local Deadlock Detection in Distributed Simulations”, *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, California, pp. 64-69, (January 1990).

[LOCU88] Lomow, G., Cleary, J., Unger, B., *et. al.*, “A Performance Study of Time Warp”, *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, California, pp. 50-55, (February 1988).

[LUBA88] Lubachevsky, B. D., “Bounded Lag Distributed Discrete Event Simulation”, *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, California, pp. 183-191, (February 1988).

[LUBA89] Lubachevsky, B. D., “Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks”, *Communications of the ACM*, Vol. 32, No. 1, pp. 111-123, (January 1989).

[LUSW89] Lubachevsky, B., Shwartz, A. and Weiss, A. “Rollback Sometimes Works … If Filtered”, *Proceedings of the 1989 Winter Simulation Conference*, Washington, DC, pp. 630-639, (December 1989).

[MCGR93] McGraw, R. M., “The Design and Test of Hardware Support for a Parallel Reduction Network”, Master’s Thesis, School of Engineering and Applied Science, University of Virginia, Charlottesville, Virginia, 1993.

[MISR86] Misra, J., “Distributed Discrete-Event Simulation”, *ACM Computing Surveys*, Vol. 18, No. 1, pp. 39-65, (March 1986).

[MIMI84] Mitra, D. and Mitrani, I. “Analysis and Optimum Performance of Two Message-Passing Parallel Processors Synchronized by Rollback”, *PERFORMANCE ‘84*, Elsevier Science Pub (North Holland), pp. 35-51, 1984.

[NICO84] Nicol, D. M., “Synchronizing Network Performance”, Master’s Thesis, School of Engineering and Applied Science, University of Virginia, Charlottesville, Virginia, January 1984.

[NICO88] Nicol, D. M., “High Performance Parallelized Discrete Event Simulation of Stochastic Queueing Networks”, *Proceedings of the 1988 Winter Simulation Conference*, San Diego, California, pp. 306-314, (December 1988).

[NICO88b] Nicol, D. M., “Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks”, *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, pp. 124-137, (1988).

[NICO90] Nicol, D. M., "The Cost of Conservative Synchronization in Parallel Discrete Event Simulations", NASA Contractor Report 182034, Institute for Computer Applications in Science and Engineering, NASA Langley, Hampton, Virginia, May 1990.

[NICO91] Nicol, D. M., "Performance Bounds on Parallel Self-Initiating Discrete-Event Simulations", *ACM Transactions on Modeling and Computer Simulation*, Vol. 1, No. 1, pp. 24-50, (January 1991).

[NICO93] Nicol, D. M., "The Cost of Conservative Synchronization in Parallel Discrete Event Simulations", *Journal of the ACM*, Vol. 40, No. 2, pp. 304-333, (April 1993).

[NIFU92] Nicol, D. and Fujimoto, R., "Parallel Simulation Today", to appear in *The Annals of Operations Research*.

[NIRE84] Nicol, D. M. and Reynolds Jr., P. F., "Problem Oriented Protocol Design", *Proceedings of the 1984 Winter Simulation Conference*, Dallas, Texas, pp. 471-474, (December 1984).

[OWGR76] Owicki, S. and Gries, D., "An Axiomatic Proof Technique for Parallel Programs I", Acta Informatica, Vol. 6, pp. 319-340, 1976.

[PANC92] Pancerella, C. M., "Improving the Efficiency of a Framework for Parallel Simulations", *Proceedings of the 1992 Western Simulation MultiConference on Parallel and Distributed Simulation*, Newport Beach, California, pp. 22-29, (January 1992).

[PARE93] Pancerella, C. M. and Reynolds Jr., P. F., "Disseminating Critical Target-specific Synchronization Information in Parallel Discrete Event Simulations", *Proceedings of the 1993 Workshop on Parallel and Distributed Simulation*, San Diego, California, pp. 52-59, (May 1993).

[PEWM79] Peacock, J. K., Wong, J. W. and Manning, E., "Distributed Simulation Using a Network of Processors", *Computer Networks 3*, North-Holland Publishing Company, pp. 44-56, 1979.

[PEWM79b] Peacock, J. K., Wong, J. W. and Manning, E., "A Distributed Approach to Queueing Network Simulation", *Proceedings of the 1979 Winter Simulation Conference*, pp. 399-406, (December 1979).

[PFBG85] Pfister, G. F., Brantley, W. C., George, D. A., *et. al.*, "The IBM Research Parallel Prototype (RP3): Introduction and Architecture", *Proceedings of the 1985 International Conference on Parallel Processing*, St. Charles, Illinois, pp. 764-771, (August 1985).

[RABJ88] Ranade, A. G., Bhatt, S. N. and Johnsson, S. L., "The Fluent Abstract Machine", YALEU/Department of Computer Science/Technical Report-573, Department of Computer Science, Yale University, New Haven, Connecticut, January 1988.

[REMM88] Reed, D. A., Malony, A. D., and McCredie, B. D., “Parallel Discrete Event Simulation Using Shared Memory”, *IEEE Transactions on Software Engineering*, Vol. 14, No. 4, pp. 541-553, (April 1988).

[REPA92] Reynolds Jr., P. F. and Pancerella, C. M., “Hardware Support for Parallel Discrete Event Simulations”, Computer Science Report No. TR-92-08, Department of Computer Science, University of Virginia, Charlottesville, Virginia, April 1992.

[REPS92] Reynolds Jr., P. F., Pancerella, C. M. and Srinivasan, S., “Making Parallel Simulations Go Fast”, *Proceedings of the 1992 Winter Simulation Conference*, Alexandria, Virginia, pp. 646-655, (December 1992).

[REPS93] Reynolds Jr., P. F., Pancerella, C. M. and Srinivasan, S., “Design and Performance Analysis of Hardware Support for Parallel Simulations”, in a special issue of *Journal of Parallel and Distributed Computing* on Parallel and Distributed Simulation, Vol. 18, No. 4, pp. 435-453, (August 1993).

[REWW89] Reynolds Jr., P. F., Williams, C. and Wagner, R. R., “Parallel Operations”, Computer Science Report No. TR-89-16, Department of Computer Science, University of Virginia, Charlottesville, Virginia, December 1992.

[REWW92] Reynolds Jr., P. F., Williams, C. and Wagner, R. R., “Empirical Analysis of Isotach Networks”, Computer Science Report No. TR-92-19, Department of Computer Science, University of Virginia, Charlottesville, Virginia, June 1992.

[REYN82] Reynolds Jr., P. F., “A Shared Resource Algorithm for Distributed Simulation”, *Proceedings of the 9th Annual Symposium on Computer Architecture*, Austin, Texas, pp. 259-266, (April 1982).

[REYN88] Reynolds Jr., P. F., “A Spectrum of Options for Parallel Simulations”, *Proceedings of the 1988 Winter Simulation Conference*, San Diego, California, pp. 167-174, (January 1991).

[REYN91] Reynolds Jr., P. F.,”An Efficient Framework for Parallel Simulations”, *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, Anaheim, California, pp. 167-174, (January 1991).

[REYN92] Reynolds Jr., P. F.,”An Efficient Framework for Parallel Simulations”, *International Journal in Computer Simulation*, Vol. 2, No. 4, (1992).

[SAMA85] Samadi, B., “Distributed Simulation, Algorithms, and Performance Analysis”, PhD Thesis, Computer Science Department, University of California at Los Angeles, Los Angeles, California, January 1985.

[SBUS90] Sun Microsystems, *SBus Specification B.0*, Sun Microsystems, Inc., Mountain View, California, 1990.

[SCHW80] Schwartz, J. T., "Ultracomputers", *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 4, pp. 484-521, (October 1980).

[SOBW88] Sokol, L. M., Briscoe, D. P. and Wieland, A. P., "MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution", *Proceedings of the SCS Multiconference on Distributed Simulation*, San Diego, California, pp. 34-42, (February 1988).

[SRIN92] Srinivasan, S., "Modeling a Framework for Parallel Simulations", Master's Thesis, School of Engineering and Applied Science, University of Virginia, Charlottesville, Virginia, May 1992.

[SRIN93] Srinivasan, S., "Adaptive Synchronization Algorithms for Parallel Discrete Event Simulation", A Research Proposal, Department of Computer Science, University of Virginia, Charlottesville, Virginia, November 1993.

[SRRE93] Srinivasan, S. and Reynolds Jr., P. F., "Hardware Support for Aggressive Parallel Discrete Event Simulation", Computer Science Report No. TR-93-07, Department of Computer Science, University of Virginia, Charlottesville, Virginia, January 1993.

[SRRE93b] Srinivasan, S. and Reynolds Jr., P. F.,"Non-interfering GVT Computation Via Asynchronous Global Reductions", *Proceedings of the 1993 Winter Simulation Conference*, Los Angeles, California, pp. 740-749, (December 1993).

[STON90] Stone, H. S., *High-Performance Computer Architecture*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

[TANE89] Tanenbaum, A. S., *Computer Networks - Second Edition*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1989.

[THIN92] Thinking Machines Corporation, *The Connection Machine CM-5 Technical Summary*, Thinking Machines Corporation, Cambridge, Massachusetts, January 1992.

[TOGA93] Tomlinson, A. I. and Garg, V. K., "An Algorithm for Minimally Latent Global Virtual Time", *Proceedings of the 1993 Workshop on Parallel and Distributed Simulation*, San Diego, California, pp. 35-42, (May 1993).

[TUXU92] Turner, S. and Xu, M., "Performance Evaluation of the Bounded Time Warp Algorithm", *Proceedings of the 1992 Western Simulation MultiConference on Parallel and Distributed Simulation*, Newport Beach, California, pp. 117-126, (January 1992).

[WALA89] Wagner, D. B. and Lazowska, E. D., "Parallel Simulation of Queueing Networks: Limitations and Potentials", *Proceedings of the 1989 ACM SIGMETRICS and PERFORMANCE '89: International Conference on Measurement and Modeling of Computer Systems,* Berkeley, California, pp. 146-155, (May 1989).

[YUGD91] Yu, M. L., Ghosh, S. and DeBenedictis, E., *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, Anaheim, California, pp. 39-43, (January 1991).