Time Warp Simulation on Multi-core Processors and Clusters

A thesis submitted to the

Division of Research and Advanced Studies
of the University of Cincinnati

in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in the School of Electric and Computing Systems
of the College of Engineering and Applied Sciences

August xx, 2015

by

Doug Weber
BSEE, University of Cincinnati, 2014

Thesis Advisor and Committee Chair: Dr. Philip A. Wilsey

Abstract

Parallel Discrete Event Simulation (PDES) systems have traditionally been designed for either clusters using message passing communication *or* for shared-memory multiprocessors using shared data structures but not both. The fine granularity of PDES appications makes the communication latency of message passing a target issue. Message passing, however, is necessary to scale up the computational power and memory required for larger simulation models that would be limited in a single shared-memory processor. Shared memory communication, on the other hand, means that the latency of exchanging events is significantly reduced since buffer copying and network communication can be avoided.

To get the best of both forms of communication, WARPED2, a PDES simulation kernel which implements the time warp mechanism, uses a hybrid approach where a set of mulithreaded processes communicate with message passing but threads within each process communicate through shared data structures. If the logical processes (LPs) of the simulation model are partitioned well, then the interprocess communication can be minimized and the simulation models can scale to more reasonable levels.

A hybrid communication approach, however, adds some significant problems that make it hard to optimize the simulation mechanism to get the best performance. The first problem that must be dealt with, which is common to any multithreaded application, is the high cost of thread synchronization to protect shared data structures. This problem not only adds significant overheads in the simulation mechanism (dispatching of local events and processing events) but with the standard runtime and message passing libraries as well. A few of the solutions what will described include separating data structures for thread local use to eliminate synchronization, using spinlocks to reduce lock aquisition latency, using alternative multithreaded memory allocators, and using a separate thread to handle message passing communication.

The second, and more time warp specific problem arises with a hybrid communication model is with developing asynchronous GVT and termination detection algorithms which are usually designed for either shared-memory multiprocessors *or* message passing systems, but not both. For both GVT and termination detection, a two level approach for each will be discussed and analyzed.

Experimental analysis will be performed on SMP nodes and clusters to gain a better understanding of the tunable performance, communication, and memory parameters of WARPED2. Preliminary results indicate that any optimization that minimizes thread synchronization is usually preferrable on a single SMP node even though rollbacks usually increase and efficiency decreases. On a cluster, minimizing remote communication is the key to minimizing rollbacks and increasing performance. Funneling communication to a dedicated thread has been shown to be more effective than a multi-threaded communication model which means that a high synchronization costs in the message passing library is more limiting than an increased communication latency.

Acknowledgments

First and foremost, I would like to thank Dr. Wilsey for his guidance throughout my research experience. It has been a great experience working with him and I have learned more than I ever thought I would.

I would also like to thank Sounak Gupta for his work with the pending event set in WARPED2 and AJ Alt for his work with the initial development of WARPED. My research would not have been possible without them. Also, thanks to all previous students in the Experimental Computing Lab who contributed to the development of the original WARPED simulation kernel.

Thank you to my family for being supportive throughout my entire education at the University of Cincinnati.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Chapter 1

Introduction

Many physical systems can be described by events that occur in discrete time intervals such as communication networks, digital logic circuits, transportation systems, or disease outbreaks. To gain a better understanding of these systems, researchers develop models of the systems and perform simulations on computing platforms. These *Discrete Event Simulation* (DES) models can take a long time to simulate large, complex systems by sequentially processing events which has ultimately led researchers to design parallel algorithms to run simulations on parallel computing platforms. The field of study that deals with parallel algorithms to speed up discrete event simulations is known as Parallel Discrete Event Simulation (PDES).

Parallel algorithms can be written to target many different architectures that support parallelism such as shared-memory multiprocessors, clusters, or any other type of system that support parallel execution. Furthermore, communication between concurrent workers in the system can be achieved by using shared data structures or by passing explicit messages between workers.

The Time Warp Mechanism [4], which is the focus of this text, is one approach to PDES where causal dependencies of events are not enforced before processing an event. A method of undoing erroneous computation and cancelling incorrectly sent events must be implemented in a time warp system to ensure causality of events.

Some time warp systems are designed for only shared memory multiprocessors. These systems minimize communication latencies and allow very fast, simple algorithms because everything can run in a single address space and use shared data structrures with pointers to prevent unnecessary copying. However, they are still limited by the computational power and memory size of the machine. Other time warp systems are completely based on a message passing scheme. These systems can be scaled to any number of machines using interconnection networks as a means of exhchanging messages. However, this approach introduces high communication latencies.

The WARPED2 simulation kernel, which is a reimplementation of the original WARPED simulation kernel, uses a hybrid approach by using shared memory between a set of *worker threads* in a single process to eliminate communication overheads, but uses message passing between processes to allow the system to scale to larger sizes. A dedicated *manager thread* handles all global operations between such as GVT and termination detection. Figure 1.1 illustrates the communication model that is used in WARPED2.

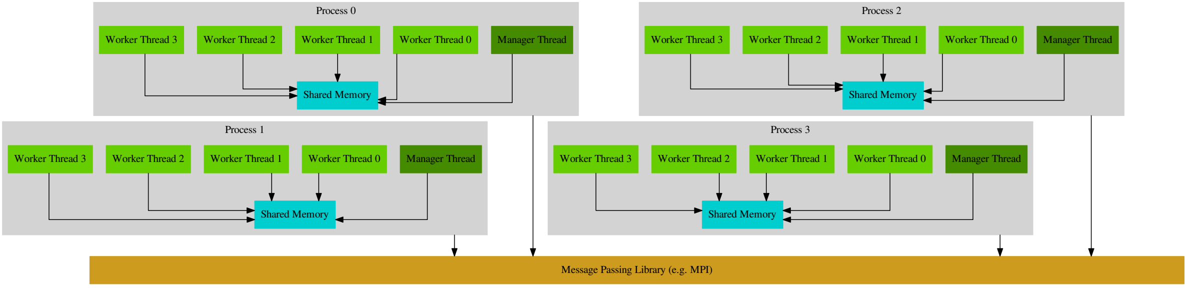


Figure 1.1: Communication Model of WARPED2

This communication model not only allows for high scalability and reduced communication overhead within processes, but allows time warp simulations to follow the critical path better since the worker threads can share a scheduling data structure to process events from. In WARPED2 this scheduling data structure is call an *LTSF Queue*. Fewer LTSF queues creates a bottleneck since multiple worker threads cannot access them simultaneously without race conditions. On the other hand, more LTSF queues allows more concurrency but spreads out the critical path of execution, allowing more rollbacks so the efficiency decreases.

To reduce communication overheads in both message passing and shared memory communication, partitioning the work between processes and threads will also be explored. Partitioning the work in a parallel discrete event simulation is achieved by simply partitioning the LPs. In WARPED2, a two phase partitioning scheme will be explored. The first phase is for partitiong LPs among processes in order to minimize interprocess communication and the second phase is for further partioning the LPs in a process among LTSF queues which will localize LP communication among worker threads, allowing better cache locality, and lowering rollbacks. To further reduce the side effects of message passing communication, messages that are sent to the same process are aggregated and sent together as a single larger message.

The combination of shared memory and message passing communication does create some complications in the implementation of a time warp system. GVT and termination detection algorithms are usually designed for either shared memory or message passing but not both. Using a single message passing algorithm between worker threads and processes would mean that some kind of message passing scheme would have to be implemented for worker threads to communicate. This scheme would still use shared memory for message communication but would still have the overheads of explicit message passing. On the contrary, it is possible to have a single shared memory algorithm that extends to distributed memory systems but would require more complex algorithms that would either not scale efficiently or would require extra synchronization and communication. For these reasons, the GVT and termination algorithms developed for WARPED2 include both a message passing algorithm and a shared memory algorithm that work in conjunction.

1.1 Thesis Overview

The remainder of this thesis is organized as follows:

Chapter 2 contains some background information on parallel simulation and parallel computing that is used in this thesis.

Chapter 3 reviews several of the prominent parallel simulation kernels that use the Time Warp synchronization protocol. The software architecture and target compute platforms for each is described.

Chapter 4 introduces the software architecture and modeling API for the WARPED2 simulation kernel.

Chapter 5 provides an overview of experiments for single SMP nodes and clusters and the models that are used in the experiments.

Chapter 6 describes the importance of efficient message passing communication and analyzes different approaches. Some preliminary results are shown and analyzed to determine the best communication model, partitioning scheme and number of aggregated messages.

Chapter 7 describes the basic event scheduling algorithms in WARPED2 as well as the data structures used and the organization of the data structures. Some parameters that can be used to tune the performance of the simulation will also be discussed.

Chapter 8 analyzes the probem of GVT and termination detection in distributed systems and describes various algorithms that have been used as well as algorithms used in WARPED2.

Chapter 9 analyzes techniques for managing memory efficiently. A brief overview of state saving techniques, fossil collection, and memory allocation will be analyzed and some of the memory tuning parameters in WARPED2 will be discussed.

Finally, Chapter 10 contains some concluding remarks and suggestions for future research.

Chapter 2

Background

This chapter first describes some of the basics of Parallel Discrete Event Simulation (PDES) with a focus on the Time Warp mechanism. Then a brief overview of parallel architectures and parallel communication models and their strengths and weaknesses are compared.

2.1 Discrete Event Simulation

Discrete Event Simulation (DES) is a method of modeling the execution of a physical system with a sequence of events that occur at discrete time intervals. A Discrete Event Simulation typically contains three main data structures

State variables: A set of variables that describe the current state of the system.

Simulation Clock: A clock to measure the progress of the simulation and determine the order of event processing.

Pending Event Set: A set of future events that are waiting to be procesed.

A *Simulation Model* describes a physical system by a set of *Logical Processes* (LPs). Each LP corresponds to a physical process that is part of the physical system. The LPs interact with timestamped events that dictate the simulation time that the event should be processed. With each event that occurs, and only when an event occurs, the state of the system is updated.

In a *Sequential* Discrete Event Simulation only one event is processed at a time. All pending events are kept in a single list which is sorted by timestamp. The next event to be processed is always the one with lowest timestamp. Each successive event updates the state of the system, advances the simulation clock, and possibly produces new future events. This is clearly not very efficient for large simulations. This method can be improved by realizing that events for different LP's are independant and will only affect the state for a single LP.

2.2 Parallel Discrete Event Simulation

Parallel Discrete Event Simulation (PDES) is a method running a discrete event simulation on a parallel computer which could be a shared-memory multiprocessor or a cluster, or a combination of both. In a parallel discrete event simulation the state of the system is usually split among the logical processes so that each one contains a portion of system's state without any sharing of state variables [5]. In addition to each logical process having it's own separate state, the logical processes also have seperate simulation clocks and pending event sets. Event's from different LP's can then be processed concurrently without the need to worry about sharing state variables and the model can be viewed as concurrent processes operating independantly which contribute to the overall progression of the simulation. This has the potential to increase performance significantly; However, it is possible that events at a receiving LP can be received and processed out of order, violating causality. These *causality errors* can occur because of the independant nature of the LPs and because the LP's can be processing events at different rates. Causality errors can produce incorrect changes in state variables and incorrect events to be sent to other LP's. Parallel Discrete Event Simulation techniques can be categorized in terms of how causality errors are handled. *Conservative* approaches use methods to detect when possible causality errors might occur and prevent them from ever occuring. *Optimistic* approaches, on the other hand, allow causality errors to occur but use methods to detect and recover from the errors. Generally, the simulation models can be developed without the knowledge of the underlying simulation mechanism. The simulation mechanism is usually implemented in a self-contained module which provides an API for the models and is commonly referred to as the *kernel* or *executive*. For the remainder of this text, only optimistic methods will be discussed, specifically the Time Warp mechanism which is the most widely used optimistic mechanism used in practice.

2.2.1 Time Warp

The Time Warp mechanism is an optimistic method of simulation which is based on the virtual time paradigm [4]. *Virtual Time* provides a method of ordering events in distributed systems which are not described by real time such as a simulation. When used for parallel discrete event simulation, Virtual Time is synonymous with simulation time. The current time of an LP's simulation clock in Time Warp is called the *Local Virtual Time* (LVT).

When a causality error is detected at an LP (next event to be processed is less than the LVT) the effects of the incorrectly processed event(s) must be undone. The process of undoing the effects is called a *rollback* and the event that triggers a rollback is called a *straggler event*. When a straggler event is detected at an LP, the first step taken during the rollback is to restore the LP's state back to a previous state before the incorrect event(s) were processed. Then the LP must "unsend" the events that were incorrectly sent by sending *negative events* or *anti-messages*. The negative events, when received by the receiving LP will stop the corresponding positive event from being processed or if the corresponding positive message has already been processed at the receiving LP then that LP must also rollback. This processes recursively occurs until all causality errors are corrected. The negative messages are never processed as normal events but serve only to annhilate an generated event produced by an incorrectly processed event.

Jefferson [4] describes how to support rollbacks with three main data structures:

1. Input Queue
2. Output Queue
3. State Queue

Every LP will have a seperate input queue, output queue, and state queues. The input queue contains the unprocessed and processed events for the LP that it belongs to. The input queue must be sorted in timestamp order and the LP's must always process from the lowest unprocessed event. The LVT is always the largest timestamped processed event and is used to detect a straggler event. The output queue contains the events that have been sent by the LP that it belongs to which will allow the LP to send anti-messages during a rollback. The state queue contains previous states of the LP and allows the proper states to be restored during a rollback.

The *Global Virtual Time* (GVT) of the simulation at a given point during the simulation is the minimum timestamp of all unprocessed events including the events that have been sent but not received [4]. There are numerous algorithms for determining the GVT which will be discussed further in chapter8. LP's cannot send events that are less than their LVT value and so the GVT acts as a lower bound on how far a rollback can occur. Because no LP's will ever rollback past the GVT value, it is often used to free memory that is no longer needed for the events in the input and output queues and the states in the state queue that have timestamps less than the GVT as well as committing I/O operations that cannot be undone. This process of freeing memory and committing I/O operations is known as *fossil collection*. Fossil collection does not have to be based on the GVT and several other methods of fossil collection have been developed which will also be discussed more in chapter 9.

The need to save the state of the LP's is one of the fundamental overheads in the Time Warp mechanism in terms of both the amount of time it takes to copy the LP's states to save them and the amount of memory that must be used to store them. Because of this, a number of different approaches have been developed to reduce the overhead of state saving such as periodic state saving, incremental state saving, and reverse computation. These methods will also be discussed more in chapter 9.

2.3 Parallel Systems Architectures

Systems that support parallel processing come in many forms. They can generally be characterized by how processors and memories are grouped together. A shared memory system is a single machine which shares a common address space for all processes and can have a either a single physical memory unit or multiple memory units. A cluster is a system comprised of multiple machines with seperate process address spaces connected over an interconnection network.

2.3.1 Shared Memory Multiprocessor Systems

A Symmetric Multiprocessor (SMP) is a type of shared memory multiprocessor system where each processor has uniform access to a single shared memory through a common bus. SMP systems cannot scale very large due to increasing memory contention as the number of processors increasing and for this reason, they usually have 8 or fewer processors. To increase available bandwidth in the system, each processor usually has one or two levels of private caches as well as a shared cache which act as a way limit the number of memory accesses. Figure 2.1 illustrates what a typical SMP system looks might look like.

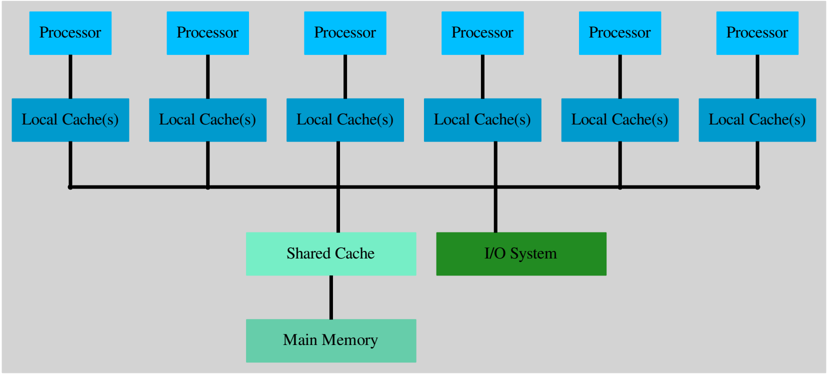


Figure 2.1: SMP System

A Distributed Shared Memory sytem, also known as a Non-Uniform Memory Access (NUMA) system has mulitple memories that are distributed. In these systems, the memories are still shared between processors, but access to different memories may have different access times. An interconnection network connects processors and memories together with one or more processors per memory. Because memory access times can vary, software on NUMA systems usually try to keep memory accesses local to a processor. However, NUMA systems can scale much larger than SMP systems because contention to single memory does not necessarily increase with an increasing number of processors. Figure 2.2 illustrates what a typical NUMA system might look like.

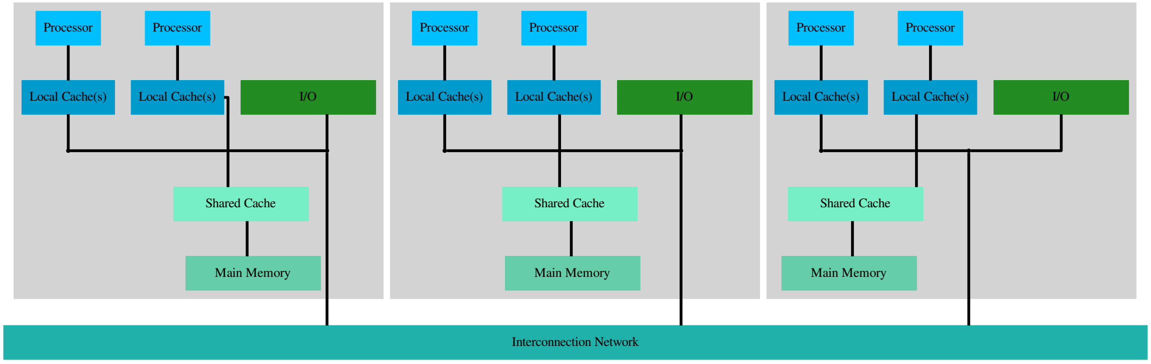


Figure 2.2: NUMA System

2.3.2 Clustered Systems

A Beowulf Cluster is a type of cluster which appears to the user as a single machine but is actually a loosely coupled set of machines connected together over a local network. A single program is executed by all machines concurrently by launching multiple processes on each machine. A program written for a beowulf cluster typically use some type of message passing to communicate among processes and typically uses parallel communication software such as MPI or PVM. Figure 2.3 illustrates a common realization of a beowulf cluster.

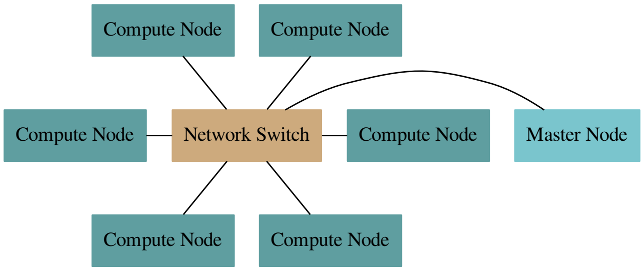


Figure 2.3: Beowulf Cluster

2.4 Parallel Systems Communication

Parallel applications are composed of multiple workers that operate independantly in parallel and may have to exchange information. The workers in a parallel application can be a process, thread, or any other type of execution context and can be within a shared memory system or a cluster or any type of parallel system. Workers can communicate by either sending explicitly passing messages to each other by means of well defined message formats or by using shared data structures that all workers can access. The former method of communication is known as *message passing* and the latter method of communication is known as *shared memory* communication. Both communication methods are fundamentally different and both have strengths and weaknesses.

2.4.1 Message Passing

In a message passing system, workers are completely isolated in different address spaces and communicate only through serialized messages. The formats of the messages must be defined so that the message can be serialized and deserialized by the sender and the reciever, respectively. Message passing can either be synchronous or asynchronous. With synchronous message passing, the send/receive operations must be done in a specific order so that the sender/receiver workers operate together in a synchronized fashion. The send operation at the sender will block until the message is received at the reciever and the receive operation at the receiver will block until the message is fully received. That means the every worker must follow a predictable communication pattern. Workers cannot continue other operations during communication operations and may slow down the whole system. On the other hand, asynchronous communication allows workers to start a send and recieve operations and immediately continue without blocking. To allow this, temporary queues must be used to hold pending operations. The workers do not have to follow a predictable communication pattern in this case because the messages will be queued and can be processed at any time and in any order. The main advantage of message passing, in general, is that the number of workers can be scaled to any size as long as the work is partitioned in the right way. Also, the workers can execute in address spaces on different machines and communicate over a local network connection. The biggest disadvantage, however, is an increased communication latency which can be vary large compared to the speed of computation. This makes message passing especially hard for fine-grained parallel applications.

An illustration of simple message passing is shown in figure 2.4 (a).

Message Passing Interface (MPI)

MPI [6] is an extensive message passing API specification for parallel applications and supports both synchronous and asychronous forms of communication. It is a standard specification for developers and MPI users and many current implementations exist. The most widely used implementations used in practice include MPICH and OpenMPI.

2.4.2 Shared Memory

The workers in a parallel application can also share a common address space and communicate through shared data structures. The producer worker will insert data directly into the data structure and the consumer worker will remove the data and use it. This takes much less time to transmit data than with a message passing scheme. Furthermore, pointers can be used to eliminate copying overheads. Access to the shared data structures, however, must be protected so that multiple workers do not simultaneously access the same data which could cause unpredictable results. Access to the data structures is usually enforced using lock synchronization mechanisms that protect entire sections of code that are executed by different workers and can end up accessing the same data structures such as mutexes or semaphores. Shared memory data structures may suffer from performance if lots of workers contend for the lock at the same time. For this reason, it is very hard to scale systems that use only shared memory as a means of communication. An illustration of a simple shared memory system is show in figure 2.4 (b).

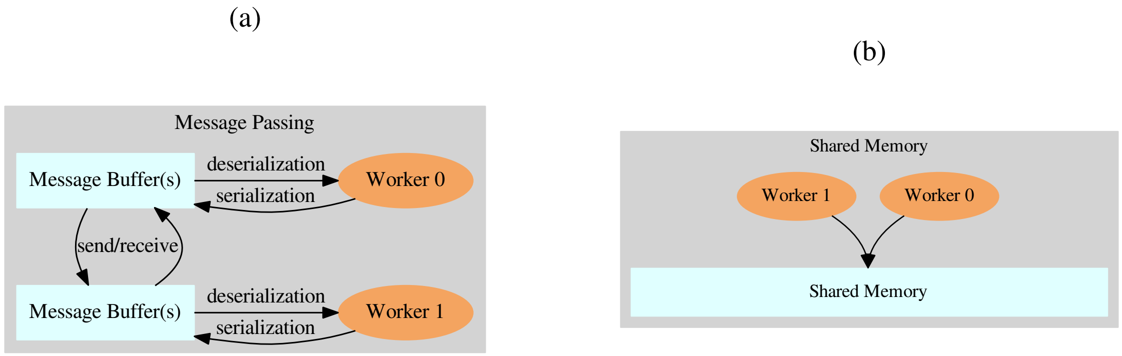


Figure 2.4: Message Passing and Shared Memory Communication

Chapter 3

Related Work

This chapter give an overview of some of the most popular Time Warp implementations. For each impementation the design will be described at a high level with a focus on the target architecture. The featured data structures and algorithms for each will also be described as well as the strengths and weaknesses.

3.1 Georgia Tech Time Warp (GTW)

Georgia Tech Time Warp was a general purpose Time Warp Simulator designed specifically for shared memory multiprocessors. It is not used anymore, however, because it was only written to target architectures that are now obselete such as the SparcStation and SGI PowerChallenge. Although GTW is not used any more, it's design has influenced the design of other simulators that are still used widely in practice. GTW simulation models run in a single multi-threaded processes and use only shared data structures to communicate between threads which are bound to single processor.

The LP's for all models are statically allocated to a single thread so that events for the LP's are processed only on a single processor. The pending event set is distributed among threads with each having its own data structures for the set of LPs that belong to it. Since the threads can only run on a single processor, the data structures also belong to a single processor. The pending event set for each processor consists of three main data structures listed below [1]:

1. The *Message Queue* is a linked list that contains positive messages that are destined for the LP's

mapped to the owning processor. Access to the message queue must be synchronized because it can be accessed by tasks running on any processor.

2. The *Cancel Queue* is a linked list that serves exactly the same purpose as the message queue except that it contains only negative messages(anti-messages). Access to this queue must also be synchronized.

3. The *Event Queue* is used to hold unprocessed and processed events and is directly used to schedule events to be processed. The event queue is actually made up of different data structures, one for processed events and one for unprocessed events. The processed events are contained within a doubly linked list and the unprocessed events are contained within a priority queue which can be configured to be either a calendar queue or a skew heap depending on a user configuration.

When messages are sent between LP's, they are inserted directly in the message queue or the cancel queue depending on whether they are positive or negative. Each thread processes events by first moving events from the message queue to the event queue and processing rollbacks. Then, the messages from the cancel queue are removed and cancellations are processed and any more rollbacks are processed. One or more of the smallest events from the event queue are then processed and added to the processed event list. This procedure is repeated over and over again by all processors. Psuedocode for the main event processing loop in GTW is show in figure 1.

```
while eventQ is not empty do
    move messages from MsgQ to EvQ and process any rollbacks
    remove anti-messages from CanQ, process annihilations and rollbacks
    remove N smallest timestamped events E from EvQ
    process the N events
```

Algorithm 1: GTW Main Event Processing Loop [1] [2]

To avoid accessing the message queues and cancel queues too often, which is a contention point, a larger value of N can be used. GTW calls this batch processing and N is the batch interval. With batch processing, multiple events will be processed at a time without processing any rollbacks or cancellations.

Since GTW is only uses shared memory communication, anti-messages do not need to be explicitly sent. Only a pointer to the event that needs to be cancelled is necessary. Fujimoto calls this method *direct* *cancellation*. Also by only using shared memory, GVT can be calculated very quickly using shared data structures instead of passing messages around to all processors. The downfall of GTW, however is that it was limited to only a single multiprocessor machine and it was only designed and optimized for specific architectures.

GTW also imposed some unnecessary requirements for the developer of particular simulation models. The partitioning of the LP's among processors must be done in the simulation model during initialization. That means that the model developer must understand some the features of the underlying architecture such as the number of processors, to effectively partition the LP's. Furthermore, the initial partitioning of the LP's is hard to dynamically balance during the simulation because there are no seperate input queues for each LP but rather a single message queue to hold all unprocessed events for each processor.

3.2 Clustered Time Warp (CTW)

Clustered Time Warp [7] (CTW) uses a hybrid sequential/time-warp approach by processing events within a *cluster* of LP's sequentially and using the Time Warp mechanism between the clusters. This design was chosen because it works well for digital logic simulation which tend to have localized computation within a group of LP's. Furthermore, digital logic simulation tends to have low computational granularity and lot's of LP's which can lead to a lot of rollbacks and a large memory footprint in a traditional Time Warp simulator. CTW is implemented for shared memory multiprocessors but only uses shared memory for use with a custom message passing system so that it can better support NUMA architectures. It is not designed for use on a a beowulf cluster.

Each cluster of LPs has a timezone table, an output queue, and a set of LP's which each have an input queue and a state queue. The timezones in the timezone table are divided by timestamps of the events received from LP's on different clusters. Whenever an event is received from a remote cluster, a new timezone is added. Only a single output queue is needed per cluster because anti-messages can only be sent between clusters and not between LP's on the same cluster.

When a straggler event arrives at a cluster, all of the LP's that have processed events that are greater than the timestamp of the straggler will be rolled back. This rollback scheme is called *clustered rollback*. The alternative to clustered rollback is *local rollback*. In a local rollback scheme the straggler event would be inserted into the receiver LP's input queue and the LP will roll back when it is detected. Although clustered rollbacks may cause some LP's to be rolled back unnecessarily leading to slower compuation, the approach was chosen for CTW because processed events will not have to be saved which requires less memory.

CTW uses a form of infrequent state savings with the timezone table used to determine the frequency. When an event is about to processed for an LP, the timezone of the last processed event is looked up and if event that is about to be processed is in a different timezone then the state is saved. This approach in which all LP's save their state every time an event is processed in a new timezone regardless of whether it receives an event from a remote cluster is called *local checkpointing*. This method reduces the state saving frequency more than a *clustered checkpointing* approach in which only the LP that receives an event from a remote cluster saves its state. The local checkpointing approach was chosen for CTW because a larger state saving frequency can increase rollback computation and it can even lead to more memory consumption because more events must be saved for coast forwarding during state restoration.

3.3 Rensselaer's Optimistic Simulation System (ROSS)

ROSS [8] is a general purpose simulator that is capable of running both conservatively and optimistically synchronized parallel simulations as well as sequential simulations. It is most often used for optimistic parallel simulations which is implemented with the time warp mechanism. ROSS started as a reimplementation of GTW and is still modeled after it but has had many enhancements. The same basic event scheduling mechanism is used but ROSS supports different priority queue implementations and different algorithms are used for fossil collection, state saving, and gvt calculation. In addition, ROSS uses processes instead of threads and uses message passing explicitly with MPI instead of shared memory for communication among the processes.

Just as in GTW, ROSS maps every LP to a process and each process contains its own pending event set structures. No locks are needed explicity within each process because there are no shared data structures among processes. The data structures are very similar to those used in GTW but have a different naming convention. The main data structures in ROSS are:

1. The *Event Queue* is analogous to the message queue in GTW. It contains the positive events for all

LP's in the corresponding process. In addition, an event queue is used to hold all remote events regardless of whether it is positive or negative. The event queue is implemented as a linked list.

2. The *Cancel Queue* is a linked list which is used to hold negative events for all LP's for the corresponding process. The cancel queue is used in the exact same way as GTW except that no locks are necessary.

3. The *Priority Queue* is analogous to the unprocessed event queue in GTW and contains events in timestamp order. ROSS also allows the priority queue to be implemented as a calendar queue, heap, splay tree, or avl tree depending on user configuration.

To reduce the time taken to fossil collect LPs, ROSS further divides the LPs in a single process into groups called Kernel Processes (KPs). All LPs in a KP are rolled back together and fossil collected together in a similar manner as a clustered rollback in CTW.

ROSS also uses reverse computation [9] to rollback LPs to a previous state instead of using the traditional copy state saving. This method allows simulations to run in a much smaller memory footprint because the previous states of LPs are not required to be saved.

3.4 WARPED

WARPED is the predecessor of WARPED2 and serves as the basis for the design and architecture of WARPED2. WARPED, unlike the previous time warp simulators, follows Jefferson's classic model with LPs having their own input, output, and state queues. It started as a completely processed based solution with only message passing for communication, but eventually, with the development of multicore processors, each process was extended into multiple threads to further enhance concurrent processing of events. Over the years, though, with many students developing new algorithms, the complexity of WARPED became unmaintainable. A great deal of indirection which, although made for a super configurable and modular design, became too complex for new students to learn it and enhance it. This has led to the development of WARPED2 which is detailed throughout the remained of this document.

3.5 Others

3.5.1 The ROme OpTimistic Simulator (ROOT-Sim)

ROOT-Sim is another general purpose Time Warp Simulator that uses message passing via MPI [10]. Like WARPED, ROOT-Sim is a more classic Time Warp implementation with each LP having their own input, state and output queues. What sets ROOT-Sim apart from other time warp simulators is the internal instrumentation tool, Dynamic Memory Logger and Restorer (DyMeLoR) that can optimize memory usage. DyMeLoR can determine whether the simulation models are better fit for copy-state saving or incremental state saving and transparently switch between them during runtime. Another service that ROOT-Sim offers is the Committed and Consistent Global Snapshot (CCGS). After each GVT calculation, ROOT-Sim transparently rebuilds a global snapshot of all LP states. Each LP can access its portion of the global snapshot on every GVT calculation. With this service, a simulation model can implement any custom global snapshot algorithm.

3.5.2 ROSS-MT

ROSS-MT [11] is a multi-threaded version of ROSS which is optimized to use shared memory to communicate among threads. The use of message passing with MPI was completely removed and all events are sent by direct insertion into the event queues. To reduce the added contention on the event queues, they are further divided by possible senders. ROSS-MT also optimizes the memory management so that it is more NUMA aware.

Chapter 4

The WARPED2 Simulation Kernel

This chapter describes the software architecture of the WARPED2 simulation kernel. The main compenents are described and the dependencies between each of them. The modeling API is also described and an example model is shown to illustrate the API.

4.1 The Software Architecture of WARPED2

The WARPED2 simulation kernel uses a modular design to make it more configurable, extendable, and maintainable. All components are configured and created at startup individually and subcomponents are accessed through pointers. Furthermore, WARPED2 is written in C++ so each component can also be an object of derived subclass that implement a well-defined interface by a base class. The main central component is the event dispatcher which is responsible for calling the a set of LP callback methods that are implemented in the simulation model during the course of the simulation. The event dispatcher can be configured for sequential simulation or a time warp simulation. The sequential simulation does not depend on any other components and contains only a single list of unprocessed events. The time warp event dispatcher on the other hand depends on multiple components that implement specialized algorithms for event scheduling, state saving, cancellation, GVT, termination, interprocess communication, and statistics. The main components for time warp and how they depend on each other are illustrated in figure 4.1.

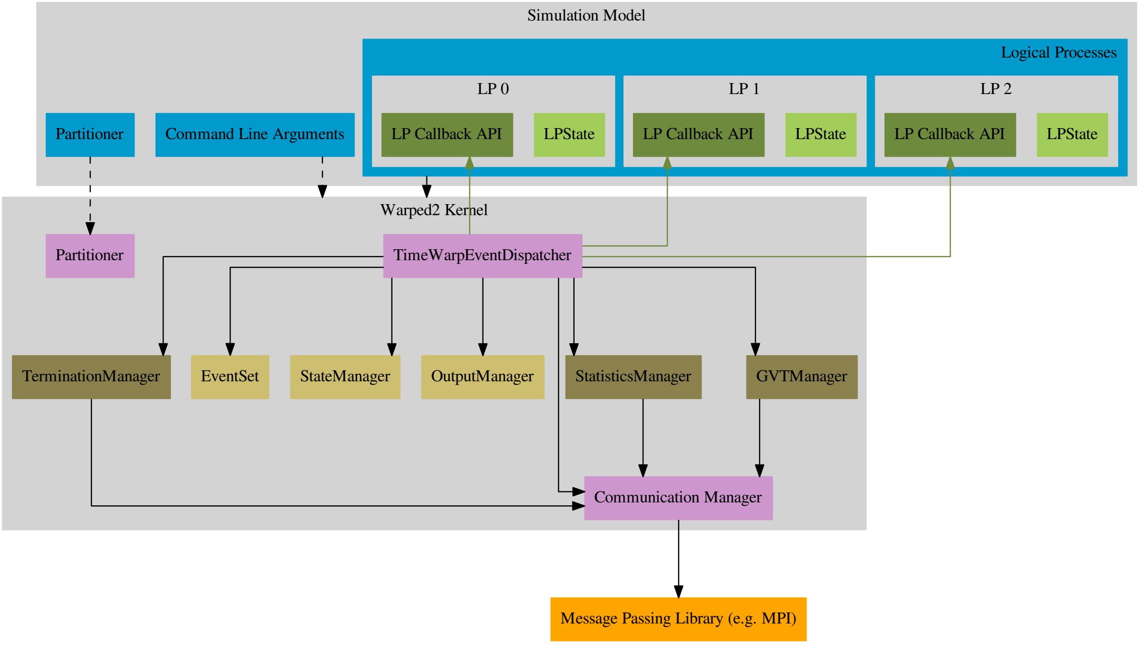


Figure 4.1: Time Warp Components in WARPED2

The time warp components can be categorized as either a local time warp component or a global time warp component. The local time warp components are used only for the local control mechanism of the individual LPs such as rolling back and fossil collecting whereas the global time warp components are concerned with the global control mechanisms such as GVT, termination detection and statistics counting. The global time warp components must be able to communicate with all other processes in the system so that it is possible to determine the global state of the system.

4.1.1 Local Time Warp Components

The three local time warp components are (1) the EventSet, (2) the OutputManager, and (3) the StateManager. The `OutputManager` and `StateManager` are abstract base classes that have a well defined API for cancellation and state saving techniques, respectively.

The EventSet contains the data structures for all of the unprocessed and processed events for the LPs that are local to the process. This includes the input queues for each LP and also the LTSF queues. The event set provides methods for obtaining the next event, processing a rollback, and fossil collecting processed events. All the data structures that have pending events must necessarily be thread safe since multiple worker threads could access them.

The OutputManager contains all of the output queues and implements a single cancellation technique. The derived class must implement methods for adding an event to an output queue, processing a rollback, and fossil collecting old output events. Currently, WARPED2 only implements aggressive cancellation through the `AggressiveOutputManager` subclass but may support more cancellation techniques in the future such as lazy cancellation.

The StateManager contains all of the state queues and implements a single technique for state saving and state restoration. The derived class must implement methods for saving the state of an LP, restoring the state of an LP, and fossil collecting old states. Currently, WARPED2 only implements periodic state saving through the `PeriodicStateSaving` subclass.

4.1.2 Global Time Warp Components

The three time warp components are (1) the GVTManager, (2) the TerminationManager, and (3) the StatisticsManager . The `GVTManager` is an abstract base class which defines a well defined API for a GVT algorithm but the `TerminationManager` and `StatisticsManager` are concrete classes which implement a termination detection algorithm and a central statistics manager, respectively.

The GVTmanager keeps track of the GVT and implements a specific Global Virtual Time (GVT) algorithm. Due to the hybrid communication design of WARPED2, the GVT algorithms will have some contribution from the worker threads and must also use the `CommunicationManager` for a message passing algorithm to use between the set of processes. Currently, WARPED2 implements an `AsynchronousGVTManager` and a `SynchronousGVTManager`. The algorithms implemented within each one will be detailed further in chapter 8.

The TerminationManager impements an algorithm for determining when all processes become inactive and initiates termination when that occurs. Just like the `GVTManager`, some contribution must come from the worker threads within each process and each process must communicate through the `CommunicationManager`. A more detailed description of the termination detection algorithm used in WARPED2 is given in chapter 8.

The StatisticsManager keeps track of all local statistics and provides methods for performing global reductions on the them.

4.1.3 CommunicationManager

The `CommunicationManager` provides an interface between the underlying message passing library and the global time warp components in the WARPED2 simulation kernel. Any interprocess communication must go through the `CommunicationManager`, including remote events that must be sent to another process or received from another process. Any class that has to do interprocess communication must register message types and a recieve callback function for each type with the `CommunicationManager`. The `WARPED_REGISTER_MSG_HANDLER` macro can be used to register the message types and recieve callbacks as long as the communicating class contains a data member that is a pointer the `CommunicationManager` and is named `comm_manager_`.

The message type must also be derived from the `TimeWarpKernelMessage` base class and must be serializable which means that all member variables must be registered with the serialization API so that a storage order can be defined. To do this, the `WARPED_REGISTER_SERIALIZABLE_MEMBERS` macro is provided. All member variable must be passed to this macro as well as `cereal::base_class<TimeWarpKernelMessage>(this)` to ensure that all members that are inherited from `TimeWarpKernelMessage` are also serialized. The order that the members are listed is completely arbitrary and does not matter. In addition, the derived message type must be registered using the `WARPED_REGISTER_POLYMORPHIC_SERIALIZABLE_CLASS` macro.

Currently, WARPED2 only implements an MPI based `CommunicationManager` through the `MPICommunicationManager` subclass.

4.1.4 Partitioner

The Partitioner is an abstract base class for implementating a method of logical process partitioning. Given a vector of LPs and some number of partitions, N, the Partitioner returns N vectors of LPs by using some partitioning technique. The WARPED2 kernel implements a RoundRobin partitioner and a ProfileGuidedPartitioner but also allows simulation models to define their own custom partitioners. More details of these partitioning techniques are given in chapter 6.

4.2 The Modeling API of WARPED2

The modeling interface of warped2 is a set of abstract base classes that contain methods to be implemented in a derived class in the simulation model. The three main base class types that must be implemented are LogicalProcess, LPState, and Event. Optionally, the user may create a custom partitioner from the Partitoner base class. In the remainder of this section, each class is described in more detail and sample implementations are shown for each.

4.2.1 The LPState Structure

The state of the LPs must be defined with the WARPED_DEFINE_LP_STATE_STRUCT macro which automatically derives from an intermediate structure that defines a necessary clone method so that the model developer does not have to. It is needed to ensure that the WARPED2 kernel can save a copy of the state and restore the state from a pointer to the LogicalProcess base class. A simple example of a LP state that contains just message counts is shown below in listing 4.1.

If the state contains complex data structures that contain pointers then the default copy constructor and default copy assignment operator will only perform shallow copies. In this case the user must implement a custom copy constructor or a custom copy assignment operator or both. The copy constructor will define

```
WARPED_DEFINE_LP_STATE_STRUCT(ExampleState) {
   unsigned int messages_received_;
};
```

Listing 4.1: Example WARPED2 State Definition

```
class ExampleEvent : public warped::Event {
public:

   const std::string& receiverName() const { return receiver_name_; }
   unsigned int timestamp() const { return time_stamp_; }

   std::string receiver_name_;
   unsigned int timestamp_;

   WARPED_REGISTER_SERIALIZABLE_MEMBERS(cereal::base_class<warped::Event>(this
      ),
                                   receiver_name_, timestamp_)
};
WARPED_REGISTER_POLYMORPHIC_SERIALIZABLE_CLASS(ExampleEvent)
```

Listing 4.2: Example WARPED2 Event Definition

the behavior for saving the state whereas the copy assignent operator will define the behavior for restoring the state. Note that the copy assignment operator will most likely not be needed since a shallow copy will usually suffice for restoring the state.

4.2.2 The Event Class

The `Event` base class is used as the basis for creating model specific events. The user must implement at least two methods: (1) `receiverName()` and (2) `timestamp()` . It is necessary so that the name of the receiver and receive time, respectively, can be obtained for each instance of an event within the kernel. The user must also that events are serializable so that they can be sent and received through the message passing system. A basic example of an event implementation is shown below in listing 4.2.

4.2.3 The LogicalProcess class

The most important class definition in the simulation model is the `LogicalProcess` class. The implementation of the `LogicalProcess` class defines the callback functions that the event dispatcher with the WARPED2 kernel calls and thus it defines the behavior of the simulation. The user must include a single `LP-State` as a data member of the `LogicalProcess` and provide three callback method implementations:

1. The `initializeLP` method is called to perform any initializations of an LP that must be done prior to the start of the simulation and must return a set of initial events.

```
class ExampleLP : public warped::LogicalProcess {
public:

   warped::LPState& getState() { return this->state_; }

   std::vector<std::shared_ptr<warped::Event> > initializeLP() override {
      this->registerRNG(this->rng_);
      std::vector<std::shared_ptr<warped::Event> > events;
      ...
      return events;
   }

   std::vector<std::shared_ptr<warped::Event>> receiveEvent(const warped::
       Event& event) {
      ++this->state_.messages_received_;
      std::vector<std::shared_ptr<warped::Event> > response_events;
      ...
      return response_events;
   }

   ExampleState state_;
};
```

Listing 4.3: Example WARPED2 LogicalProcess Definition

2. The `receiveEvent` method is called to perform the forward computation based on the event that is passed. The implementation of this method should process the event by updating the state of the LP and returning a set of new events with future timestamps.

3. The `getState` method provides a way for the WARPED2 kernel to get the current state of the LP so that it can be saved.

It is necessary that at least one LP has an initial event that is returned by initializeLP, otherwise no events can be received and simulation will terminate immediately. Also note that it will be called once for *every* LP instance so it is possible that initial events are returned only in some cases. An example of a `LogicalProcess` implementation is shown below in listing 4.3.

4.2.4 The Partitioner class

The WARPED2 kernel already provides a set of partitioners but the model developer can define their own partitioner that is customized for the model. The user must derive from the Partitioner base class and implement just a single method which takes a vector of all LPs and the number of partitions desired and

```
class ExamplePartitioner : public Partitioner {
   std::vector<std::vector<LogicalProcess*>>
   partition(const std::vector<LogicalProcess*>& lps, const unsigned int
      num_partitions) {
      std::vector<std::vector<LogicalProcess*>> parts;
      ...
      return parts;
   }
};
```

Listing 4.4: Example WARPED2 Partitioner Definition

returns a vector of vectors of LP's. In general, the partitioner should work for any number of partitions and not impose any constraints because the partition method is called back in a general way from the kernel. A simple template of a partitioner is shown in listing 4.4.

4.2.5 Random Number Generation

If the simulation model must use random number generators, then they must all be registered with the WARPED2 kernel so that the state of the random number generator can be saved and restored in case of rollbacks and provide deterministic result. The random number generators can be any type as long as they implement the << `operator` and >> `operator` to allow the kernel to save and restore the internal state of the random number generator. The random number generators in the C++11 standard libraries [12] all fit this requirement and can all be used so it is not necessary to always build a custom random number generator. To register the random number generator, the `registerRNG` template function must be used which is a member of the `LogicalProcess` class. All LP's must have separate random number generators and must be registered in the initializeLP callback function as shown in listing 4.3.

4.2.6 Command Line Arguments and the Kernel Entry Point

Once all the necessary structures and classes have been defined, the model's main function must be implementd which is where all calls into the kernel are made. First, the model can create specific command line arguments but they must be registered with the kernel. This must be done first so that it can be passed to the constructor of a `Simulation` instance and so that the command line arguments can be displayed without unnecessarily running a simulation. The kernel uses a third-party library called TCLAP for command line arguments. After setting up the command line arguments, all of the LP objects and optionally a partitioner object must be instantiated and passed to the kernel through the `simulate` method of `Simulation` object. Two versions of the simulate method is available, one for a model with a custom partitioner and one without as listed below:

1. `void simulate(const std::vector<LogicalProcess*>& lps);`

2.
   ```
   void simulate(const std::vector<LogicalProcess*>& lps,
                                 std::unique_ptr<Partitioner> partitioner);
   ```

A sample implementation of a models main function is shown in listing 4.5.

```
int main(int argc, const char **argv) {
   unsigned int num_lps = 10000;

   TCLAP::ValueArg<unsigned int> num_lps_arg("o", "lp-count", "Number␣of␣lp’s"
      , false,
                                          num_lps, "unsigned␣int");
   std::vector<TCLAP::Arg*> cmd_line_args = { &num_lps_arg };
   warped::Simulation simulation {"Example␣Simulation", argc, argv,
      cmd_line_args};

   num_lps = num_lps_arg.getValue();
   std::vector<ExampleLP> lps;
   for (unsigned int i = 0; i < num_lps; i++) {
      std::string name = std::string("LP_") + std::to_string(i);
      lps.emplace_back(name, 1, i);
   }

   std::vector<warped::LogicalProcess*> lp_pointers;
   for (auto& lp : lps) {
      lp_pointers.push_back(&lp);
   }
   simulation.simulate(lp_pointers);

   return 0;
}
```

Listing 4.5: Exmple WARPED2 Main Definition

Chapter 5

Plans of Study

Experimental results will be presented throughout the remaining chapters as the various aspects of WARPED2 are discussed. All chapters will contain experimental results for only x86 SMP nodes and clusters and will be targeted towards understanding and determining the best design options and configurations.

The single node SMP experiments will be run on an Intel® Xeon® X5675 processor. The intel® Xeon ® X5675 has 6 cores with hyperthreading so that 12 simultaneous threads are supported. Cluster experiments will be run on Intel® Xeon® E5410. The Intel® Xeon® E5410 is a quad core processor but each machine contains two sockets so that 8 cores are available. A summary of the architectural features and runtime systems for both machines are summarized in table A.1 in appendix A.

Throughout the rest of this text, the *efficieny* and *event rate* will be two of the main measures used to analyze behavior. Other measures will be introduced as needed. The efficiency is defined as:

$$Efficiency = events_{committed}/events_{processed} * 100$$

where $events_{committed}$ is the number of events that are processed and not rolled back and $events_{processed}$ is the total number of events that are processed whether they are rolled back or not. Thus, the efficiency is a measure of the percentage of events that are processed and not rolled back. The event rate is defined as:

$$EventRate = events_{committed}/runtime$$

5.1 Experiments

The experiments carried out on a single node SMP machine will be strongly tied to thread synchronization, state saving, memory allocation, and fossil collection. On the other hand, experiments carried out on a cluster will be strongly tied to communication, partitioning, state saving, and scaling up simulations. The experiments for a single SMP node are summarized in table 5.1 and for a cluster in table 5.2.

Comparison	Measure(s)	Figure(s)	Machine(s)
Number of LTSF Queues	Event Rate and Efficiency	Figure 7.2	X5675
LTSF Queue Partitioning	Time and Efficiency	Figures 7.4, 7.3	X5675, E5410
State Saving Period	Event Rate, ...	Figure 7.5	X5675
Spinlocks	Speedup	Figure 7.7	X5675
Fossil Collection Techniques	Event Rate	Figure 9.1	X5675
Memory Allocators	Event Rate	Figure 9.2	X5675
Memory Allocators	Efficiency	Figure 9.3	X5675
LTSF Queues	% Memory Increase	Figure 9.4	X5675
State Saving Period	% Memory Decrease	Figure 9.5	X5675
GVT Period	Memory and Time	Figure 9.6	X5675

Table 5.1: SMP Node Experiments

Comparison	Measure(s)	Figure(s)	Machine(s)
Communication Model	Event Rate and Efficiency	Figure 6.1	E5410
Communication Model	% Remote Events	Figure 6.2	E5410
Interprocess Partitioning	Time, Efficiency, % Remote Events	Figure 6.4	E5410
Aggregate Messages	Time, Rollbacks, Remote Events	Figures 6.6, 6.7	E5410
Scaling LPs	Efficiency, % Remote Events, Event Rate	Figure 6.8	E5410
State Saving Period	Event Rate, ...	Figure 7.6	E5410
State Saving Period	% Memory Decrease	Figure 9.5	E5410

Table 5.2: Cluster Experiments

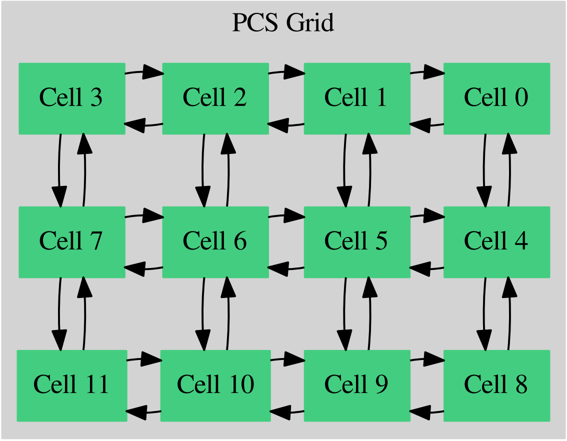


Figure 5.1: PCS Model Logical Processes

5.2 Simulation Models

This section describes some of the WARPED2 simulation models that will be used for experiments. The model configurations that are used are included within tables in appendix B.

5.2.1 PCS

The model describes a type of wireless communication network known as a Portable Cellular Service (PCS) network. A PCS network provides services for a number of *portables* which are the subscribers to the network. The service area of the network is divided into *cells* and a single *port* covers a single cell which has a certain number channels that it can allocate for calls. When an incoming or outgoing call is made, a port must allocate a channel from the port and if one cannot be allocated then the call is *blocked* [13]. The only type of logical processes in the PCS simulation are the cells. The cells in the WARPED2 model form a rectangular grid as shown in figure 5.1. Portables can only move to other cells from an adjacent cell with a wrap around occuring at the edges.

The portables stay within a cell for a period of time which follows a poisson distribution before moving to another cell. If the all cells are busy in the cell that the portable is moving to then the call is dropped. This is called a *handoff block*.

Every LP keeps track of the following state variables:

- Number of idle channels
- Number of call attempts
- Number of channel blocks
- Number of handoff blocks

Call arrivals to a portable also follow a poisson distribution. The cells in the model generate all of their own incoming calls in a self-initiating process. Four types of events are used to model the network: (1) NextCall, (2) CallCompletion, (3) PortableMoveOut, and (4) PortableMoveIn . The NextCall, CallCompletion, and PortableMoveOut events are all sent to self whereas the PortableMoveIn event is sent to an adjacent cell based on a random variable with uniform probability.

5.2.2 Traffic

The traffic model describes a grid of intersections and the movement of cars through the intersections and between the intersections. All intersections are four way intersections and have three lanes in each direction. The LPs in the traffic model are the intersections and form a rectangular grid in the same way as that of the PCS model.

The simulation starts with a uniform distribution of cars at each intersection and each car is a assigned a destination which it finishes at. With each arrival at an intersection the car always goes in the direction that gets it closer to its destination. The number of cars going out of an intersection in the same direction is limited to a maximum number due to traffic on the destination road. An incoming car that cannot travel in a specific direction is forced back in the same direction it came from and tries another route. For each intersection, the state consists of:

- Number of cars coming into the intersection from each direction
- Number of cars going out of the intersection to each direction

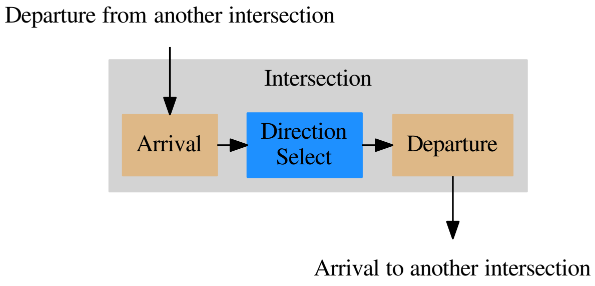

Departure from another intersection
Intersection
Arrival
Direction Select
Departure
Arrival to another intersection

Figure 5.2: Intersection Event Sequence

- Total cars arrived at the intersection
- Total cars finished at the intersection

A car goes through three event phase in every intersection as shown in figure 5.2. The three types of events are: (1) Arrival, (2) Direction Select, and (3) Departure . The arrival and departure events are mainly used for simple state updates and timing of the next events. The direction select event is where the complexity of the simulation is and determines the direction that the car should go in order to reach its destination or whether the car should take another route due to traffic.

All events in the traffic model follow an exponential distribution and a single event is generated with the processing of each event. The departure and direction select events are self-generated but the arrival event is from an adjacent intersection.

5.2.3 Epidemic

The epidemic model describes the spreading of an infectious disease across a set of geographic locations in a region and across a set of regions. The logical processes in the simulation are geographic locations which represent a portion of the population. The interactions between people in different geographic locations and regions are modeled with a diffusion network. An abstract view of the simulation model is shown below in figure 5.3.

The model is based on reaction-diffusion model [14]. A reaction process models the behavior of the

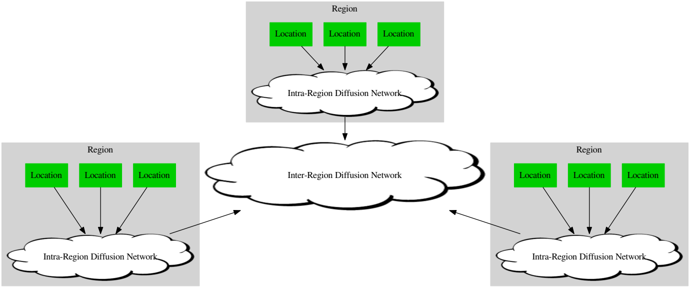

Region
Location
Location
Location
Intra-Region Diffusion Network
Region
Location
Location
Location
Intra-Region Diffusion Network
Inter-Region Diffusion Network
Region
Location
Location
Location
Intra-Region Diffusion Network

Figure 5.3: Epidemic Model

disease within a person as well as the transmission of the disease between people within the same location. A probabilistic reaction function defines the behavior of the disease between individuals [15]. The disease within an individual is modeled by a Probabilistic Timed Transition System (PTTS) [15] which is a finite state machine describing states of the disease and the transitions between states. Together, the reaction function and PTTS models of the disease form the *disease model*. The diffusion network models the social interactions of people between locations and between regions [15].

5.2.4 Airport

The airport model describes the departure and arrivals of airplanes between a connected set of airports. The logical processes represent airports and the events are simply departures and arrivals. For simplicity, the airports are connected in a rectangular grid and can only fly to aiports to the north, east, south, or west of the aiport departed from. The time spent between takeoff and landing and between landing and takeoff both follow an exponential distributions.

Chapter 6

Interprocess Communication

This chapter first provides an overview of different hybrid communication models that can be used with MPI and compares the performances of them. Then various partitioning schemes that are implemented in WARPED2 are discussed and analyzed. Lastly, the message aggregation technique that is used is described and the behavior with various aggregate message counts are explored.

6.1 Multithreaded Message Passing with MPI

In a hybrid communication model, the choice of which threads and how many threads should be involved in message passing communication is an important design decision that can significantly impact performance and stability of the simulator. The MPI standard specifies four level of thread support that each implementation may support:

MPI_THREAD_SINGLE: The application uses only a single thread.

MPI_THREAD_FUNNELED: The application is multithreaded but only a single thread will make MPI calls.

MPI_THREAD_SERIALIZED: The application is multithreaded but only a single thread at a time will make MPI calls.

MPI_THREAD_MULTIPLE: The application is multithreaded and threads have no restrictions.

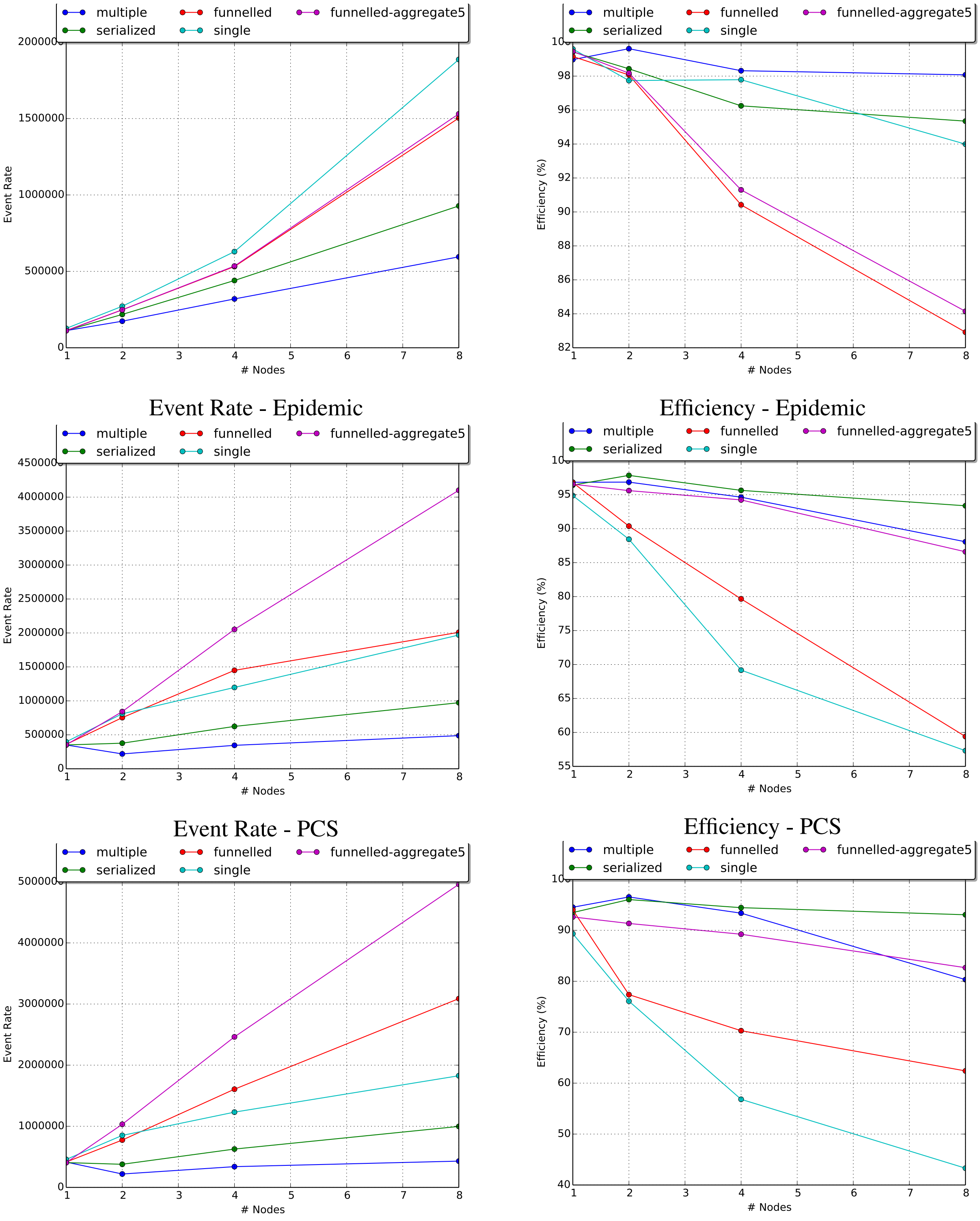
Funnelling the communication to the manager thread can increase latency because an extra step for temporarily queueing the message is necessary. Furthermore, with too fine of a granularity, the number of messages can pile up and create a bottleneck in the communication and cause instability. Serialized or full multiple thread communication, however, can increase contention for access to MPI library calls and slow down the worker threads and also forces worker threads to take more time for serialization, deserialization, and memory allocation/deallocation.

The plots in figure 6.1 summarize the performance and efficiency of single, funnelled, serialized, and multiple threaded communication in WARPED2 for the Epidemic, PCS, and Traffic models. Funnelled communication with aggregated messages is also shown.

For serialized communication, a single global spinlock is used to protect access to all MPI library calls. For all multithreaded communication models, 7 worker threads and a manager thread are used per node and for single threaded communication, 8 processes per node are used. Aggregated messages contain 5 ordinary messages which could be positive events, anti-messages, or other control tokens for GVT and termination detection.

For the traffic and PCS simulation models, aggregated funnelled messages is a clear winner. It is also apparent that when using funnelled communication, aggregating messages makes a huge impact on increasing the efficiency. The epidemic model, on the other hand, works the best with a single threaded communication model. Both of these observations can be explained by looking at the percentage of events that are sent to a remote MPI process in figure 6.2.

Only a small fraction of a percent of events are remote for epidemic but for PCS and traffic, the percentage of remote events is much larger. Having a much smaller percentage of remote events means that there will be very little synchronization overhead between threads for communication and funnelled communication will only increase the communication latency due to the queueing delay, and thus increase rollbacks. As remote communication increases, however, the synchronization costs of worker threads increases and funnelling all communication to the manager thread can help to reduce synchronization and increase performance but the queueing delay and rollbacks will both increase. Aggregating messages with funnelled communication can decrease the rollbacks even though the queueing delay is still increased. Not sure why this is.



Event Rate - Epidemic

Efficiency - Epidemic

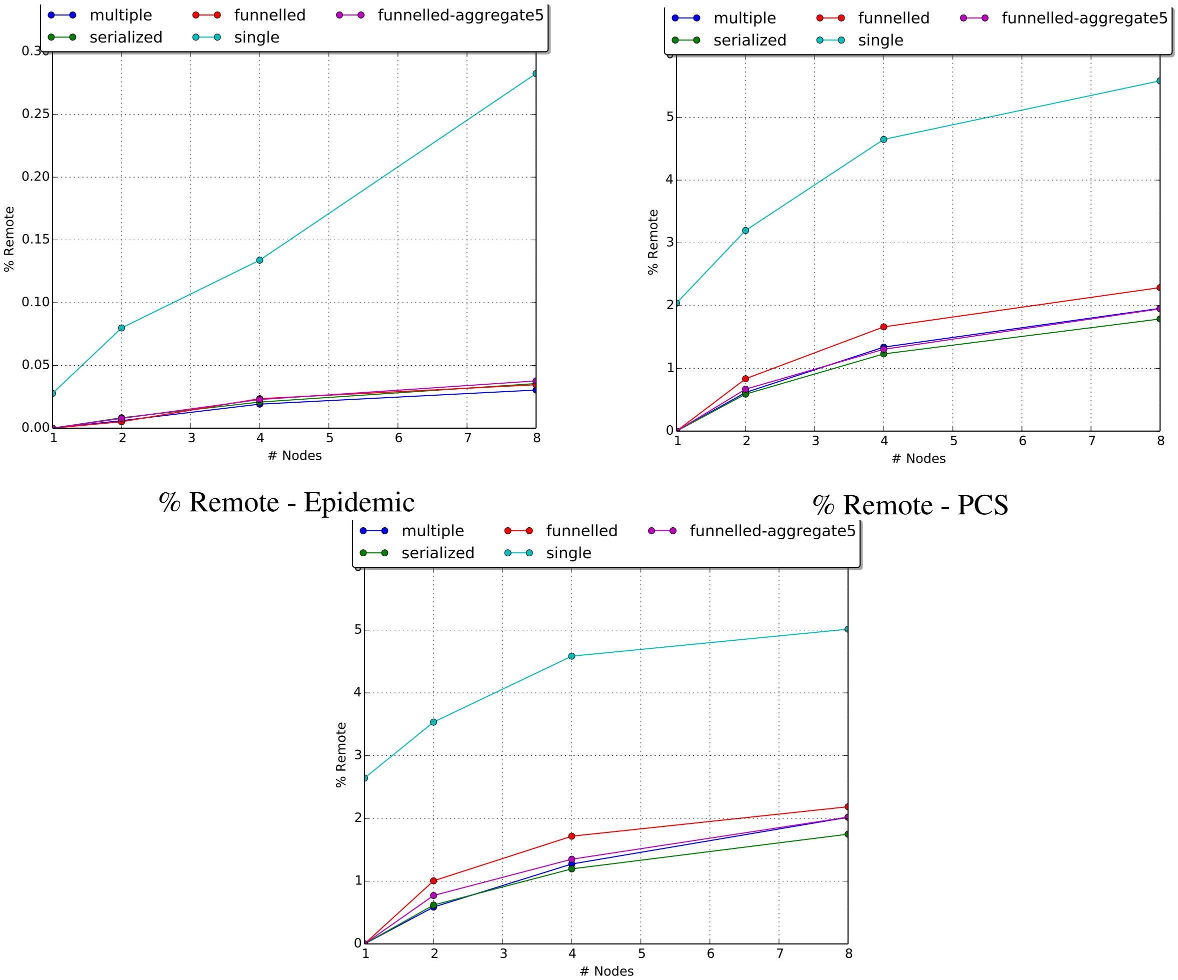
Event Rate - PCS

Efficiency - PCS

Event Rate - Traffic

Efficiency - Traffic

Figure 6.1: Communication Models: Speedup and Efficiency



% Remote - Epidemic

% Remote - PCS

% Remote - Traffic

Figure 6.2: Communication Models: % Remote Events

6.2 Communication Tuning Parameters

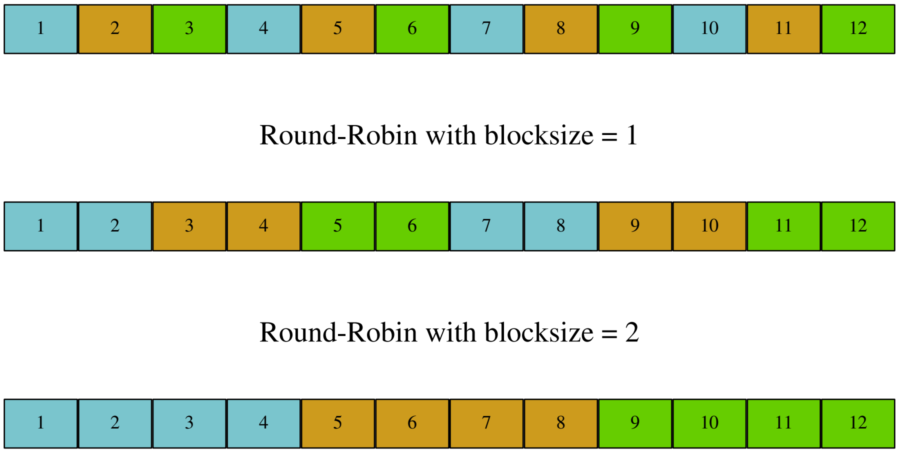
To effectively scale simulations to larger number of nodes in a cluster, the interprocess communication should be minimized as much as possible to avoid synchronization and communication costs and rollbacks. To allow different types of models to run on larger scale cluster, WARPED2 allows the user to choose how to partition LPs among the MPI processes and how many messages to aggregate for each receiver process. The next two sections will provide any overview of supported partitioning schemes and message aggregation in WARPED2.

6.2.1 Interprocess Partitioning

Partitioning the LPs among processes is necessary so that LPs that communicate with each other a lot remain on the same node and minimize network communication in clusters. When network communication is minimized, the latency of sending events is decreased and will help to keep the number of rollbacks down. Currently, WARPED2 supports three partitioning schemes: (1) Round-Robin, (2) Profile-Guided, and (3) Custom .

Round-Robin partitioning is a simple partitioning scheme where $blocksize$ number of LPs are allocated to each process at a time and in order of increasing process id. $blocksize$ can be between 1 and $NumLPs/NumProcesses$ where the latter is simply known as block partitioning. From now on, a $blocksize$ of 1 will also be called simply round-robin. A block size in between the extreme values can be used in some models to make a second level partitioning scheme easier. Figure 6.3 illustrates how a round-robin partitioning scheme would work with different values for $blocksize$, where different partitions are represented by different colors and the numbers represent an LP id.

Profile-Guided partitioning is based on profile data that is collected with a sequential simulation. The profile data contains a count of events for each communicating pair of LPs. When a time warp simulation is run, the partitioner uses that data to build a weighted graph and uses a graph partitioning tool to create partitions that minimize the cut-set so that cross-partition communication is minimized. Profile-Guided partitioning should be used if a simulation model has a peculiar communication pattern among LPs that is



Round-Robin with blocksize = 1

Round-Robin with blocksize = 2

Round-Robin with blocksize = $NumLPs/NumProcesses$ (Block)

Figure 6.3: Round Robin Partitioning

not suitable for round-robin and does not implement a custom partitioning scheme. It will also work well for any odd number MPI processes which may not work well with other schemes.

The plots in figure 6.4 show a comparison of the round-robin, block, and profile-guided partitioning schemes. In general, more performance is closely tied with a lower percentage of remote events and a higher efficiency. The biggest exception is with the airport model, which has a very low efficiency with the profile-guided partitioning. This could just be a result of the model size which is only 2500 LPs making each node individually less efficient.

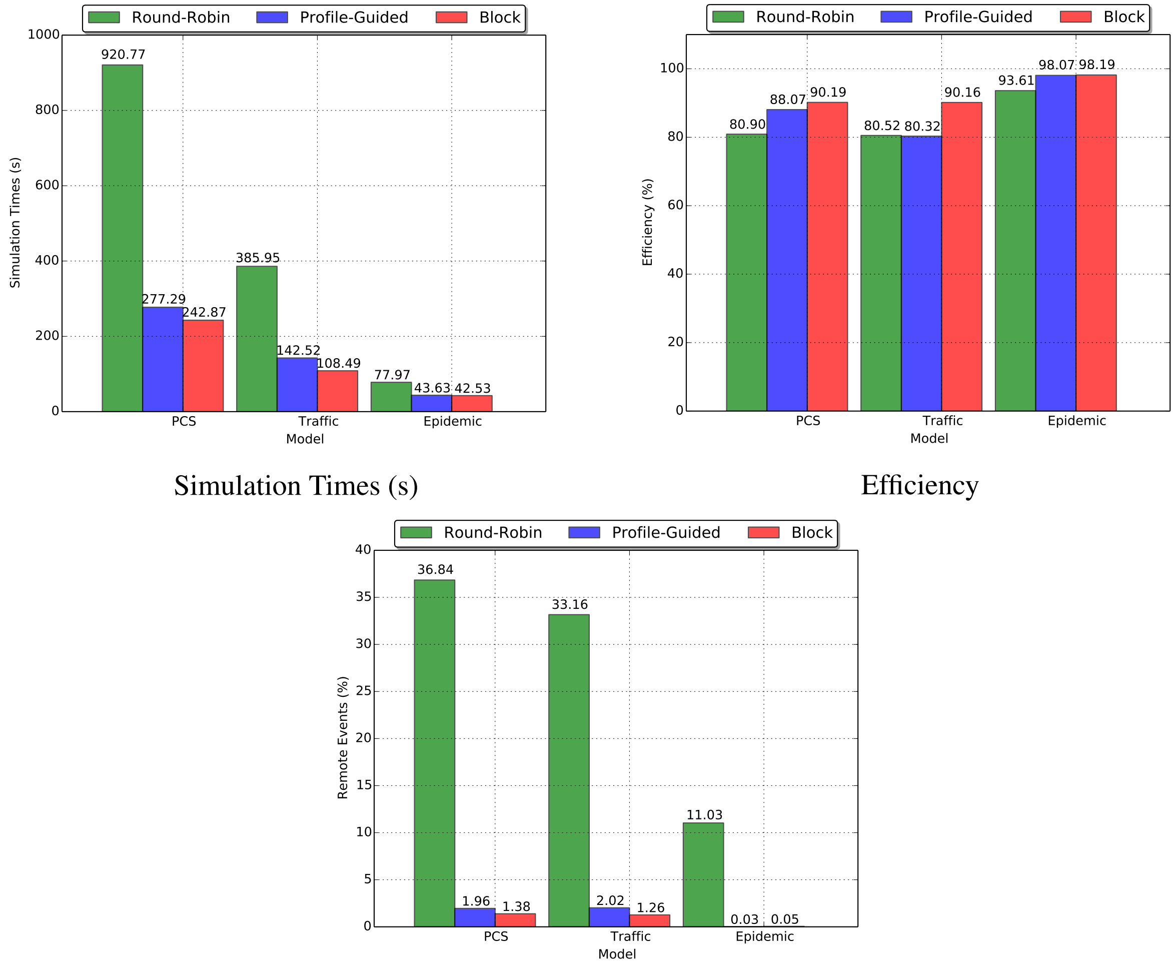
6.2.2 Aggregate Message Count

In general, MPI implementations are usually designed to efficiently use networking bandwidth and hence are better for coarse-grained parallelism with large message sizes. Parallel discrete event simulation is inherently fine-grained and the events are usually pretty small which makes it less suitable for MPI. To coarsen the granularity and make use the available network bandwidth, messages that are sent to the same destination process can be aggregated at the sender and split at the receiver. As seen previously, aggregating messages can also increases the efficiency of simulations.

The idividual messages are aggregated into a single aggregate message until the aggregate message contains N idividual messages or until a control message is included. A control message could be a GVT token, a termination token, a GVT update message, or termination status update message.

The combined message is started with a header which includes the number of individual messages and the lengths of each individual message. The locacation of each individual message can be found by adding the length of the header with the length of the previous message and the length of the header is easily determined by knowing how many individual message there are. Figure 6.5 illustrates the format of the aggregated message with the sizes of each field.

Size of the aggregated messages in bytes are:



Simulation Times (s)

Efficiency

% Remote Events

Figure 6.4: Interprocess Partitioning: 8 Node Cluster

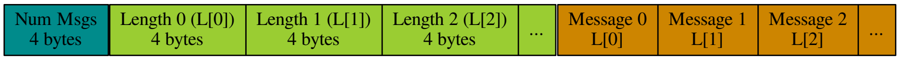


Figure 6.5: Aggregate Message Packet Format

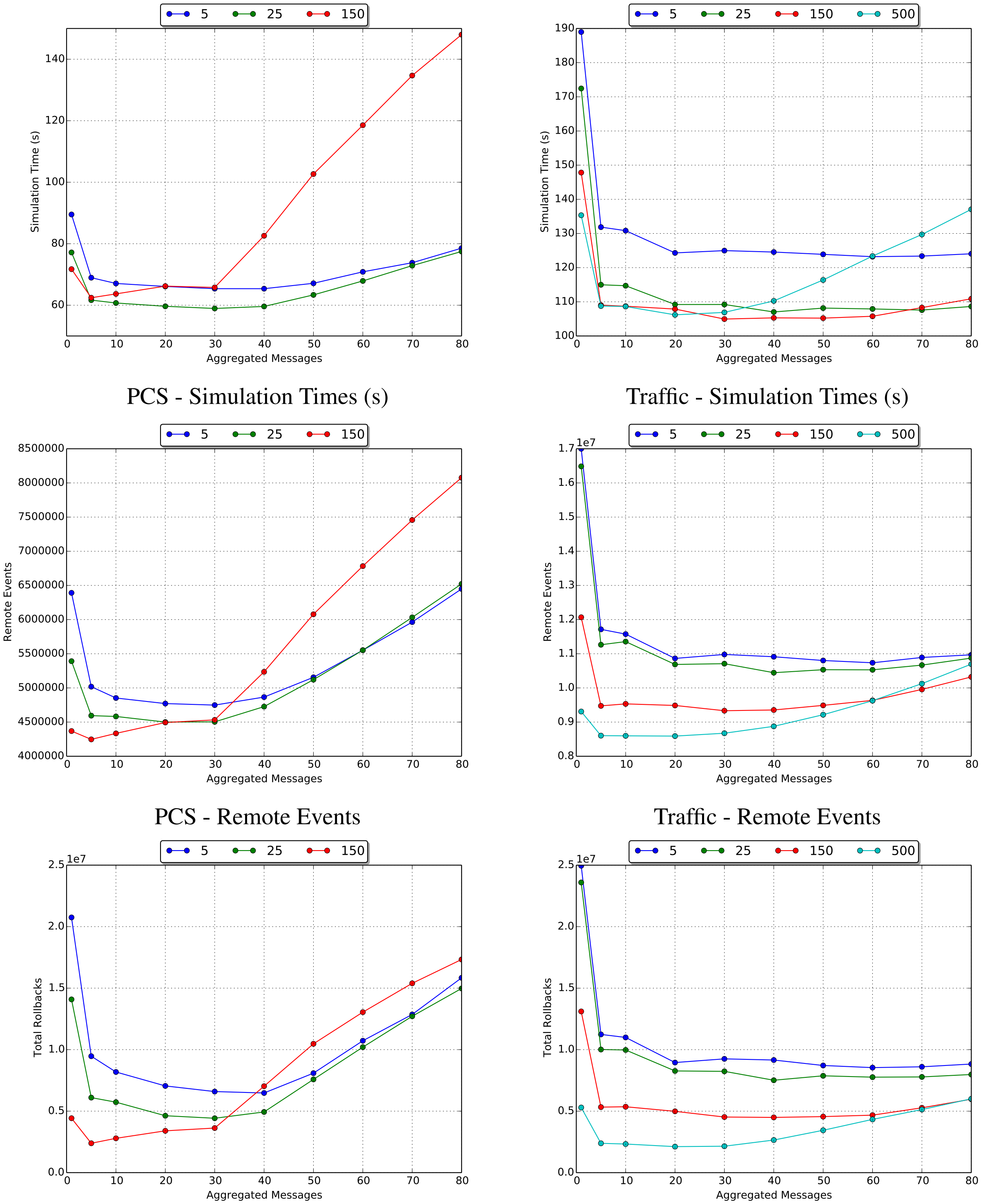
$$4 * (N + 1) + \sum_{n=0}^{N-1} length[n]$$

A message count approach was chosen over size base approach in WARPED2 since the size of the messages would not be known until they are serialized and a seperate temporary buffer would be needed for each individual messaage before copying them into a new aggregate buffer. Using a header rather than a token seperator within the aggregate message is also necessary since a binary serialization technique is used making it impossible to deserialize the aggregate messages when they are received.

The plots in figures 6.6 show how various message aggregate sizes change performance, rollback, and remote events for the PCS and Traffic models. The results are also plotted for different state saving periods since different state saving period has a significant impact on the number of rollbacks and the number of remote events.

The results for PCS and Traffic indicate that just a small number of aggregated messages has a durastic effect on increasing performance, and decreasing remote events and rollbacks. With a larger state saving period, the performance drops off much faster with an increasing number of aggregated messages than with a small state saving period. This is probably due a decrease in the forward computation time which would make the increased latency from message aggregation much more intolerable.

Not all models benefit from message aggregation, however. Figure 6.7 illustrates this for the epidemic model which has very little interprocess communication. With such little remote communication due to excellent partitioning capabilities, any amount of message aggregation only hurts performance by increasing communication latency.



PCS - Simulation Times (s)

Traffic - Simulation Times (s)

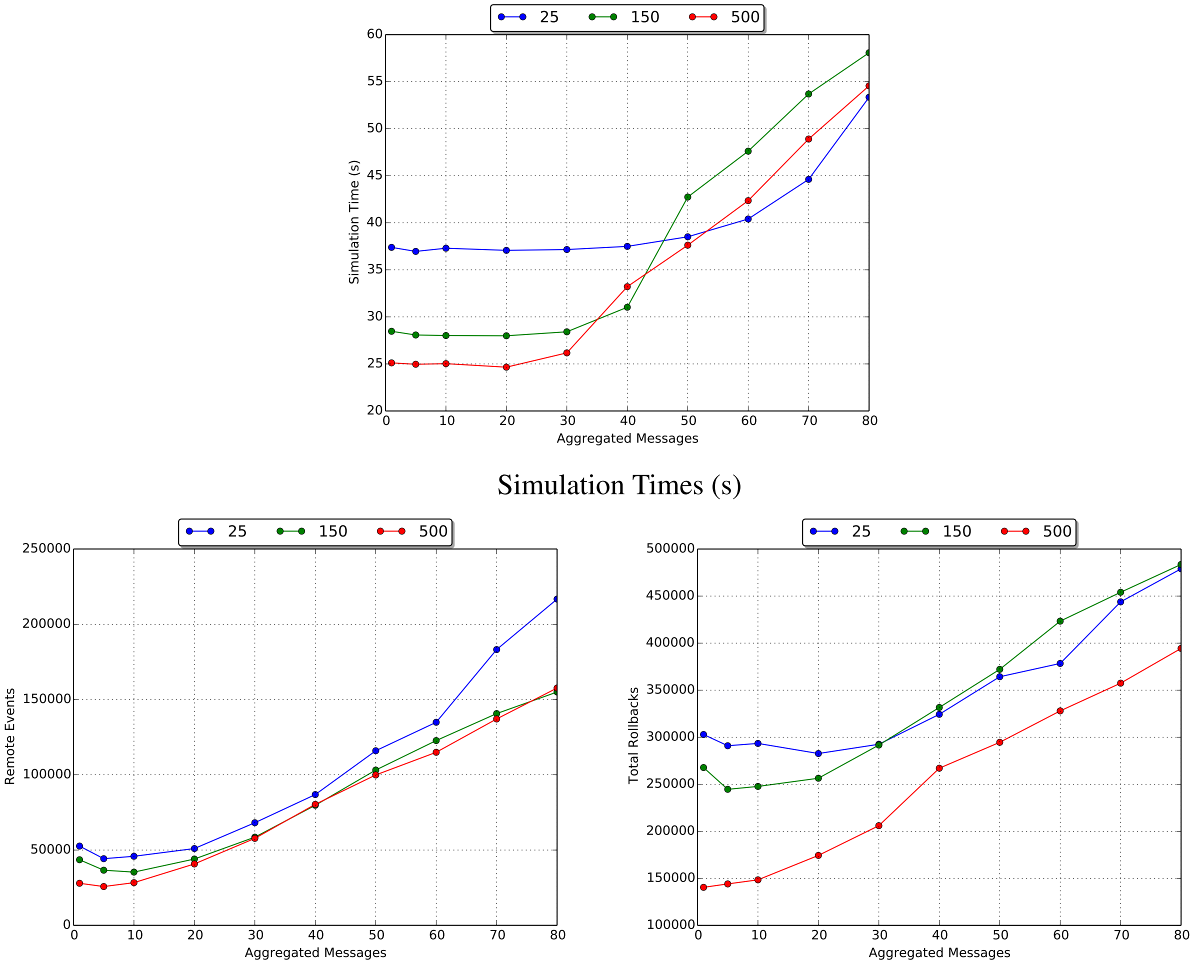
PCS - Remote Events

Traffic - Remote Events

PCS - Rollbacks

Rollbacks

Figure 6.6: Aggregate Messages: PCS and Traffic Models



Simulation Times (s)

Remote Events

Rollbacks

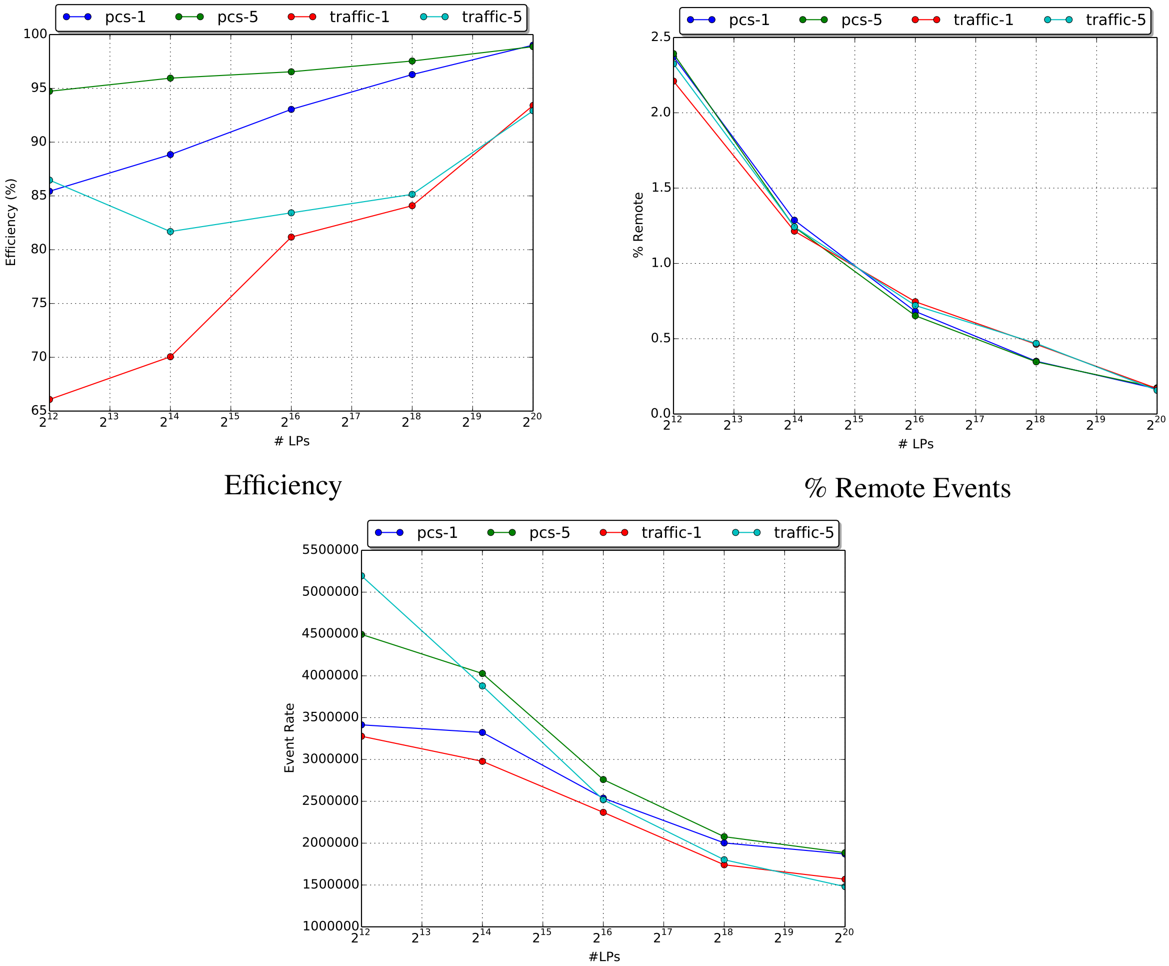
Figure 6.7: Aggregate Messages: Epidemic Model

6.3 Scaling up the Simulation Model

All previous simulation results have been from simulation models that were configured for 10,000 LPs. This section analyzes the behavior of the simulations when the number of LPs are scaled up to larger numbers on an 8 node cluster. Figure 6.8 shows how the traffic and PCS simulation model scale up with 5 messages aggregated and no message aggregation.

It appear, from these results, that as these models are scaled up, the benefit of aggregated messages decreases. That is, the efficiency and percent remote events for aggregate versus individual messages converge with more LPs.

In general, whether aggregate messages are used or not, scaling up the number of LP always has the



Efficiency

% Remote Events

Event Rate

Figure 6.8: Scaling #LPs on a Cluster

same effect. The efficiency increases significantly due to the work being spread out among more LPs. Statistically, the odds that two LPs will send events to the same LP and the receiving LP will process them out of order diminishes with a scaled simulation model. Also, each process processes events for a larger subset of the LPs. If the simulation model can be partitioned well in any way, scaling up will only make partitioning more effective since the cross-partition communication will remain small as the partition size grows. This is what ultimately reduces the reduction in the percentage of events that are remote. The event rate tends to decrease, however as the number of LPs increases. The reason for this is that each worker thread will have to use a larger percent its time to fossil collect LPs instead of processing events.

Chapter 7

Event Scheduling and Data Structures in WARPED2

7.1 Pending Event Set Data Structures

The pending event set is the set of events that are ready to be processed at any given time. Every process contains a logically seperate pending event set for a dedicated set of LPs. Events that are being sent to an LP that is local to the process are directly inserted into the pending event set. Events that must be sent to a remote process are inserted into a remote event send queue where the manager thread will remove it, form an event message and send the message to the recieving process. The manager thread at the receiving process will then unpack the message and insert it into the pending event set. The pending event set is made up of a queue for each LP known as an *unprocessed queue* which is directly accessible from all threads and allows for the sending of events.

The unprocessed queue contains both positive events and anti-messages and remains sorted at all times. The anti-messages are given priority over their positive counterparts so that the anti-messages are always processed first and any unnecessary rollbacks can be prevented which could create a chain of rollbacks that could cause instability [16]. To avoid any unnecessary copying from the simulation model to the kernel and with the kernel, the unprocessed queue only contains pointers to the unprocessed events which are passed in by the simulation model.

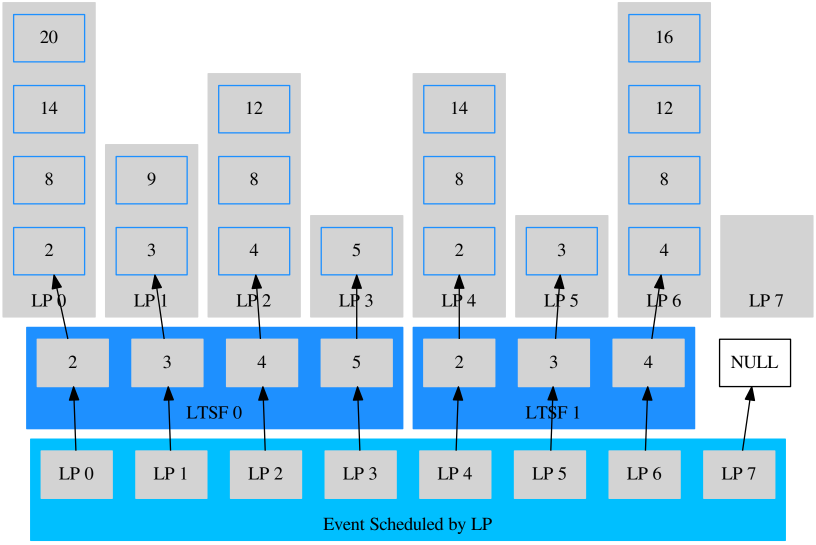


Figure 7.1: WARPED2 Pending Event Set Data Structures

A second type of data structure, an *LTSF (Lowest TimeStamp First) Queue*, provides order among events from multiple LPs. At most, a single event from each LP is *scheduled* into a single LTSF queue. Scheduling an event to an LTSF queue means only inserting a pointer to an event into the LTSF queue with no removal from the unprocessed queue. Every process has one or more LTSF queues which the worker threads obtain the lowest timestamped event to process.

A third type of data structure is also used to keep track of the events that have been scheduled or are currently being processed but organized by receiving LP. It is used for two main reasons. First, when inserting an event into the recieving LPs unprocessed queue, it provides a way to determine if a smaller event has already been scheduled for the LP. The new event can then be scheduled in place of the already scheduled event. Second, it serves to prevent multiple worker threads from processing events from the same LP which could cause out of order committing of events without rolling back as would be necessary. Figure 7.1 illustrates the pending event set data structures in WARPED2.

7.2 Processing Events

One or more of the worker threads are assigned to each LTSF Queue to process events from and they all follow the exact same procedure. First, an event is removed from an LTSF queue to be processed but remains in the unprocessed queue until it is completely processed or cancelled out. The event is first checked against the last processed event of the receiving LP to see if it is a straggler, and rolled back if necessary. An anti-message is also considered to be a straggler if its positive counterpart was the last processed event for the LP. If the event is an anti-message, the positive counterpart cannot be assumed to be in the LPs unprocessed queue. If the positive event is present then it is cancelled out but if it is not present then the anti-message remains in the unprocessed queue. In both cases, an attempt to reschedule a new event to the LTSF queue occurs and the worker thread continues to the next event. If the next event to be processed is positive then it is processed normally, the state of the LP is saved, and new events are sent to other LPs. The event is then moved into the processed queue and a new event is scheduled into the LTSF queue. To summarize, the worker thread processing loop is shown in psuedocode in algorithm 2.

```
  while termination not detected do
1     $e \leftarrow$ getNextEvent()
      $lp \leftarrow$ receiver of $e$

      if $e <$ last processed event for $lp$ then
          rollback $lp$

      if $e$ is an anti-message then
          cancel event with $e$ if possible
3         schedule new event for $lp$
          continue

      process event $e$
      save state of $lp$
2     send new events
      move $e$ to processed queue
3     replace scheduled event for $lp$
```

Algorithm 2: WARPED2 Main Event Processing Loop

When events are inserted into the unprocessed queues, they are compared to the currently scheduled event for the LP. If there is no currently scheduled event or the new event is less than the currently scheduled event, then the new event is scheduled to prevent a rollback from occuring.

7.2.1 Ordering of Events

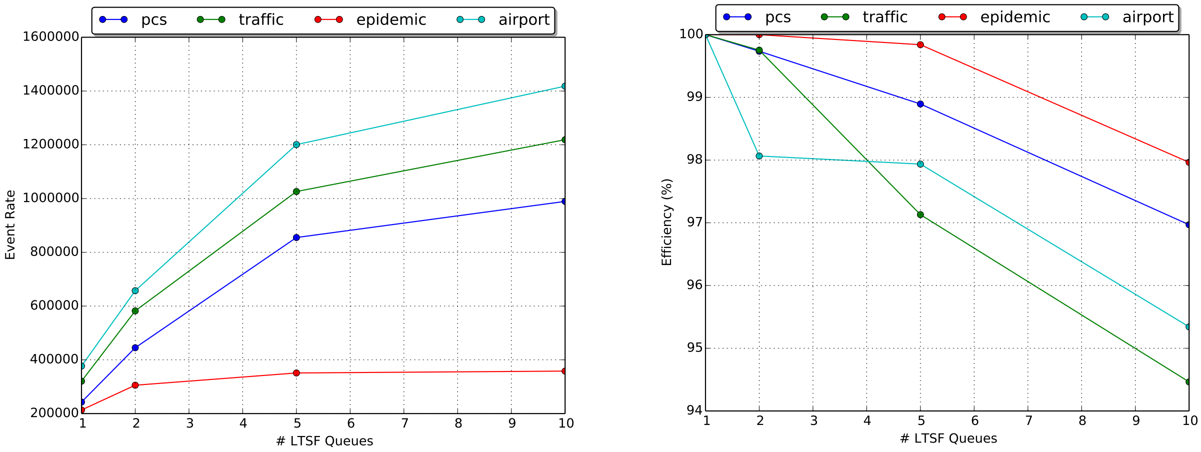
The time warp mechanism assumes that every event is labeled with a totally ordered clock value based on virtual time [4]. This is important so that causal dependencies are preserved and so that simulation results are deterministic [17]. To ensure this, WARPED2 uses a 4-tuple scheme which provide a total ordering of events:

1. Receive Time
2. Send Time
3. Sender LP Name
4. Generation

The last three serve as a tie breaker for simultaneous receive times. The send time is necessary to ensure correct causal dependencies. It is analogous to a Lamport logical clock [18] in real time distributed systems but for virtual time systems. The send time will work as long as LPs only send a single event with the same send time, receive time, and receiver LP. Otherwise, a more strict ordering of sends would be required. However, WARPED2 forbids this behaviour in the simulation model. The sender LP name is necessary to ensure that order is determined between events received with the same receive time and send time from different senders. Without it, it is possible that different results could occur on different runs of the simulation [17]. The generation is necessary for distributed memory systems to differentiate between the same events which could be resent after rolling back [17]. In WARPED2 it is implemented as a single counter per LP and keeps track of the number of events that have been sent from the LP. The count is then tagged in the event when it is sent and used for comparison at the receiver.

7.3 Performance Tuning Parameters

To allow WARPED2 to run on a large range of machines with different architectures and characteristics, a number of parameters are provided for the user to tune the simulation mechanism for their needs. In the



Event Rate $Efficiency$

Figure 7.2: LTSF Queues: Performance on SMP Machine

current implementation, the four main parameters that will determine performance on a single SMP machine are the number of worker threads, the number of LTSF Queues, the LTSF partitioning method, and the state saving period.

7.3.1 Worker Threads and LTSF Queues

The number of worker threads that should be used will depend on the number of processors of the machine so it is left as a tunable parameter. WARPED2 also allows the user to choose the number of LTSF queues. The user can specify any number of LTSF queues, however, it must be smaller than the number of worker threads and also be a factor of the number worker threads.

If all worker threads share a single LTSF queue then a huge contention point is created and performance will suffer. At the opposite extreme, however, with all worker threads processing events from a seperate LTSF queue, the the critical path of the simulation will be spread out and the number of rollbacks can increase significantly. The extent to which these matter varies depending on the characteristics of the simulation model. Furthermore, they may also vary as the data structures and algorithms in WARPED2 evolve to minimize contention.

The plots shown in figure 7.2 illustrate the effects of increasing the number of LTSF queues with 10 worker threads.

Even though the number of total rollbacks increase with more LTSF queues, there is still a growing speedup. That means that the contention of the LTSF queue locks outweighs the extra time needed to process more rollbacks. By looking back at figure 2, we can see all the critical points where worker threads can access the LTSF queues. The lines with a number on the margin indicate possible access to the LTSF queue. For every event the LTSF queue is accessed at least 3 times: (1) obtaining the next event, (2) sending new events, and (3) rescheduling new events . However, access to the LTSF cannot be minimized further by refactoring. Access to the LTSF queue will always be necessary to obtain the next event and reschedule events and since the LTSF queue is only accessed while sending an event if the event is a straggler, it will save a future access to the LTSF queue when the straggler is scheduled and also prevent a rollback.

It is also worth noting that these simulation models are configured for only 10,000 LPs and with larger models, the efficiency would be expected to be better. Therefore, scaling up the number of worker threads and LTSF queues together may not hinder performance if the simulation model is also scaled up since efficiency and contention will not get worse.

7.3.2 LTSF Queue Partitioning

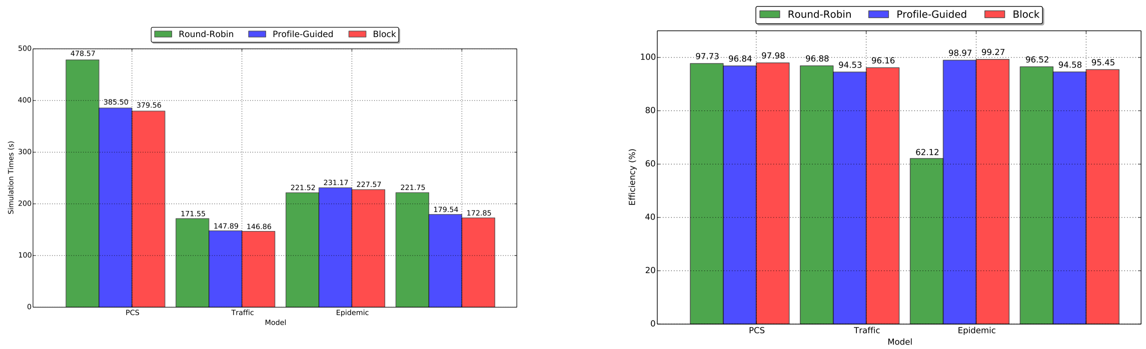
The LPs can also be partitioned between LTSF queues in various ways just like partitioning between processes. In fact, there is only a single partitioning parameter for both levels of partitioning. This makes sense because the LTSF queue partitioning will be neglible compared to the interprocess but it allows the user to choose the LTSF queue partitioning directly on a single node simulation.

Figures 7.3 and 7.4 show the effects of LTSF queue partitioning on simulation runtime and efficiency for the Intel® Xeon® E5410 processor and Intel® Xeon® X5675 processor, respectively.

The simulations on the Intel® Xeon® E5410 are configured with 8 LTSF queues, and 8 worker threads and the simulations on the Intel® Xeon® X5675 are configured with 10 LTSF queues, and 10 worker threads.

Just as with the interprocess partitioning, the block partitioning scheme tends to give the best average performance. However, only the epidemic model has any significant difference in average efficiency. This can be explained by noting that the epidemic model has a very slow event rate compared to any other model so the efficiency is easy to diminish when the number of rollbacks increase.

Another interesting observation is that the round-robin partitioning is much worse on the Intel® Xeon®



Simulation Times (s)

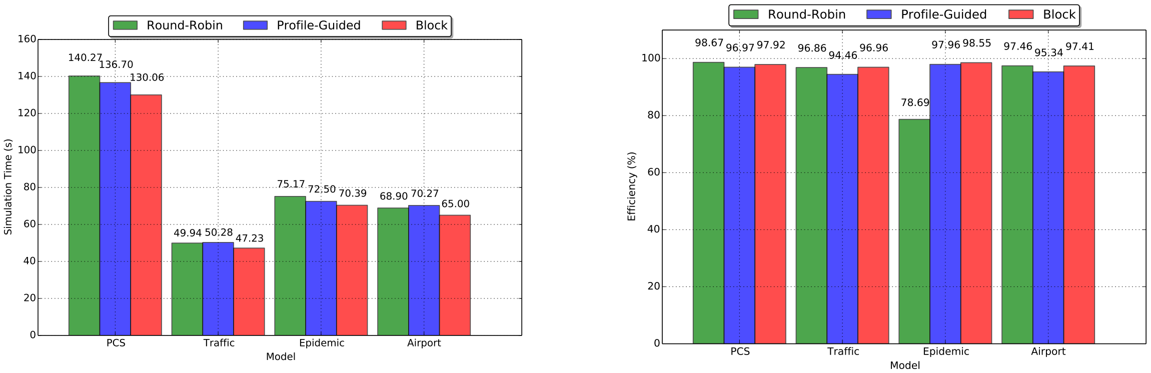
Efficiency

Figure 7.3: LTSF Partitioning: Intel® Xeon® E5410 Processor

E5410 for the PCS, traffic, and airport models but not on the Intel® Xeon® X5675. This, however, is simply a matter of the simulation model properties, and configurations of the model and kernel. The LPs of these models are connected as a grid of 100 x 100 LPs. Since the number of LTSF queues is a factor of the grid dimensions, the round-robin partitioning will still provide north-south locality. The same is not true with 8 LTSF queues so the simulations will be more unbalanced.

7.3.3 State Saving Period

Instead of saving the state of the LPs after every processed event processed, WARPED2 allows the user to choose a state saving period value, N, so that each LP only has its state saved every N events. A larger value of N reduces the amount of time taken to copy states and thus speeds up forward computation. The



Simulation Times (s)

Efficiency

Figure 7.4: LTSF Partitioning: Intel® Xeon® X5675 processor

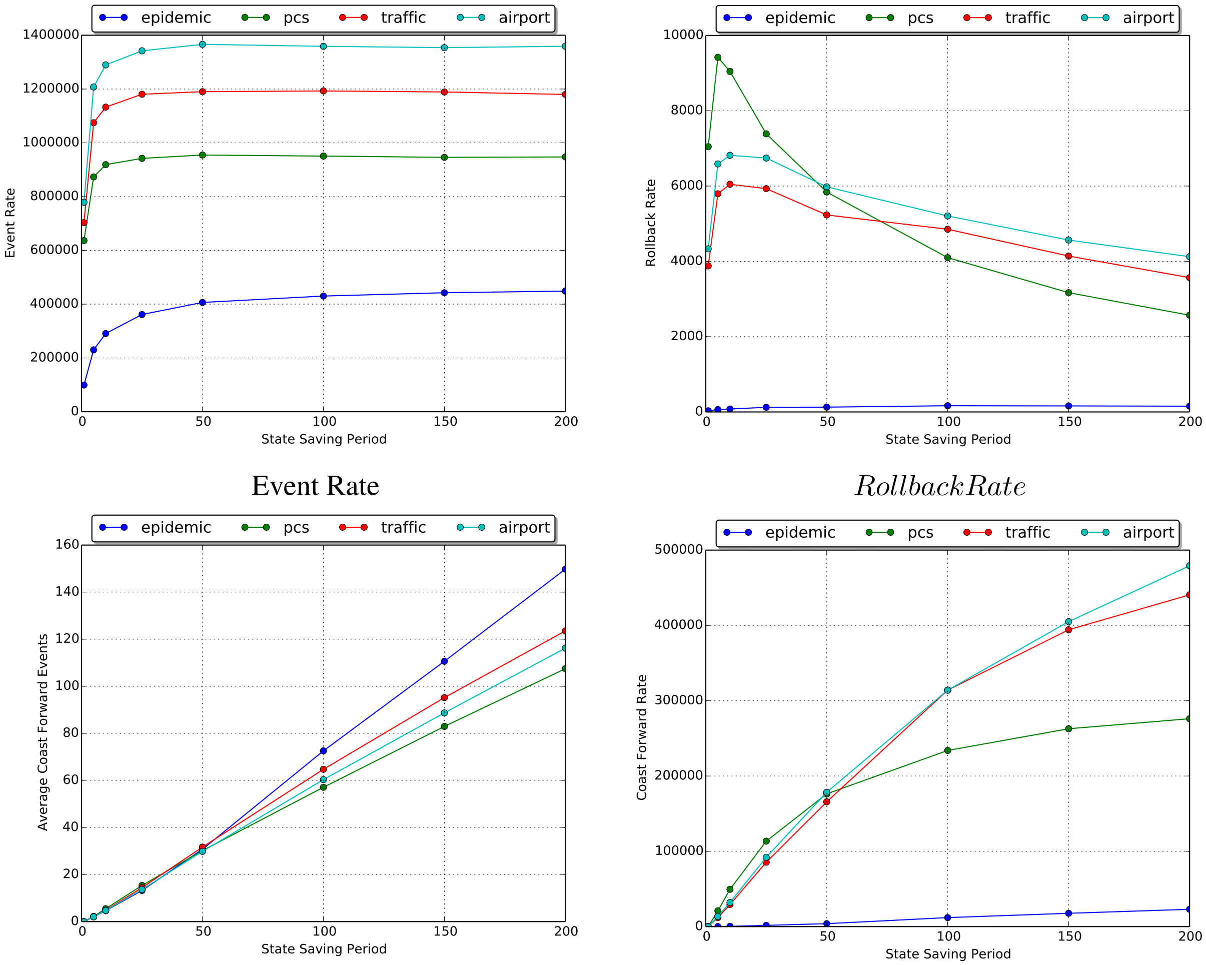
downside is that it also increases rollback time since not all states are available to roll back to, making it necessary to process more events to get to the necessary state in a process known as *coast forwarding*. The total time spent rolling back, however, is small compared to the time taken for forward computation for moderate values of N.

Figure 7.5 shows the event rates and average number of coast forwarded events per rollback as the state saving period is increased for all models on a single SMP machine. The coast forward rate and rollback rate are also used to analyze state saving and are related by:

$$CoastForwardRate = \frac{CoastForwardEvents}{Rollbacks} * RollbackRate$$

For a single SMP machine, the speedup increases significantly for low state saving periods and flattens out pretty quickly. The initial speedup is a result of decreasing forward computation time by removing extra copying. As the state saving period increases though, the coast forward increases to the point where it will cancel out the gains time from reduced copying. An interesting result is that even though the rollback rate increases for lower state saving periods, a speedup is still observed. The coast forwarded rate depends on both the rollback rate and the average number of events coast forwarded per rollback. The rollback rate decays exponentially but the average number of coast forward events per rollback increases linearly. Therefore the coast forward rate still increases logarithmically as the state saving period increases since the average number of events coast forwarded per rollback grows at faster rate than the rollback rate decreases. Similar results can be seen in figure 7.6 for a cluster.

There are few key observations that become apparent on a cluster that are not seen on a single SMP machine. The most apparent is that once the state saving period is increased large enough, a noticable jump in the rollback rate starts degrading performance. This is easily explained by looking at the magnitude of the rollback rates and coast forward rate for a cluster versus a single SMP machine. The cluster has a much larger coast forward rate due to an increased rollback rate which is a result of an increased discrepancy in communication latency and event processing time.



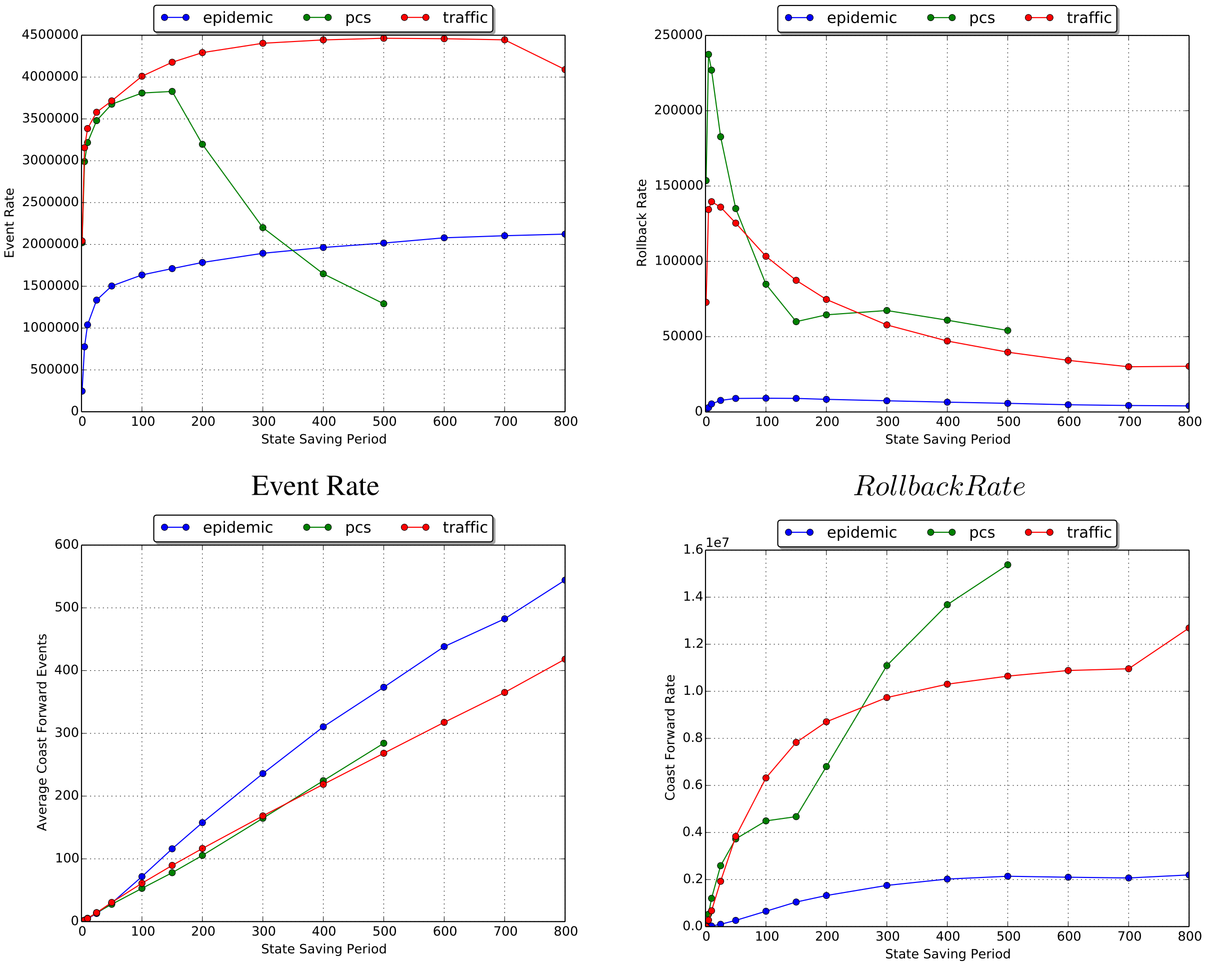
Event Rate

$RollbackRate$

$\frac{CoastForwardEvents}{Rollbacks}$

$CoastForwardRate$

Figure 7.5: Periodic State Saving: Performance on SMP Machine



Event Rate

$RollbackRate$

$\frac{CoastForwardEvents}{Rollbacks}$

$CoastForwardRate$

Figure 7.6: Periodic State Saving: Performance on Cluster

7.4 Protecting Access to LTSF Queues

Worker threads must be able to access shared data structures without the possibility of race conditions. To deal with this, WARPED2 uses mutexes to serialize worker thread access to any section of code that accesses the LTSF queue. This method, however can significantly slows down access to the LTSF queues. In the default configuration, blocking mutexes are used which means that if a thread cannot aqcuire a the lock, it will be put to sleep and rescheduled for execution at a later time. Blocking mutexes method can waiste a lot of exta time if access to the critical section is quicker than putting the thread to sleep and waking it up again. All of the accesses to the LTSF queues are quick in WARPED2 so the locks should only held for a short period of time.

A *spinlock* is a type of mutex that does not force threads to be rescheduled but instead continue to attempt to aqcuire the lock over and over until it successfully acquires the lock or the timeslice of the thread expires. The downside of spinlocks is that the CPU time can be waisted, especially as the number of contending threads increase. Furthermore, if the number of total threads is larger than the number of available processors, spinlocks are even worse because a thread can be involuntarily pre-empted while holding the lock and entire timeslices can be waisted.

7.4.1 Spinlocks in WARPED2

WARPED2 allows the user to optionally build the kernel to use spinlocks instead of blocking mutexes. It is not set as the default because spinlocks do not work well in all cases such as with a system that is running more threads than processors. There are many ways to implement spinlocks with all varying attributes and complexities. Ticket locks were chosen for WARPED2 because they are simple to implement, provide fairness to all threads, and generally have good performance for a small number of threads [19]. A short psuedocode algorithm to illustrate the ticket lock aqcuisition and release procedures is shown below in algorithm 3.

To acquire the lock, each thread atomically increments a counter and takes the old value as a ”ticket” to provide FIFO ordering. Then when the central ticket cound reaches the ticket number of the waiting thread, access to the critical section is given. To release the lock, the thread simply increments the central ticket counter and allows the next thread to access the critical section. If there is no contention for the lock, the

```
lock
    $my\_number \leftarrow$ fetchAndIncrement(next_number)
    repeat until $ticket = my\_number$

unlock
    fetchAndIncrement(ticket)
```

Algorithm 3: Ticket Lock Procedures [3]

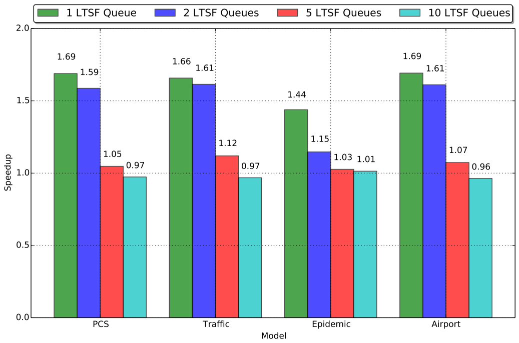


Figure 7.7: Speedup with Spinlocks

next number will be equal to the ticket number and a thread can access the critical section immediately.

All previous experiments were run with the default blocking mutexes in which threads will yield the processor when a lock is contended. To reduce the average lock acquisition latency, the kernel can be configured with spinlocks instead. Figure 7.7 shows the speedup for all models and varying number of LTSF queues. As expected, the speedup is much greater with more contention when there are more worker threads per LTSF queue. It can also be seen that all models except for epidemic have very similar speedups. This is probably due to the high cost of saving such a large complex state. Contention becomes less of a concern if the event processing is slowed down by other computation.

7.5 Data Structures to Support Rollback and Cancellation

Jefferson [4] described the rollback and cancellation process in terms of three data structures: the input queue, output queue, and state queue. The data structures that are implemented in WARPED2 follows pretty closely to Jefferson's description except that in Jefferson's approach, the input queue holds both unprocessed and processed events whereas WARPED2 has two seperate data structures to allow easier scheduling of new

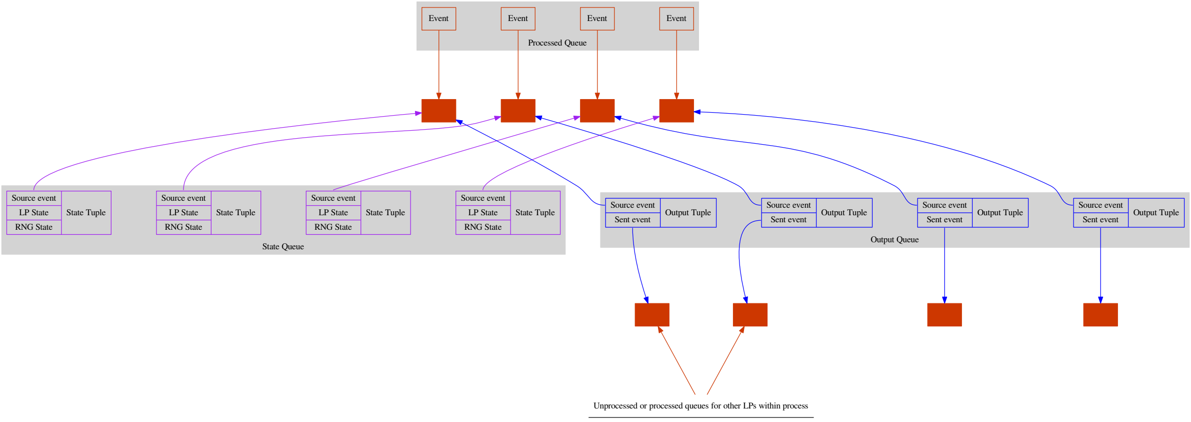


Figure 7.8: Rollback and Cancellation Data Structures in WARPED2

events.

The state queues contain a 3-tuple value for each state. The first value is a pointer to a memory location which holds the saved copy of the LP state. The second value is a pointer to the event that produced the state of the LP and is used for comparison against a straggler event on a rollback to determine which state to restore or against the GVT during fossil collection. The third value is the state of the LPs random number generators at the time the state was saved. The random number generator states are saved in a linked list and are also restored during a rollback to ensure deterministic results.

Similar to the state queue, the output queue also contains a tuple of values. In the output queue though, only a pointer to a source event and a sent events are used. In the same way as the state queue, the source event is used for comparison against a straggler event during a rollback but is used to determine which events should be sent as anti-messages.

The processed queues only contain a pointer to the processed events. The dereferenced events from the processed queue are the same as dereferenced source events from output queue and state queue so that no copying of the events is needed. Furthermore, if events are sent within the same process, the same events can be pointed to from the output queue and an unprocessed or processed event from another LP. The organization of the processed queue, state queue and output queue for a single LP is illustrated in figure 7.8.

Although the organization of these data structures prevents copying of events, there are also some limitations. First, it introduces a lot of indirect memory accesses which could slow down access to memory, especially in a multiprocessor system. Secondly, it adds some complexity when freeing memory used by events. To solve this problem though, WARPED2 uses the smart pointers offered by the standard C++11 libraries which employ a reference counting scheme.

Chapter 8

GVT and Termination Detection

Algorithms that compute the Global Virtual Time (GVT) and detect termination conditions are both examples of algorithms that can be solved by determining the global state of a distributed system. The global state of a system is defined as the combination of all the local states of the processes in the system and all messages in transit. Global state determination algorithms are also commonly used for deadlock detection, garbage collection, debugging, and checkpointing for failure recovery in distributed systems.

8.1 Global Snapshots

Chandy and Lamport [20] described a basic global state algorithm by using *global snapshots* for distribtuted systems that use only FIFO channels for communication. To start the algorithm, an initiator process records it's state and sends a control token out of all outgoing channels. When a process receives a control token, and it hasn't yet recorded its state, the process records its state and sends more control tokens out of all its outgoing channels. The algorithm terminates at each process when it has recieved a token through all of its incoming channels.

Lai and Yang [21] describe an algorithm for non-FIFO systems which computes a consistent cut by piggybacking a control bit onto basic messages. The control bit is used to indicate whether or not the sending process has recorded its state. The processes can be explained in terms of colors. A process that has not recorded its state is colored white and a process that has recorded its state is red. White processes send white messages and red processes send red messages. The control bit in the messages indicates the color. All processes are initially white and turn red when a red message arrives. When a red message arrives at a white process, the process must record its state BEFORE actually receiving the message. The snapshot only relies on basic messages and no control tokens are used. This algorithm has several drawbacks. First, it assumes that every process will eventually receive a red message and record its state which is not guarenteed. Second, to ensure transient messages are recorded, the processes must record all incoming and outgoing messages and send them to other processes within the basic messages. That way the transient messages can be calculated by differences in incoming and outgoing messages.

Mattern [22] extended the algorithm by Lai and Yang by adding a seperate control token which is used to create two cuts by cirulating the control token to every process twice. The control token is used to color the processes instead of the basic messages. This guarentees that every process will eventually record its state because the token is always circulated. Furthermore, processes do not have to keep track of sent and received messages. Instead counters can be used to keep track of the differences in sent and received white messages at each process. The control token can then accumulate the counts. When the white message counts accumulate to zero when the token arrives back at the initiator proccess, it can be determined that the snapshot is complete. These counters can be *vector counters* or *scalar counters*. Vector counters keep track of messages to/from each process individually whereas scalar counters keep track of just a single count at each process. When the algorithm is complete, only the initiator process can produce a global snapshot so if all processes must use the snapshot, it must be broadcasted to the other processes.

8.2 GVT

Although GVT algorithms can be implemented with the basic global snapshot algorithms described above, it is usually more efficient to build custom solutions on top of the basic algorithms. In a GVT algorithm, the local minimum clock of process is the local state and the basic messages are events that are sent between processes. This section first describes the key ideas that must be considered when developing a GVT algorithm. Then, a few of the classic GVT algorithms and modern GVT algorithms that are commonly used in practice today are described. Finally, the algorithm that is implemented in WARPED2 is described as well as the reasons for choosing the approach.

GVT algorithms can be synchronous or asynchronous. In a synchronous GVT algorithm, since all other computation will be blocked, event processing will be halted. Synchronous GVT algorithms are usually very simple to implement but halting the event processing can be very costly. On the other hand, asynchronous GVT algorithms calculate GVT concurrently with event processing. For this reason, asynchronous GVT algorithms perform much better than synchronous GVT algorithms but are harder to implement because special cases must be considered.

There are two special cases that must be considered in a GVT algorithms:

Transient Message Problem: is caused by messages(events) that have been sent by the sending process but not yet recieved by the receiving process. If not carefully considered in a GVT algorithm, these *transient messages* can be completely missed which could lead to an erroneous GVT calculation if they contain the minimum timestamp of all other events in the system.

Simultaneous Reporting Problem: is caused because processes can report their local minimum clock at different points in real time. Consideration must be taken into account to ensure that a process does not report its local minimum clock value and then receive an event from a process that has not yet reported its local minimum clock value, completely missing the event.

There are many synchronous and asynchronous GVT algorithms that have been developed over the years that all have some method of either solving these problems.

8.2.1 Asynchronous GVT Algorithms

Message Passing Algorithms

Most of the time warp systems in use today are based on message passing and classically GVT algorithms have been designed for message passing systems. In this section Samadi's and Mattern's message passing algorithms are discussed and in the next section Fujimoto's shared memory GVT algorithms is discussed as well as the Seven O'Clock algorithm which is an extension of Fujimoto's shared memory algorithm extended for distributed memory systems.

Samadi's Algorithm [23] in the most general form uses acknowledgements for all events that have been received. All processes must track all events that have been sent but have not been acknowledged. Furthermore, the received messages must also be tracked so that acknowlegements can be sent. All transient message can then be calculated from the tracked messages. A process initiates the GVT algorithm by broadcasting a start message to all processes. After this start message is received by a process, it is marked(colored) and all acknowledgements sent from a marked process are also marked(colored). All processes then calculate their local minimum by taking the minimum of the unacknowledged received events, the marked acknowledgments sent, and the local simulation clock. Marking the acknowledgements sent after the start of the GVT calculation ensures that the simultaneous reporting problem will not occur.

Mattern's Algorithm [22] for GVT calculation is an extension on his general snapshot algorithm that was described above. The white message counts are used to determine whether transient messages are still in the sytem. They also serve as the basis for determining when the snapshot is complete which occurs when the accumulated white message count of all processes is zero. To ensure that the simultaneous reporting problem does not occur, red messages sent are recorded and used in the local minimum value of the sending process. The algorithm is initiated by sending a control token to all processes in some defined order. The token accumulates the white message counters, local minimum clocks, and minimum red message timestamps. When the accumulated count reaches zero and the token is back at the initiator process, the GVT is approximated using the minimum of the accumulated clocks and red message timestamps.

Other GVT Algorithms that are based on message passing model are usually based on the same ideas from Samadi's algorithm or Mattern's algorithm. Many algorithms are just extensions or variations of the algorithms that aim to optimize it in some particular way.

Shared Memory Algorithms

Fujimoto's Algorithm [2] is a fast GVT algorithm that exploits properties of shared memory architectures. In most shared memory architectures, processors cannot observe memory operations in different orders. For this reason, it is not possible to have transient messages if a shared data structure is used to communicate between tasks running on different processors. Furthermore, a shared flag variable can be used to initiate the GVT algorithm. However, it is still possible that the flag can be read at different times, so the simultaneous reporting problem can still occur. To solve the simultaneous reporting problem, two things must be done, First, the start flag is checked after sending events and recorded if the sending task has not yet reported its local minimum value and is eventually used when the reporting is done. Second, the start flag must be read into a temporary local variable before obtaining a new event to process.

Seven O'Clock Algorithm [24] is an extension of Fujimoto's algorithm for distributed memory systems. Although the algorithm can be used in message passing systems, the algorithm does not require any messages and is still uses shared memory ideas. The key idea in the algorithm is that all processors in a distributed system all have a consistent view of wall clock. Hence, the processors can carry out an operation in an atomic manner without having to explicitly interact. The atomicity can be achieved by using cycle level counters which are available on most modern architectures. Unlike Fujimoto's algorithm though, transient messages can still be missed in the calculation. To solve that problem, each processor must wait a small time interval which is calculated based on the worst round trip time for network transactions.

8.2.2 Synchronous GVT Algorithms

Global Reductions provide a good way to do a synchronous minimum calculation which and is a common way to implement a synchronous GVT algorithm. ROSS implements a synchronous GVT algorithm which uses global reductions to implement a variant of Mattern's algorithm [25]. First, to prevent the transient message problem, a global reduction on message counts of each process is performed until the total reaches zero. Any events received after the start of the algorithm are recorded and included with the local minimum of the receiving process. When the messages count reaches zero a global reduction is performed on the local minimum timestamp of all processes which yields a GVT value for all processes. This algorithm is efficient when time warp simulation are run on large supercomputing platforms like the blue gene machine which perform collective operations very quickly.

8.2.3 GVT in WARPED2

WARPED2 implements both synchronous and asynchronous GVT algorithms. For the asynchronous GVT algorithm, Fujimoto's shared memory algorithm is used between worker threads and a variation of Mattern's algorithm with scalar message counters is used between processes. The synchronous algorithm is based on the global reduction algorithm mentioned above but also includes a reduction between worker threads as well as two thread synchronization barriers.

The details of how a GVT algorithm is integrated into the worker threads in WARPED2 is outlined below in algorithm 4 which shows the main event processing loop with an emphasis on GVT contribution as well as the send routine and the remote event receive routine. This system of GVT integration is fairly ambiguous and should work for any combination of shared memory and message passing GVT algorithms whether they are synchronous or asynchronous.

```
processEvents
    while termination not detected do
        $localGVTFlag_i \leftarrow localGVTFlag$
        $e \leftarrow$ getNextEvent()
        $lp \leftarrow$ receiver of $e$
        if $localGVTFlag_i$ and ($workerThread_i$ hasn't reported local min) then
            reportThreadMin($timestamp_e$)
        if $e$ is straggler then
            Rollback $lp$
            continue
        $newEvents \leftarrow$ processEvent($e$)
        foreach $ne$ in $newEvents$ do
            sendEvent($ne$)

receiveRemoteEvent
    GVTReceiveEventUpdate($timestamp_e$, $color_e$)
    $lp \leftarrow$ receiver of $ne$
    Insert into Unprocessed Queue for $lp$

sendEvent
    if $ne$ is remote event then
        $color \leftarrow$ GVTSendEventUpdate($ne$)
        Insert into Send Queue
    else
        $lp \leftarrow$ receiver of $ne$
        Insert into Unprocessed Queue for $lp$
    if $localGVTFlag_i$ and ($workerThread_i$ hasn't reported local min) then
        reportThreadSendMin($timestamp_n e$)
```

Algorithm 4: GVT: Worker Threads

The routines for `ReportThreadMin`, `GVTReceiveEventUpdate`, `GVTSendEventUpdate`, and `ReportThreadSendMin` are intentionally abstract since they will vary with the exact GVT algorithm implementation. In general though, the purpose of each doesn't really change. The purpose of each is:

ReportThreadMin is used for the worker threads to report a minimum event timestamp which will usually be the current event since the LTSF queue orders events.

ReportThreadSendMin is used for the worker threads to report any events that are sent after the localGVTFlag has been set and it has not yet called `ReportThreadMin`.

GVTReceiveEventUpdate is used to update any message counters, send acknowledgements or record the timestamps of received events.

GVTSendEventUpdate is used to update any message counters, track sent messages, or get the current color of the process to tag events being sent.

This system of GVT calculation will work as long as the localGVTFlag indicator is retrieved before obtaining the next event and the sends are reported after sending an event. The manager thread controls the initiation and termination of the GVT algorithms and vary greatly depending on the exact implementation.

Asynchronous

The asynchronous GVT algorithm in WARPED2 is based around a variant of Mattern's algorithm for the message passing part. In the WARPED2 implementation, scalar counters are used to keep track of whites message instead of vector counters. That means that each process keeps track of the number of sent messages minus the number of received messages to and from all other process as a single count. The reason that scalar counters are used over vector counters is that vector counters require extra space and are only necessary if communication is stopped after receiving a control token until all known transient messages that are bound for the receiving process are received. With scalar counters, this is not possible because all counts are combined into a single counter. However, with scalar counters it is possible that more than two rounds of the control token will be necessary. This is usually not a problem in practice though since two rounds is usually sufficient. Also, each process must maintain a counter for both white messages *and* red messages so that counters can be consistent between multiple runs of the algorithm. The roles of the colors must also be switched between successive runs.

Another major different between the classic Mattern algorithm and the variant in WARPED2 is that the classic algorithm assumes that the timestamps of the white messages that are received will be processed right away and counted in the minimum clock value when a token is received. Since this is not the general case in WARPED2, a running minimum of the white message timestamps received is recorded and included in the minimum clock value when a token is sent to ensure that all white messages are included in the final GVT value.

A GVT token will be sent by the manager thread when a the local GVT calculation based on Fujimoto's shared memory algorithm is completed. The local GVT calculation will begin either when a token is received or when the initiator process determines that is should start. The initiator process is the process with PID of 0 and the start of a new calculation is some fixed time interval from the start of the previous calculation.

The asynchronous algorithm desribed above works well in cases where the remote communication is relatively low but has been shown to be problematic under heavy remote communication. The root of the problem is that the GVT token must compete with remote events that must be sent and received. Based on some observations that show large message counts, the bigger problem with being able to complete receives quickly. To get around this constraint, users can either increase the number LPs or decrease the number of worker threads to reduce the network load.

Synchronous

The synchronous GVT algorithm is a simple extension of the reduction algorithm implemented in the ROSS simulation kernel. The biggest difference is that the manager thread must block the worker threads from processing any more events. Also, unlike the algorithm used in ROSS, the processes are explicitly colored so that any events that are inserted into the remote send queue after the GVT calculation is started are known to be included with the sending process's local minimum value. The algorithm executed by the manager thread is summarized in the psuedocode in algorithm 5.

When the worker threads report their local minimum, they must wait at *two* thread barriers. Both barriers

```
progressGVT
    $color \leftarrow RED$
    $localGVTFlag \leftarrow 1$
    $THREAD\_SYNCHRONIZATION\_BARRIER$

    while $true$ do
        Flush send queue
        Receive messages
        sumAllreduce(localMsgCount, globalMsgCount)
        if $globalMsgCount = 0$ then break

    $localMin \leftarrow recvMin$
    foreach worker thread, i do
        $localMin \leftarrow \min(localMin, \min(localMin_i, sendMin_i))$

    minAllreduce(localMin, GVT)

    $color \leftarrow WHITE$
    $localGVTFlag \leftarrow 0$
    $THREAD\_SYNCHRONIZATION\_BARRIER$
```

Algorithm 5: Synchronous GVT: Manager Thread

are needed so that the manager thread can ensure that no more remote events are sent. When every worker thread calls the first barrier, it indicates to the manager thread that it all local minimums have been reported and it can continue. The second barrier is an indicator to the worker threads so they can be released and continue processing events.

The advantage of the synchronous GVT algorithm in WARPED2 is that it is very robust and works well even under very heavy remote communication. Furthermore, it will be produce the exact GVT unlike the asynchronous algorithm since event processing will not continue until it is complete so the GVT cannot increase.

8.3 Termination Detection

Termination detection, like GVT, is a problem of determining the global state of the system. Termination is a *stable property* of a distributed system, meaning that once termination conditions occur, the system will remain with termination conditions forever until further action is taken.

A process in the system can be in one of two states at any time: active or passive. A process is considered active if some basic computation still remains and passive otherwise. When all processes in the system become passive and no messages are left in transit, then the system should be terminated. The purpose of the termination detection algorithm is to determine when this occurs. A passive process can become active with the arrival of an *activation message*. For parallel discrete event simulation the basic computation is event processing and the activation messages are events. A termination detection algorithm for any parallel discrete event simulation must satisfy the following properties:

Safety Property: Termination will not be detected if any unprocessed event is still present in system which includes all unprocessed events that have been received at each process and events still in transit.

Liveness Property: Termination will be detected at some finite amount of time after all events have been processed.

Just like GVT algorithms, termination detection algorithms can implemented with message passing or shared memory communication. Termination algorithms vary widely because different systems can define termination conditions in such different ways. Furthermore, termination usually does not affect system performance as long it does not block event processing so correctness is more important than optimization. The rest of this section will only focus on the algorithms that are implemented in WARPED2.

8.3.1 Termination in WARPED2

The termination detection algorithm in WARPED2 is actually two independant algorithms, a message passing algorithm and a shared memory algorithm, merged into a single algorithm. In the opposite manner as the GVT algorithm, the shared memory algorithm is actually used to initiate the message passing algorithm.

Shared Memory algorithm

The purpose of the shared memory algorithm is to determine when all of the worker threads in a process become passive (inactive). The algorithm is fairly straightforward and uses just a shared counter variable to keep track of the number of active worker threads and a array to keep track of the state of each worker thread. Each worker thread updates their own state when they change and atomically update the counter.

The manager thread periodically checks to see if the counter has reached zero. If the count has reached zero and the process is the master, then the interprocess message passing alorithm is initiated. The integration of the shared memory algorithm is illustrated in algorithm 6. To allow the GVT calculation to continue, worker threads in a passive state must still report an infinite local minimum value.

```
processEvents
    while termination not detected do
        $localGVTFlag_i \leftarrow localGVTFlag$
        $e \leftarrow$ getNextEvent()
        if $e \neq nullEvent$ then
            if $state_i = PASSIVE$ then
                $state_i = ACTIVE$
                $activeCount \leftarrow activeCount + 1$

            $lp \leftarrow$ receiver of $e$
            if $e$ is straggler then
                Rollback $lp$
                continue

            process event $e$
            send new events
        else
            if $state_i = ACTIVE$ then
                $state_i = PASSIVE$
                $activeCount \leftarrow activeCount - 1$
            if $localGVTFlag_i$ and ($workerThread_i$ hasn't reported local min) then
                ReportThreadMin($\infty$)
```

Algorithm 6: Termination: Worker Threads

The safety property is not necessarily achieved with this algorithm because it is possible that the last active thread sends an event to another LP which will be processed by another thread. Then if the sending thread becomes inactive, all threads can be temporarily inactive and the manager thread can falsely detect termination. However, it is very unlikely that the sending thread will report itself as passive before the receiving thread gets the event and reports itself as active. Furthermore, the manager thread is forced to check the active thread counter at least twice with no changes in any worker thread state. The challenge with completely solving this problem is that it is not known which thread will process the event since LPs are mapped to LTSF queue and not directly to worker threads.

The liveness property is achieved because the manager thread periodically checks the active worker thread count so the inactivity of all worker threads will eventually be discovered.

Message Passing Algorithm

The algorithm that is carried out among processes is an asynchronous message passing algorithm based on Mattern's "sticky flags" algorithm [22]. Just like the GVT algorithm, the token is passed in a logical ring to increasing PIDs. However, the initiator process of the algorithm can change with each execution of the algorithm. The initiator of each successive execution is the first active process that the token reaches during the previous circulation. For the first circulation, the initator is the process with the PID of 0.

Each process has two state indicators, one for the actual state and one for the *sticky state*. When the sticky state of all processes are simultaneously passive, then the simulation can terminate safely. The sticky state becomes active when the actual state becomes active but sticks to active when the actual state becomes passive. It can only change to passive on the arrival of a token which forces the token to circulate two times with no process becoming active and ensures the safety property. Without this scheme, a process that has already forwarded the token because it was passive could receive an activation message and then the sender of the activation message could become passive before receiving the token and false termination could then be detected.

The algorithm is started when the initiator process becomes passive which is determined by the shared memory algorithm. When a process receives the token it becomes the initiator but loses its roles as initiator when it sends the token. That way, only a single process can be the initiator. Furthermoe, since the token always stops at an active process, the token is always guaranteed to start again when it becomes passive which also guarantees the liveness property. When the token is received back at the initiator with a passive state, then termination is signaled to all processes. The procedure is illustrated in algorithm 7.

It should also be noted that this algorithm is only gaurenteed to work if the message order is preserved between any two processes or if it can be determined that no transient messages can exist. For this reason, the termination algorithm also keeps message counters and accumulates the counters from each process just like with the Mattern GVT algorithm.

```
$initiator \leftarrow true$
**if** $state_{sticky} = PASSIVE$ **and** $PID = minitiator$ **then**
    **if** $mstate = PASSIVE$ **and** $mcount = 0$ **then**
        Signal termination
**else if** $state_{sticky} = PASSIVE$ **then**
    $initiator \leftarrow false$
    SendToken ($mstate, minitiator, mcount$)
$state_{sticky} \leftarrow state_{actual}$
```

Algorithm 7: Termination Token Receive

Chapter 9

Memory Management

This chapter will focus on main topics related to memory management in a time warp system and their implementations in WARPED2. The topics will include state saving, fossil collection and memory allocation. The memory requirements and performance for different state saving, fossil collection, and memory allocation techniques can vary greatly.

9.1 State Saving

9.1.1 State Saving Techniques

Copy-state Saving is the classic state saving technique that saves every past state of every LP and requires that the all states are copied and saved into the state queues. This method requires a lot of extra memory and requires a lot of extra computation to copy the states and insert them into the state queue. For this reason, copy-state saving is usually used in conjuction with other technique or not at all.

Periodic State Saving is another state saving technique where the state of each LP is saved only once every N events, where N is a number greater than one. Periodic state saving can significantly reduce the overhead of the time taken to copy the state of the LPs into the state queue. However, due to state history being lost, more processed event have to be saved so that the state of LPs can be restored correctly. These saved events must be reprocessed to restore the state in a process known as *coast forwarding*. During coast forwarding the events are processed normally but only to update the state. The only difference is that no events are sent during the coast forwarding phase. The downside is that the rollback length is increased due to the loss in state saving. However, periodic state saving is a significant improvement over the base copy-state staving approach since the reduced time in state saving far outweighs the extra rollback time for increased rollback length.

Incremental State Saving is a technique that aims to reduce the amount of memory needed to store past states of the LPs. Instead of saving entire snapshots of the past states, the differences in specific state variables are saved. This method requires that metadata describing which variables are modified for each event. Therefore, this approach works well as long as only a small portion of the state variables change during the forward execution of events.

Reverse Computation is a modern approach to state state saving that does not actually save any past states of the LPs. Instead, the *control state* of the forward computation is saved as a set of bits that describe the forward control flow. By saving the control flow of the forward execution for each event, the state variables that are modified and how they are modified are known and can be reversed. This is the case at least if all state variables are reversible without the need to know the histories of the variables. Also, a reversible random number generator is required. The major problem with this approach is the necessity to understand forward and reverse computation precisely in the implementation of the simulation model.

9.1.2 State Saving in WARPED2

WARPED2 currently implements periodic state saving because it is simple to understand and does not require that model developers know anything about it. Furthermore, it can increase performance and decrease memory requirements for any type of simulation model.

9.2 Fossil Collection

When the events and states in the processed queues, output queues, and state queues are no longer needed, the memory that they consume is no longer needed and can either be freed or reused. Fossil collection is the process of reclaiming memory for future use. There are many methods of fossil collection that have been used in practice.

9.2.1 Fossil Collection Techniques

GVT-based Fossil Collection is the traditional method of fossil collection. Since rollbacks cannot occur past the GVT, it can be used be used as a marker to indicate what can be fossil collected. Fossil collection is triggered after every GVT calculation and all memory is explicitly freed so that it can be reallocated for future use.

On-the-fly Fossil Collection is a method of fossil collection used in GTW which does not use GVT as a starting point and memory is not explicitly freed. Instead, after events are processed, they are added to one of the per-processor free memory lists. New events can then be allocated from a list as long the first event in the list has a timestamp less than the GVT. If it does not then the current event being processed is either aborted until the GVT reaches a higher value or the list is searched for a lower timestamp event. This approach however, has been shown to produce instability caused by imbalances in rollbacks which creates imbalances in the free lists, and ultimately leads to a lot of aborted events or a long search time.

Optimistic Fossil Collection is another method of fossil collection which, like on-the-fly aims to reuse memory instead of explicitly freeing it. However, optimistic fossil collection is based on a statistical bound for rollbacks. It is still possible that memory that was fossil collected could still be used. This situation is called an OFC fault. Therefore, methods such as global state checkpointing must also be implemented to recover from these faults.

9.2.2 Fossil Collection in WARPED2

Currently, WARPED2 implements a GVT-based fossil collection. The GVT is calculated by the manager thread but there are two ways that fossil collection can be achieved in WARPED2. The first way is to have the worker threads fossil collect one LP at a time as they process events. The last fossil collect time can be recorded for each LP so that the worker threads can know when fossil collection is needed by comparing the times to the current GVT. The second way is to have the manager thread fossil collect all LPs. Once the GVT is calculated, all LPs can be fossil collected all at once.

With manager thread fossil collection, the worker threads are relieved from unnecessary work that can slow down event processing but extra locks will be needed to protect access to the processed queue, output queue, and state queue for each LP. Worker thread fossil collection will be the more scalable approach for both the number of LPs and the number of processes since the manager thread must perform other tasks such as GVT and interprocess communication.

Figure 9.1 shows how the event rates compare in a single SMP machine with 10 worker threads, 10 LTSF Queues, a GVT period of 100ms, and a state saving period of 25 events.

The results show that on average, having the manager thread do fossil collection will yields a slightly better event rate, but not significant. It is uncertain what will happen when the manager thread has more LPs to fossil collect or if it had remote messages to send and receive. The time taken to fossil collect with more

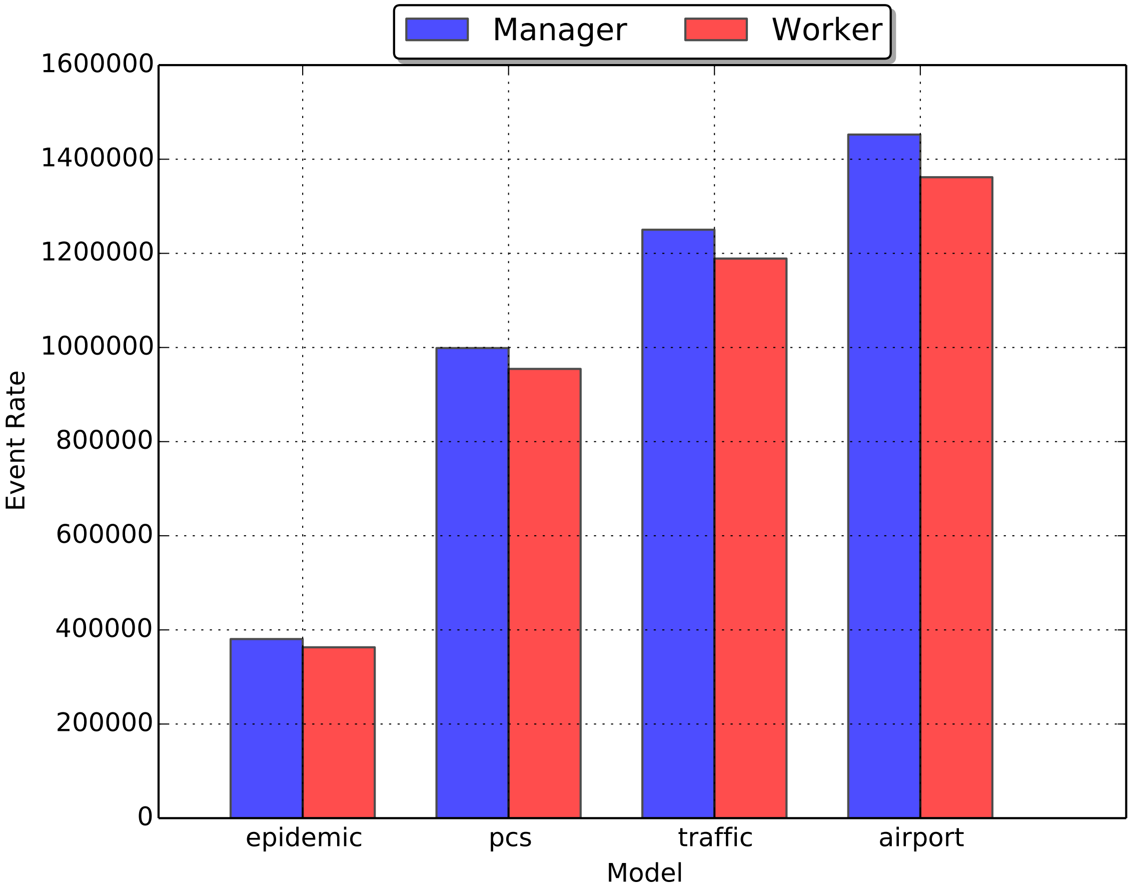


Figure 9.1: Performance Comparison for Fossil Collection Methods

LPs could interfere with sending and receiving remote messages or with calculating the GVT.

One observation is that unstable behavior will occur with manager thread fossil collection if the state saving period is set very low with model that have a large state size such as with epidemic. The GVT calculation tends get longer and longer as the simulation progresses. This could be due to fossil collection interfering with the GVT calculation which would further allow memory to build up faster than it can be freed. Each fossil collection period and GVT period will grow and grow in a positive feedback loop until memory in the system runs out.

Ultimately, a worker thread approach was chosen so that the WARPED2 will be able to safely scale up simulations in terms of both the number of LPs and the number of nodes. We feel that adding better scalability outweighs the loss in the single node event rate, especially since scaling up the number of nodes will still allow faster event rates anyway.

9.3 Memory Allocation

Parallel discrete event simulation systems must frequently allocate and free memory for events and states. This can be a huge overhead if used memory and free memory is not maintained in an efficient way. Furthermore, memory must be allocated to and deallocated from each process by the operating system as needed which can be even more costly.

In a standard system, the application will never know when memory is allocated from the operating system or returned to it because it is usually handled by runtime libraries for the programming language that is used which provide an API for allocation/deallocation for each specific process. To completely remove any OS overheads, some time warp systems allocate a large fixed sized memory space before the start of the simulation and use their own memory management schemes. In these time warp systems, the simulation kernel must provide a way for the simulation models to allocate memory for events that need to be sent instead of calling the standard programming language APIs.

Dynamic memory management in WARPED2 is achieved through the standard C++ library interface and does not pre-allocate any memory. The simulation model is responsible for the allocation of all states and events. The kernel only receives pointers to the states and events through callback functions and accesses them indirectly. By doing this, we allow simulation model developers to implement their own memory management schemes or override the default malloc with allocators such as TCMalloc, JEMalloc, Hoard, etc.

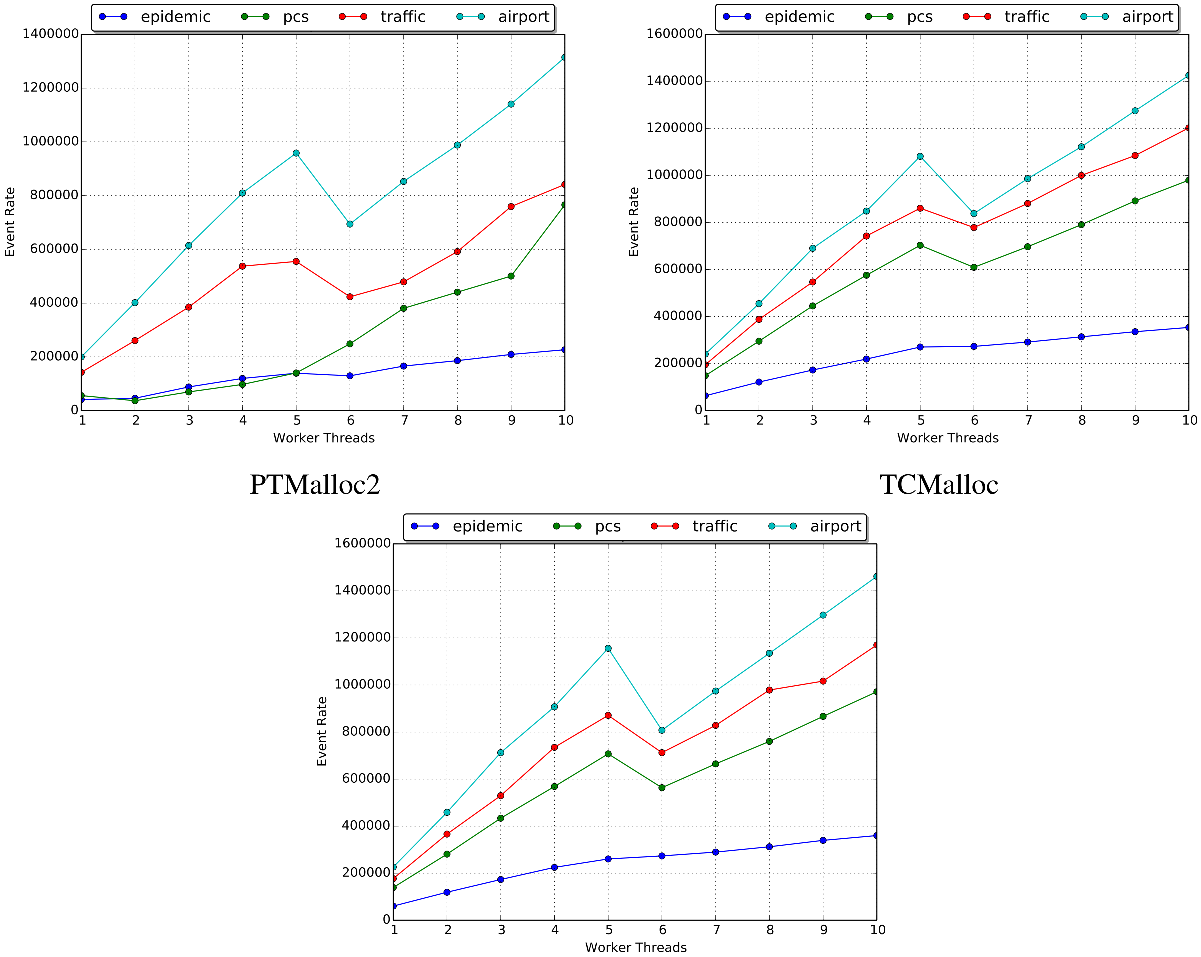
9.3.1 TCMalloc and JEMalloc

TCMalloc and JEMalloc are two good examples of memory allocators that are designed to efficiently allocate memory in multi-threaded applications liked WARPED2. In both TCMalloc and JEMalloc, seperate thread-local caches of free memory are maintained to avoid contention when allocating and freeing memory.

All results shown before now have all been with TCMalloc but until now, we have not shown why. A comparison of the event rates for ptmalloc2 which is the default allocator for GLIBC, jemalloc and tcmalloc is shown in figure 9.2.

TCMalloc is the clear winner in terms of speed and scalability. For the aiport model, it may appear that ptmalloc2 almost performs as well as tcmalloc but this is just a special case since aiport is a smaller model with less LPs and thus requires less memory. The more interesting observation is what happens when the number of worker threads reaches six and the event rate drops. However, it is just a result of the underlying architecture of the Intel® Xeon® X5675. As shown in table A.1 in appendix A, it supports up to 12 simultaneous threads on 6 cores. By adding a sixth worker thread which is the seventh thread counting the manager thread, we may start seeing some bad cache effects such as false sharing, or inconsistent task scheduling. An imbalance can also be observed as a result of these cache effects by looking at the efficiency as shown in figure 9.3.

Figure 9.3 also illustrates how ptmalloc2 can lead to unstable behavior with a smaller number of threads. The reason for this may come from the data structures that are used by the allocators and how often that the OS is called to allocate more memory from the system. Ptmalloc2 has memory blowup problems that stems from the separation of the thread local free memory pools that cannot ever merge back with a central heap.

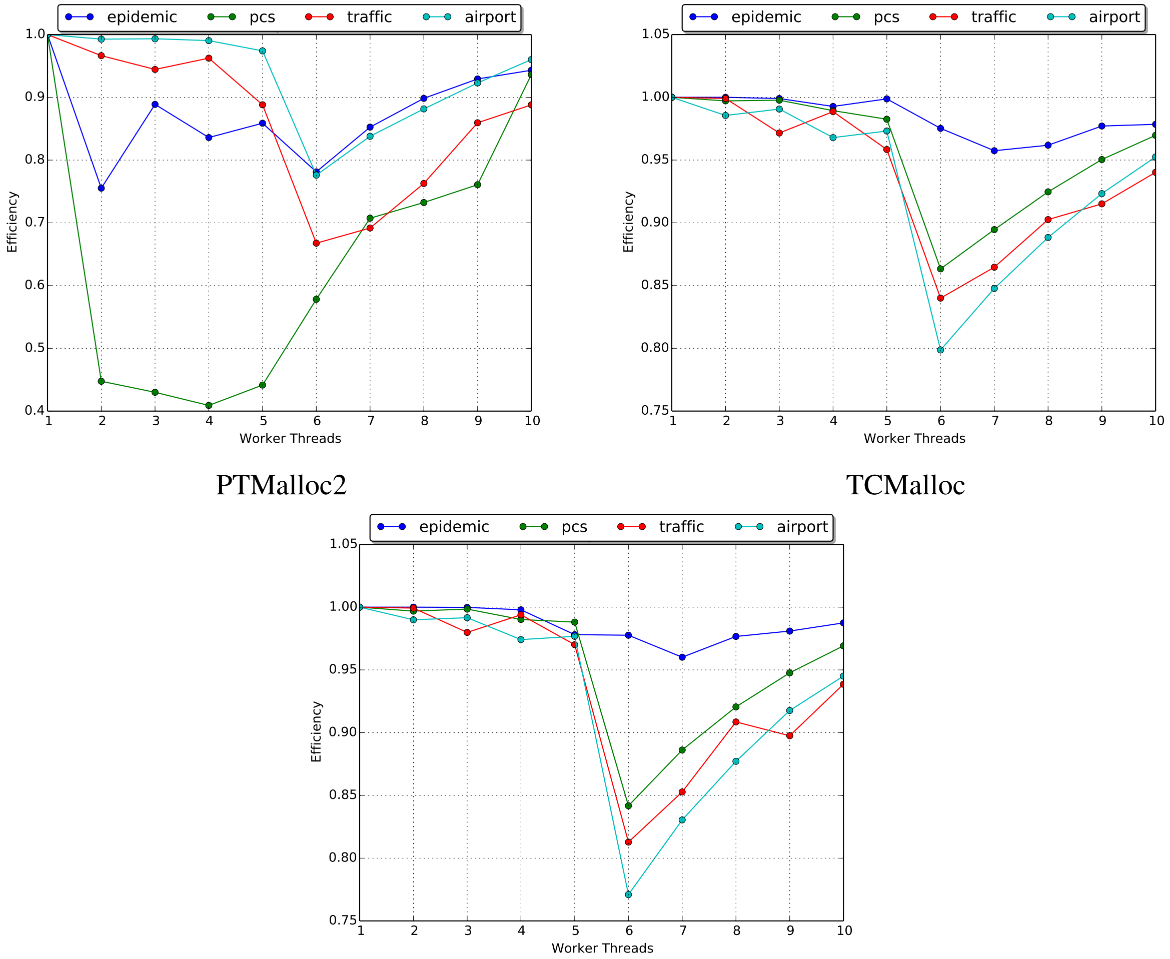


PTMalloc2

TCMalloc

JEMalloc

Figure 9.2: Memory Allocator Event Rate Comparison



PTMalloc2

TCMalloc

JEMalloc

Figure 9.3: Memory Allocator Efficiency Comparison

9.4 Memory Tuning Parameters in WARPED2

To allow WARPED2 to run on machines that might have different memory constraints, some parameters are available for fine tuning the memory footprint of the simulation. The three main parameters that make a difference with regard to memory are the number of LTSF queues, the state saving period and the GVT period. For this analysis, the Average Peak Memory will be defined as:

$$AveragePeakMemory = \frac{\sum_{n=0}^{N-1} mem[n]}{N}$$

where $mem[n]$ is the amount memory allocated just after the n^{th} GVT calculation but before fossil collection and N is the total number of GVT calculations.

9.4.1 Number of LTSF Queues

Increasing the number of LTSF Queues can speed up the simulation significantly by removing the necessity to be shared among worker threads. However, by increasng the number of LTSF queues, more memory is required. The plot in figure 9.4 illustrates this by showing the percent increase in average peak memory. For all data points, the LTSF queues were shared among 10 worker threads. The memory measurements are taken from the actual memory used by a WARPED2 process, and not the memory allocated by the OS for the process.

The reason for this increase in memory as more LTSF queues are used is due to spread out of the critical path of the simulation which increases the odds that events will be received out of order and cause a rollback. Because of this, the GVT will progress more slowly and more previous states and events will have to be saved.

9.4.2 State Saving Period

The state saving period is just as important for reducing memory requirements as it is for speeding up event processing. A larger state saving period means that fewer states must remain in memory but also means that more processed events must be saved. Therefore the amount of memory required will depend on the relative sizes of states and events. The percent memory reduction with an increasing state saving period is shown in

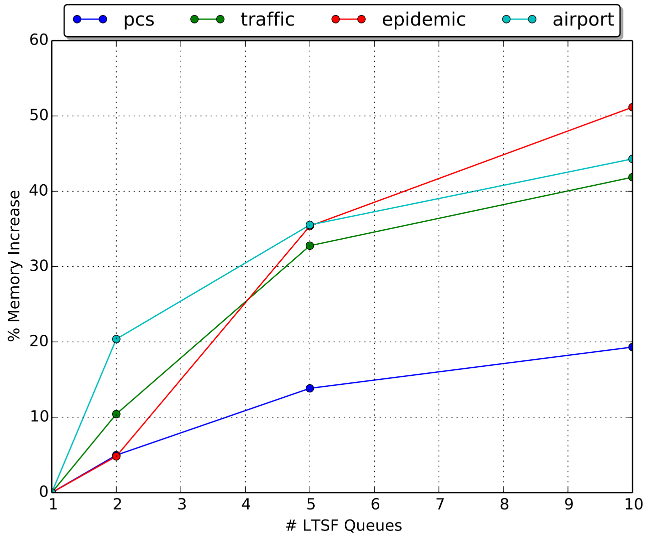
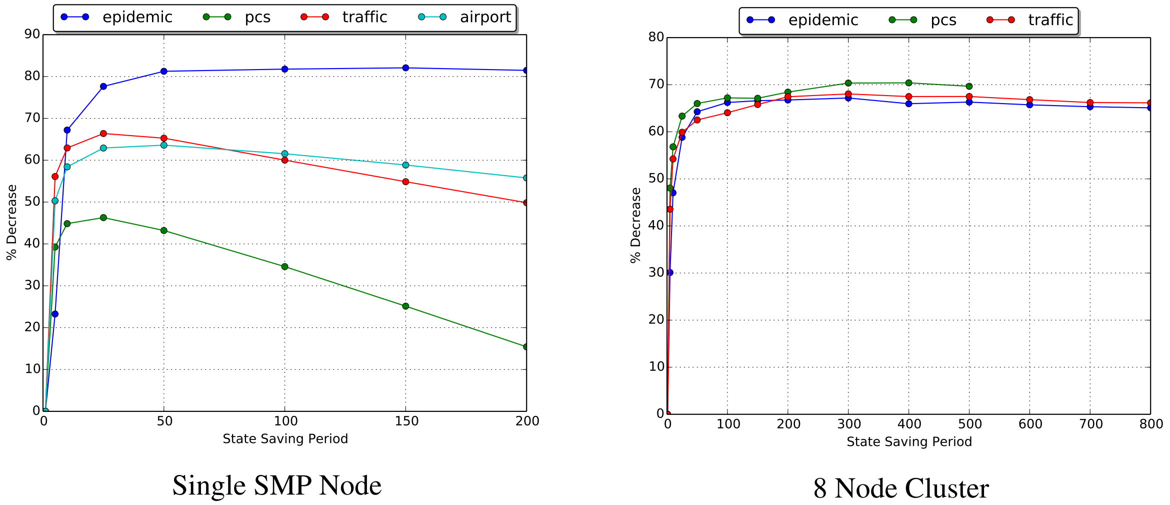


Figure 9.4: LTSF Queues: % Increase in Average Peak Memory

figure 9.5 for various simulation models.

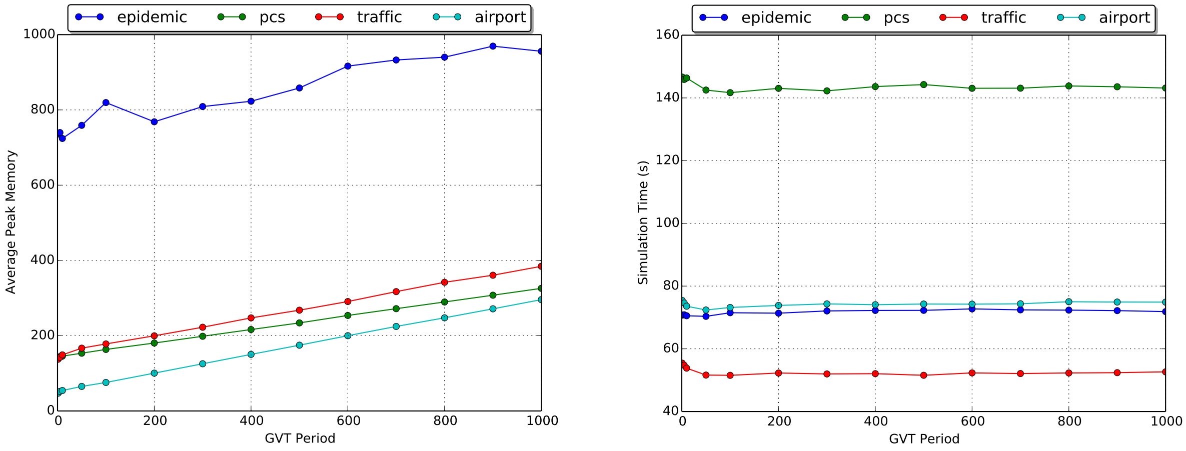
The best state saving period for memory reduction will be different for all models because it will depend on the relative size of the events and states. The period for the PCS, Traffic, and Airport models appears to be the best around 25 but for the Epidemic model which has a large state to event ratio will be much larger.



Single SMP Node

8 Node Cluster

Figure 9.5: Periodic State Saving: % Decrease in Average Peak Memory



Memory (MB)

Simulation Time (s)

Figure 9.6: GVT Period: Memory and Performance Comparison

9.4.3 GVT Period

The GVT period is a time value given in milliseconds which dictates how often to start a new GVT calculation. A smaller GVT period means that fossil collection will be more efficient and the memory consumption will stay lower. However, the GVT period is only a lower bound since a new GVT calculation cannot be started while a previous one is still in progress. This has to be taken into account especially when scaling up to a larger number of processes since a token must be circulated to all processes at least least twice.

Figure 9.6 shows the effect of an increasing GVT interval on average peak memory and performance with fossil collection by the worker threads, and the same configurations as the previous experiments. The peak memory is the point in time right after GVT is calculated since fossil collection is initiated by a GVT calculation. As expected, the average peak memory increases with an increased GVT period for all models. The performance of the simulations, on the other hand, is not affected by the GVT period at all except for extremely low values at 1ms which is as small as it can be and probably smaller than it should ever need to be used. Even with an extremely small period, the performance is only worsened by a very small amount.

Chapter 10

Conclusions and Suggestions for Future Research

10.1 Summary of Findings

To better allow the system to scale to a larger number of nodes, a seperate manager thread which all remote communication is funnelled to has been shown to be the best communication model. The reason is that the high contention for resources within the MPI library seem to be much more of a problem than a small queueing delay. In general, though, avoiding remote communication at all costs by partitioning effectively and aggregating messages reduces rollbacks and increases performance.

For a single SMP node, increasing the number of LTSF queues to the same as the number of worker threads so that there is no sharing between the LTSF queues shows the best performance. Furthermore, partitioning the LPs to the LTSF queues using a block partitioning scheme seems to work the best for all models that were tested. Other compile time optimizations to further increase performance such as replacing the default ptmalloc2 allocator with tcmalloc or jemalloc and replacing blocking mutexes with spinlocks has also been proven to effective.

10.2 Detailed Conclusions

The limitations of WARPED2 all seem to be centered around the necessity of synchronization and communication. For message passing between nodes in a cluster, any optimization that reduces remote communication will usually give optimistic results and for a shared memory communication between worker threads, any optimization that reduces synchronization of shared data structures is usually better. The two are connected, however, when it comes to synchronizing access to the message passing library. We found that thread synchronization is more costly than a small increase in remote communication caused by extra rollbacks.

For a single SMP node, increasing the number of LTSF queues, LTSF queue partitioning, replacing blocking mutexes with spinlocks, and replacing the default allocator all work to minimize the costs associated with thread synchronization and all have been shown to increase performance to some extent. For a cluster, reducing communication overheads is achieved by partitioning and aggregating messages by decreasing remote remote communication. For models that partition very well, message aggregation has no significant impact on performance.

There is also a tradeoff between scalability and performance when it comes to fossil collection. For better scalability, a fossil collection method which is carried out by the worker threads is more suitable so that remote communication can be completed with minimal queueing delay by the manager thread. Although this method slows down the maximum event rate for a single node, it will actually allow a larger event rate to be achieved by scaling up.

10.3 Suggestions for Future Work

It is still unknown what will be observed as the simulations are scaled up to larger multi-core processors and/or larger clusters and whether a strong scaling or a weak scaling approach should be taken. A strong scaling approach will increase remote communication but it will decrease the amount of time taken to fossil collect essentially pushing more work to the manager thread. On the other hand, a weak scaling approach may end up pushing the bounds on memory requirements due to a diminishing GVT granularity and could possibly lead to imbalances which will force us to rely more on better partitioning and load balancing.

As the number of threads increases, memory allocations and deallocation occur more frequently and it becomes harder to manage memory which tends to lead to increased memory fragmentation and contention. The increased memory fragmentation leads to more frequent memory allocation from the OS which could also lead to more imbalance.

As the number of nodes is increased, the harder it becomes to calculate a good GVT estimate efficiently. With an asynchronous GVT algorithm, the time taken to calculate a GVT estimate increases and becomes increasingly less accurate but with a synchronous GVT algorithm, the necessity to stop all event processing kills performance. An alternative fossil collection scheme that does not rely on GVT such as optimistic fossil collection could be worth pursuing in the future.

Appendix A

x86 SMP Nodes and Cluster

		Intel® Xeon® X5675	Intel® Xeon® E5410
Processor	ISA	x86_64	x86_64
	# Cores	6	8
	# Threads	12	8
	# Sockets	1	2
	Frequency	3.06 GHz	2.33 GHz
	L1 Data Cache	32kB	32kB
	L1 Inst Cache	32kB	32kB
	L2 Cache	256kB	6MB
	L3 Cache	12MB	N/A
Runtime	OS Kernel	Linux 3.16.0-4-amd64	3.5.0-41-generic
	C Library	Debian GLIBC 2.19-18+deb8u1	Ubuntu EGLIBC 2.15-0ubuntu20.1
	Compiler	GCC v4.9.2	GCC v4.8.5
	MPI	MPICH v3.1	MPICH v1.5
	Malloc	TCMalloc v2.4	TCMalloc v2.4

Table A.1: x86 Assessment Platforms

Appendix B

Simulation Model Configurations

For all experimental analysis in the following sections, the simulation models will be configured with the parameters shown in figures B.1 and B.2 unless noted otherwise.

Model	Parameter	Value
PCS	# Cells X	100
	# Cells Y	100
	Max Channels per Cell	15
	Mean Call Interval	200
	Mean Call Duration	50
	Mean Move Interval	100
	# Portables per Cell	50
Traffic	# Intersections X	100
	# Intersections Y	100
	# Cars per Intersection	25
	Mean Interval	400
Airport	# Airports X	50
	# Airports Y	50
	Mean Ground Time	50
	Mean Flight Time	200
	# Airplanes per Airport	50

Figure B.1: Model Parameters: PCS, Traffic, and Airport

Model	Parameter Type	Parameter	Value
Epidemic	Diffusion	K	8
		β	0.1
	Disease	Transmissibility	0.12
		Latent Dwell Time	200
		Latent Infectivity	0
		Incubating Dwell Time	100
		Incubating Infectivity	0.3
		Infectious Dwell Time	400
		Infectious Infectivity	1.0
		Asympt Dwell Time	200
		Asympt Infectivity	0.5
		Prob ULU	0.2
		Prob ULV	0.9
		Prob URV	0.5
		Prob UIV	0.1
		Prob UIU	0.3
		Location Refresh Interval	50
	Region	# Regions	1000
		Min Locations per Region	10
		Max Locations per Region	10
	Location	Min Persons per Location	100
		Max Persons per Location	100
		Min Travel Time to Hub	50
		Max Travel Time to Hub	400
		Min Loc Diffusion Interval	200
		Max Loc Diffusion Interval	500

Figure B.2: Model Parameters: Epidemic

Bibliography

[1] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette, "GTW: a Time Warp system for shared memory multiprocessors," in *Proceedings of the 1994 Winter Simulation Conference* (J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, eds.), pp. 1332–1339, Dec. 1994.

[2] R. M. Fujimoto and M. Hybinette, "Computing global virtual time in shared-memory multiprocessors," Aug. 1994.

[3] Wikipedia, "Ticket lock — wikipedia, the free encyclopedia," 2015. [Online; accessed 5-October-2015].

[4] D. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 405–425, July 1985.

[5] R. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, pp. 30–53, Oct. 1990.

[6] M. P. Forum, "Mpi: A message-passing interface standard," tech. rep., Knoxville, TN, USA, 1994.

[7] H. Avril and C. Tropper, "Clustered time warp and logic simulation," in *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pp. 112–119, 1995.

[8] C. D. Carothers, D. Bauer, and S. Pearce, "Ross: A high-performance, low memory, modular time warp system," *Journal of Parallel and Distributed Computing*, pp. 53–60, 2000.

[9] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto, “Efficient optimistic parallel simulations using reverse computation,” in *Proceedings of the thirteenth workshop on Parallel and distributed simulation, PADS’99*, pp. 126–135, May 1999.

[10] A. Pellegrini, R. Vitali, and F. Quaglia, “The rome optimistic simulator: Core internals and programming model,” in *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools ’11, (ICST, Brussels, Belgium, Belgium), pp. 96–98, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011.

[11] D. Jagtap, N. Abu-Ghazaleh, and D. Ponomarev, “Optimization of parallel discrete event simulator for multi-core systems,” in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 520–531, May 2012.

[12] cppreference, “Pseudo-random number generation,” 2011.

[13] Y.-B. Lin and P. Fishwick, “Asynchronous parallel discrete event simulation,” *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, vol. 26, pp. 397–412, Jul 1996.

[14] K. S. Perumalla and S. K. Seal, “Discrete event modeling and massively parallel execution of epidemic outbreak phenomena,” *Simulation*, vol. 88, pp. 768–783, July 2012.

[15] C. Barrett, K. Bisset, S. Eubank, X. Feng, and M. Marathe, “Episimdemics: An efficient algorithm for simulating the spread of infectious disease over large realistic social networks,” in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pp. 1–12, Nov 2008.

[16] B. D. Lubachevsky *et al.*, “Rollback sometimes works...if filtered,” in *Winter Simulation Conference*, pp. 630–639, Society for Computer Simulation, Dec. 1989.

[17] R. Rönngren and M. Liljenstam, “On event ordering in parallel discrete event simulation,” in *Proceedings of the Thirteenth Workshop on Parallel and Distributed Simulation*, PADS ’99, (Washington, DC, USA), pp. 38–45, IEEE Computer Society, 1999.

[18] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of ACM*, vol. 21, pp. 558–565, July 1978.

[19] Lockless Inc., “Spinlocks and read-write locks,” 2010.

[20] K. M. Chandy and L. Lamport, “Distributed snapshots: Determining global states of distributed systems,” *ACM Transactions on Computer Systems*, vol. 3, pp. 63–75, Feb. 1985.

[21] T. Lai and J. Yang, “On distributed snapshots,” *Information Processing Letters*, vol. 25, pp. 153–158, May 1987.

[22] F. Mattern, “Efficient algorithms for distributed snapshots and global virtual time approximation,” *Journal of Parallel and Distributed Computing*, vol. 18, pp. 423–434, Aug. 1993.

[23] B. Samadi, *Distributed Simulation, Algorithms and Performance Analysis*. PhD thesis, Computer Science Department, University of California, Los Angeles, CA, 1985.

[24] D. Bauer, G. Yaun, C. D. Carothers, M. Yuksel, and S. Kalyanaraman, “Seven-oclock: A new distributed gvt algorithm using network atomic operations,” in *In Proceedings of the Workshop on Parallel and Distributed Simulation (PADS) 05*, pp. 39–48, IEEE Computer Society, 2005.

[25] A. O. Holder and C. Carothers, “Analysis of time warp on a 32,768 processor ibm blue gene/l supercomputer,” in *Proceedings of the 2008 European Modeling and Simulation Symposium (EMSS ’08)*, 2008.