I, Karthikeyan Muthalagu, hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Engineering.

It is entitled: Threaed WARPED: An Optimistic Parallel Discrete Event Simulator for Cluster of Multi-Core Machines

Student's name: Karthikeyan Muthalagu

This work and its defense approved by:

Committee chair: Philip Wilsey, PhD

Committee member: Fred Beyette, PhD

Committee member: Wen Ben Jone, PhD
Threaded WARPED: An Optimistic Parallel Discrete Event Simulator for Cluster of Multi-Core Machines

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

November 13, 2012

by

Karthikeyan Muthalagu
BEMECT, Thiagarajar College of Engineering, 2006

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

Parallel Discrete Event Simulation (PDES) is an active area of research for many years. Studies with PDES have focused primarily on either shared memory or distributed memory platforms. However, the emergence of low-cost multi-core and many-core processors suitable for use in Beowulf clusters provides an opportunity for PDES execution on a platform containing both shared memory and distributed memory parallelism. This thesis explores the migration of an existing PDES simulation kernel called WARPED to a Beowulf Cluster of many-core processors. More precisely, WARPED is an optimistically synchronized PDES simulation kernel that implements the Time Warp paradigm. It was originally designed for efficient execution on single-core Beowulf Clusters. The work of this thesis extends the WARPED kernel to contain parallel threaded execution on each node as well as parallelism between the nodes of the cluster. The new version of warped will be called threaded WARPED.

In this thesis, warped is redesigned with thread safe data structure protected by various constructs. In particular atomic instructions are used to deploy lock-free data-structures and synchronization. With the addition of thread to WARPED the work also required adjustments and extensions to several of the sub-algorithms of Time Warp. In particular, adjustments to the algorithm for computing Global Virtual Time (GVT), and termination detection were required. This thesis explains the modifications made to implement threaded WARPED and evaluates the performance capabilities of the two solutions for managing the shared data structures.
Acknowledgments

I would like to thank Dr. Philip A. Wilsey for inspiring me to pursue research in parallel simulation, and for the material and the guidance he has provided throughout my thesis. I thank Sandeep Chippada and Sanchit Saxena for their valuable input in system design and testing. I thank Vignesh Subbian and Ryan Beeler for helping me in writing this document. I would like to thank Rob Lancaster and Ryan Child for their help, and my five wonderful roommates for making my graduate life cherishable.

I thank my friend Vignesh Jothivel for his immense support throughout my life. I would like to dedicate this work to my family.
Contents

1 Introduction 1
   1.1 Research Statement ........................................ 2
   1.2 Thesis Overview ........................................... 2

2 Background 4
   2.1 Discrete Event Simulation .................................. 4
   2.2 Parallel Discrete Event Simulation ......................... 5
   2.3 Time Warp Mechanism ....................................... 6
      2.3.1 Straggler And Rollback ................................. 7
      2.3.2 State Saving ........................................... 8
      2.3.3 Virtual Time ........................................... 10
      2.3.4 Fossil collection ....................................... 11
   2.4 Other Time Warp Simulators ................................ 11

3 Evolution of WARPED 13
   3.1 Introduction ............................................ 13
   3.2 Warped v1.x ............................................ 13
      3.2.1 Modular Design ....................................... 13
      3.2.2 APIs ................................................ 14
      3.2.3 Limitations .......................................... 14
   3.3 WARPED v2.x ............................................ 16
      3.3.1 Limitations .......................................... 16
# Contents

3.4  WARPED v3.x ................................. 17

4  WARPED Overview ......................... 18

4.1  Introduction ............................... 18

4.2  WARPED Kernel Interface ................. 18

4.2.1 Simulation Manager ..................... 19

4.2.2 Event List Management ................. 20

4.2.3 Output Manager ......................... 21

4.2.4 Anti-Message Transmission ............. 21

4.2.5 State Manager ......................... 22

4.2.6 Communication Manager ............... 23

4.2.7 Termination Manager ................... 24

4.2.8 GVT Manager .......................... 27

4.2.9 Mattern’s GVT Estimation Algorithm ... 27

4.3  Preparing WARPED for Multi-core Cluster 29

4.3.1 First Threaded version ................. 30

5  Threaded WARPED ......................... 31

5.1  Introduction ............................. 31

5.2  Design Overview ......................... 31

5.3  Implementation Details .................. 32

5.3.1 POSIX Threads ......................... 32

5.4  Threaded Data Structures ............... 34

5.4.1 LTSF Queue ........................... 34

5.4.2 Threaded Event Set ..................... 35

5.4.3 Output Queue .......................... 36

5.4.4 Straggler Messages .................... 37

5.4.5 Anti-Messages ......................... 38

5.4.6 Send Buffer Object .................... 40
5.4.7 Threaded State Manager ................................................................. 40
5.5 Simulation Engine ............................................................................ 42
5.5.1 The Master Thread ........................................................................ 42
5.5.2 Worker Threads ............................................................................ 44
5.6 GVT Estimation ................................................................................ 46

6 Performance Analysis ..................................................................... 49
6.1 Simulation Models .......................................................................... 49
6.1.1 Pingpong ..................................................................................... 49
6.1.2 RAID .......................................................................................... 51
6.1.3 SMMP ........................................................................................ 53
6.1.4 Phold .......................................................................................... 54
6.2 Experimental Setup ........................................................................ 54
6.2.1 Model Configurations .................................................................. 55
6.3 Experiments ..................................................................................... 56
6.4 On Beowulf Cluster ......................................................................... 56
6.4.1 Simulation Time Vs Worker Thread Count ................................. 56
6.4.2 TimeWarp Vs ThreadedTimeWarp .............................................. 56
6.4.3 Performance of Threaded GVT Manager .................................... 58
6.5 On Magny-Cours Machine ............................................................. 61
6.5.1 Simulation Time Vs Thread Count .............................................. 61
6.5.2 Heavy weight Vs Threaded Warped .......................................... 62
6.6 Summary ........................................................................................ 63

7 Conclusions & Future Research ...................................................... 64
7.1 Detailed Conclusions ..................................................................... 64
7.2 Suggestions for Future Work ......................................................... 64
7.2.1 Alternate Multi-Threaded Design .............................................. 65
7.2.2 Configurable LTSF Queue .......................................................... 65
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2.3</td>
<td>Dynamic Partitioning / Load Balancing</td>
<td>65</td>
</tr>
<tr>
<td>7.2.4</td>
<td>Real Simulation Models</td>
<td>66</td>
</tr>
<tr>
<td>7.2.5</td>
<td>Atomic Data Structures</td>
<td>66</td>
</tr>
</tbody>
</table>

A WARPED Configuration File 67
List of Figures

2.1 Mapping Physical Process to Logical Process . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2 Data Structures Used in PDES . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.3 Straggler arriving at an LP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.4 Unprocessed and Processed Queues of the LP . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.5 After Processing a Straggler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.6 Rollback and CoastForwarding in an LP . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

3.1 Logical Process in WARPed with N Simulation objects . . . . . . . . . . . . . . . . . . . . 14
3.2 Execution Flow of WARPed v1.x and v2.x . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

4.1 Components Of WARPed Running in a Cluster . . . . . . . . . . . . . . . . . . . . . . . . . 19
4.2 Negative Events aggregated as Negative Message . . . . . . . . . . . . . . . . . . . . . . . 22
4.3 Token Passing Scenario: LP3 acts as a terminator . . . . . . . . . . . . . . . . . . . . . . . 25
4.4 Token Passing Scenario: LP1 acts as a terminator . . . . . . . . . . . . . . . . . . . . . . . 26
4.5 Mattern’s GVT Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
4.6 Execution Flow of Mattern’s GVT Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . 28
4.7 Execution Flow of WARPed v2.x . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
4.8 Components of WARPed Running in a Cluster . . . . . . . . . . . . . . . . . . . . . . . . . 30

5.1 Atomic Locks and its Interface in WARPed . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
5.2 Schedule Queue of WARPed v3.x . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
5.3 Threaded Event Set With Atomic Locks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
<table>
<thead>
<tr>
<th>FIGURE</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4</td>
<td>Execution Flow : Straggler handling in Threaded WARPED</td>
<td>36</td>
</tr>
<tr>
<td>5.5</td>
<td>Straggler Handling in Threaded WARPED</td>
<td>37</td>
</tr>
<tr>
<td>5.6</td>
<td>Multiple Straggler Handling in Threaded WARPED</td>
<td>38</td>
</tr>
<tr>
<td>5.7</td>
<td>Execution Flow in handling Negative Message</td>
<td>39</td>
</tr>
<tr>
<td>5.8</td>
<td>Straggler handing in Threaded WARPED</td>
<td>40</td>
</tr>
<tr>
<td>5.9</td>
<td>Solving State Saving Ambiguity in WARPED v2.x</td>
<td>41</td>
</tr>
<tr>
<td>5.10</td>
<td>Three Tuple State Saving Index in Threaded WARPED</td>
<td>42</td>
</tr>
<tr>
<td>5.11</td>
<td>Architecture of Threaded WARPED</td>
<td>42</td>
</tr>
<tr>
<td>5.12</td>
<td>Execution Flow : Master Thread</td>
<td>43</td>
</tr>
<tr>
<td>5.13</td>
<td>Input Queue Status While Rollback</td>
<td>44</td>
</tr>
<tr>
<td>5.14</td>
<td>Execution Flow of Master Thread</td>
<td>45</td>
</tr>
<tr>
<td>5.15</td>
<td>LVT Computation In Threaded WARPED</td>
<td>46</td>
</tr>
<tr>
<td>5.16</td>
<td>Execution Flow of LVT Computation In Threaded WARPED</td>
<td>47</td>
</tr>
<tr>
<td>6.1</td>
<td>Simple PingPong Model with Four PingObjects</td>
<td>50</td>
</tr>
<tr>
<td>6.2</td>
<td>Simple RAID5 model with Four Disks</td>
<td>50</td>
</tr>
<tr>
<td>6.3</td>
<td>Simple SMMP Model with Four Processors Sharing a Common Memory</td>
<td>52</td>
</tr>
<tr>
<td>6.4</td>
<td>PHOLD Model with Four Simulation Objects</td>
<td>53</td>
</tr>
<tr>
<td>6.5</td>
<td>Performance of RAID with increase in worker thread count</td>
<td>57</td>
</tr>
<tr>
<td>6.6</td>
<td>Performance of SMMP with increase in worker thread count</td>
<td>57</td>
</tr>
<tr>
<td>6.7</td>
<td>Performance of PHOLD with increase in worker thread count</td>
<td>57</td>
</tr>
<tr>
<td>6.8</td>
<td>Performance comparison between WARPED and THREADED WARPED for RAID.</td>
<td>59</td>
</tr>
<tr>
<td>6.9</td>
<td>Performance comparison between WARPED and THREADED WARPED for SMMP.</td>
<td>59</td>
</tr>
<tr>
<td>6.10</td>
<td>Performance comparison between WARPED and THREADED WARPED for PHOLD.</td>
<td>59</td>
</tr>
<tr>
<td>6.11</td>
<td>Performance of Threaded GVT Manager with RAID</td>
<td>60</td>
</tr>
<tr>
<td>6.12</td>
<td>Performance of Threaded GVT Manager with SMMP</td>
<td>60</td>
</tr>
<tr>
<td>6.13</td>
<td>Performance of Threaded GVT Manager with PHOLD</td>
<td>60</td>
</tr>
<tr>
<td>6.14</td>
<td>Performance of The Threaded kernel with different Worker Count on Magny-cours Machine.</td>
<td>61</td>
</tr>
</tbody>
</table>
6.15 Performance Comparison between WARPED and THREADED WARPED on Magny-cours Machine. 62
List of Tables

6.1 Configuration File of RAID ............................................. 51
6.2 Configuration of a node in the Beowulf Cluster ..................... 54
6.3 Configuration of the Intel i7 Machine ................................. 55
6.4 Configuration of the AMD’s Magny-cours Machine ............... 55
Chapter 1

Introduction

Simulation is a process of mimicking the operation of a system over time. It assists prototyping a new system design and helps understanding the characteristics of an existing system [1]. Systems that change their states continuously are represented by a continuous system model and the ones that change at discrete time interval are called as discrete systems. A discrete event simulation model is a representation of a real word system as series of states that change with respect to time. Based on the progression of simulation process over the time, they are classified as event driven simulation and time driven simulations [2]. The work of this thesis is concerned with simulations that are discrete and event driven. Discrete Event Simulations (DES) are used to mimic the operation of a discrete system in which time leaps through distinct intervals known as events [3]. Simulation of large scale complex systems in a sequential computing platform suffer from long execution times and small available memory constraints [4]. Researchers addressed the problem by designing algorithms for the parallel execution of DES called Parallel Discrete Event Simulation (PDES) [5].

WARPED is a parallel simulation kernel that is optimistically synchronized using the Time Warp mechanism [5]. The WARPed kernel was developed primarily for experimental research for the development and assessment of optimizations to the Time Warp mechanism and it has gone through several major versions. The initial version (v1.x) was designed with compile time bindings of configuration of the sub-algorithms of Time Warp. The second main version (v2.x) was designed to allow runtime configuration of the Time Warp algorithms [4]. In both the v1.x and v2.x series kernels, the system used heavyweight processes communicating using MPI or TCP/IP executing on a Beowulf cluster.
This research focuses mainly on creating a new version series for the WARPed kernel (v3.x) that adds threading to the system to support parallelism within each node and between the nodes of a multi-core/many-core Beowulf Cluster. This version series is also called Threaded WARPed. Threaded WARPed is designed based on master-worker paradigm of parallel programming [6] with atomic lock assisted data structures. It is integrated with the existing system with run time configuration parameters. The new threaded version of WARPed can be used as a basis for future research in the area of PDES on multi-core and many-core Beowulf Clusters.

1.1 Research Statement

The primary goal of this thesis is to effectively utilize all CPU cores available in every node of a heterogeneous cluster by combining the shared memory and distributed memory design of the Time Warp mechanism. The principle idea behind this thesis is to redesign various data structures of the warped kernel to be thread safe that allows concurrent execution of simulation objects within a single node of the cluster without blocking other simulation objects, thereby achieving more parallelism and effective utilization of CPU cores in each node of the cluster.

A previous attempt to add threads to WARPed [6] was largely unsuccessful due to a configuration of data structures and locks that resulted in excessive time lost from lock acquisition and release. The new design removes this constraint by allowing other threads to access an executing object’s data structures, and thereby reducing the wait time.

In addition to concurrent execution, the master-worker paradigm of parallel programming [6] separates event execution from housekeeping activities. By categorizing the work, WARPed removed the housekeeping and non critical functionality from the critical path and expedited the simulation. To support multi-threaded execution, the Time Warp data structures are protected with atomic instructions instead of locks to achieve concurrency.

1.2 Thesis Overview

The remainder of this thesis is organized as follows:
Chapter 2 provides a general overview of DES, PDES, the Time Warp synchronization mechanisms and several key optimizations for the Time Warp mechanisms. It also highlights several other Time Warp simulators.

Chapter 3 discusses the history of development of the WARPED simulation kernel, its advantages and limitations. It elaborates various features that were added and revised in each version.

Chapter 4 explains various components of the WARPED kernel and its data structures. It also explains some of the distributed algorithms used in WARPED.

Chapter 5 presents a detailed design of Threaded WARPED. It explains the atomic instructions and locking mechanism in contrast to its alternatives. This chapter lists different algorithms, modules and data structures used along with appropriate reasoning and background for choosing it. It also describes scenarios which are unique to threaded WARPED and the strategies used to handle them.

Chapter 6 gives an overview of simulation models and experimental setup used for performance analysis. It analyzes various data and results taken from experiments. Finally it attempts to explain the results with sufficient substantiation.

Finally, Chapter 7 projects the bigger picture of the result and describes the accomplishments of this thesis and possible future research work that could be pursued to improve the system performance.
Chapter 2

Background

2.1 Discrete Event Simulation

Discrete Event Simulation (DES) is an active research area for many years [7]. Revolutionary advancement in computer hardware and software transformed simulation into a problem solving technique [7]. DES mimics the operation of a discrete system in which time jumps through distinct intervals. It is also used to model physical systems that are considered as a collection of processes. Each process representing the physical system consumes and produces events, which takes place at a discrete time.

Modelling is a method of mapping physical processes (PP) of a real world system into logical processes (LP) of a simulation system [8]. As shown in the Figure 2.1, each AND gate of the physical processes is mapped into logical process. LPs interacts with each other through a time-stamped entity called events. In the Figure 2.1, events are nothing but the signals flowing between the gates. The current input and output of an LP determines the state of the LP at the given discrete time. When an LP processes an event, it will generally update its state and generate new events. In a normal DES, LPs are executed sequentially one at a time with the arrival of events at its input queue. Simulation is said to be completed when a defined global termination condition is reached by all the LPs [9].
2.2 Parallel Discrete Event Simulation

As simulation models grow larger, sequential DES can take more time and memory to complete. In some cases, the time and space requirements are excessively large and can make sequential simulation infeasible. To solve this problem, researchers have pursued studies of PDES. In PDES, LPs are executed concurrently and synchronized through a mechanism that takes the simulation forward in a meaningful way. LPs achieve synchronization by storing essential information as different data structures. Figure 2.2 shows data structures used in PDES. Three essential components of any DES are:

- Event List - list of events that belongs to an LP.
- State Queue - list of saved states of an LP.
- Global clock - a synchronization parameter to track the simulation time.
Parallel simulation can use multiple mechanisms to implement and coordinate the parallel processing within the simulation. These synchronization mechanisms can be broadly classified into *centralized* and *distributed*. In a centralized solution the decision controlling event execution is determined by a single centralized mechanism. In contrast, a distributed solution is organized so that decisions about event execution are made at many locations throughout the parallel solution. In either case, the challenge is to obtain maximized parallelism of the simulation that also satisfies the casual ordering of event executions throughout the entire simulation. The class of distributed synchronization algorithms can be further divided into two categories namely: consecutive and optimistic.

- The *conservative* method sequences event execution across LPs in strict increasing timestamp order [10]. Conservative PDES is a method of avoiding the causal order violation by designing appropriate lookahead mechanism. Lookahead method associated with LPs determine whether it is safe to execute an event from event list with a particular timestamp. The most commonly used conservative synchronization method is known as Chandy, Misra, and Bryant (CMB) algorithm [10] [11].

- In contrast to conservative method, where the casual ordering of events is strictly enforced, *optimistic* methods do not always strictly enforce casual order of events. The most widely optimistic method is the Time Warp mechanism [12–14].

This thesis is concerned with adding threads to the warped simulation kernel, which uses the Time Warp mechanism [15–17].

### 2.3 Time Warp Mechanism

In Time Warp based PDES, each LP has an input queue of sorted events that are executed without considering the events in input queue of other LPs [18]. Since events are processed without explicit synchronization with other concurrently executing LPs, the possibility that the LP could prematurely process events at timestamps greater than an event sent from a remote LP. Such a remote event with lesser timestamp is called as a *Straggler Event* [18].
2.3.1 Straggler And Rollback

The arrival of a straggler causes an LP to reset itself to an earlier time that is needed to process the straggler event in its proper timestamp order. This process of an LP returning to a past state is called Rollback [18]. Rolling back an LP involves restoring all the data structures associated with that LP to the time of rollback. Rollback process affects the state of local LP and the prematurely sent events to other LPs. In Time Warp PDES, sent events of each LP are stored in a separate queue called the output queue. Handling a straggler involves three main steps:

- **Restoring the State:** The LP that receives a straggler event will restore itself to a past saved state from the state queue. Restored state corresponds to an earlier time when the straggler event can be properly processed. In Figure 2.3 the LP is restored to state D corresponding to time 45.

- **Restoring Input Queue:** The arrival of a straggler event indicates premature processing of future events in an LP. The rollback process restores the proper execution order by reprocessing those prematurely processed events. Time Warp stores processed events and unprocessed events in separate queues as shown in Figure 2.4. This helps in isolating and restoring premature events back to the unprocessed queue.
In the example shown in Figure 2.3, a straggler event with a timestamp of 50 arrives at an LP that has processed events to a local time of 110. The LP rollbacks the input queue by moving all the events with timestamp greater than or equal to straggler (50) for reprocessing.

- Sending anti-messages: Time Warp nullifies the prematurely sent output events by sending a special type of messages called anti-messages with the same timestamp as the prematurely sent messages. An LP generates these anti-messages by maintaining a list of sent events called the output queue. Output events with timestamp greater than the straggler are sent as anti-messages. In Figure 2.3, anti-messages are sent for events from 53 to 115.

Anti-messages also called Negative messages cancels or removes the equivalent prematurely sent positive events from the unprocessed queue in the receiving LP.

### 2.3.2 State Saving

The LP recover from casuality error by saving its state. State saving is to save the state of an LP every time it changes in response to the processing of an event. The rollback process restores the state of the LP when it receives straggler message. Frequent state saving would result in large memory usage. To reduce the memory, an LP can save states periodically (Periodic State Saving) after it has processed a specified number of events. The number of processed events between state saves is called the checkpoint interval.

Instead of static determination of a checkpoint interval, a simulation can dynamically (Dynamic State Saving) adjust it for each LP. To determine the optimal checkpoint interval for time performance, the costs...
of saving states needs to be balanced. Algorithms for dynamically adjusting the checkpoint interval was developed by Lin [19], Palaniswamy [20], and Ronngren [21]. These algorithms uses an analytical model that derives the formula used for determining optimal checkpoint intervals. These formulas are fairly complex and involves a significant amount of overhead. Another algorithm was developed by Fleischmann and Wilsey where the calculation is simple [22].

In periodic and adaptive state saving, the kernel does not save all states of the objects. Hence, sometimes the rollback process might not restore the LP with the ideal state from which it should reprocess the events. In such cases, the output manager restores the LP to the next available saved state. The LP executes events till it reaches the ideal state. This processing is called coast forwarding. An LP does not send any output events during the coast forward phase; but it updates its state. WARPED calls the timestamp of restored state as restoredTime and the straggler’s time as rollbackTime.

WARPED executes events between restoredTime and rollbackTime without sending the output events to other LPs. LPs normally coast forward from restoreTime to RollbackTime.

In the example 2.6, when processing the straggler 50, the LP restores the object to state 25 and moves events greater than 25 to unprocessed queue. Finally, LP coast forwards events from 25 to 50.
2.3.3 Virtual Time

Virtual Time (VT) is a paradigm introduced by Jefferson for organising and synchronizing distributed systems [23]. Time Warp adapts Virtual Time to synchronize all LPs involved in the simulation. VT provides an abstraction for real time [23] and hence used as a reference to compare entities across all LPs. In Time Warp, every LP has a simulation VT called Local Virtual Time (LVT). LVT of an LP is the timestamp of the last executed event.

Global Virtual Time (GVT) of a simulator at a given time is the minimum of LVT’s of all LPs involved in the simulation as well as the minimum timestamp of all unprocessed event messages [23]. Many shared and distributed memory algorithms are proposed for computing GVT [24] [25] [12]. Mattern’s algorithm is one of the popular distributed GVT algorithms [9]. GVT calculation is not in the critical path of the simulation, yet it plays a major role in assisting housekeeping activities of the distributed simulation [9].

Figure 2.6: Rollback and CoastForwarding in an LP
2.3.4 Fossil collection

Time Warp mechanism achieves optimistic synchronization of LPs by saving sufficient information in the input, state, and output queues. Complex simulation models with larger LP count increase the memory requirement of the simulation. Time Warp satisfies this requirement by reclaiming the events and states that are not needed. This process of periodically reclaiming memory is called Fossil Collection [26].

GVT is the lowest timestamp of all unprocessed events of all LPs in a simulation [26]. GVT is used as a reference for fossil collection. States and events with timestamp less than GVT can be safely reclaimed [26]. Reclaiming an event or state of timestamp greater than GVT creates irrecoverable damage to the simulation. Many algorithms are proposed over the years for fossil collection [27] [28] [29] [30].

2.4 Other Time Warp Simulators

Several research groups are working on developing parallel simulators and optimizations for Time Warp mechanism. Simulation development evolves along with computing platforms [7]. Researchers are working on building parallel simulator that can utilise multi-core, many-core machines and GPGPUs [31]. This section provides a brief overview of some other contemporary parallel simulators and their key features.

- GTW, Georgia Tech Time Warp: GTW is a Time Warp based optimistic PDES designed and developed for shared memory multi processor computing environment by researchers at Georgia Tech. GTW’s main design principle is to exploit the cache coherency of a multi processor platform [12] with minimum Time Warp overhead. Similar to WARPED, GTW is a collection of LPs communicating to each other through messages using shared buffers. Each LP has data structures like event list and state queues that are protected by locks with a unique owner at any time of execution [12]. A non-blocking GVT algorithm is used in GTW in which event execution is interleaved while computing GVT in an LP.

- ROSS, Rensselaer Optimistic Simulation System: It is also a Time Warp optimistic PDES developed at Rensselaer Polytechnic Institute. ROSS project was an initiative after successful results of GTW over sequential simulators in terms of memory and speed. ROSS is mainly designed for executing massive simulation models with millions of simulation objects [13]. ROSS uses reverse computation
instead of state saving to do rollback of LP. Reverse computation enables ROSS to save memory by
avoiding state saving at the cost of reverse computation overhead. ROSS uses MPI for communica-
tion between LPs. The latest version, called Threaded ROSS added threading capability to ROSS.
Threaded ROSS uses pthread library to parallelize execution of events in an LP [32]. This version
uses shared memory for thread communication and eradicates message passing latency between LPs.
Shared memory is protected with condition variables and mutex between threads.

• CTW, Clustered Time Warp: CTW is a parallel digital logic simulator that uses both sequential sim-
ulation (within a node) and optimistic Time Warp principle (between the cluster) [14]. Simulating
thousands of gates as simulation objects makes rollback frequent and memory management complex.
CTW forms group of gates into a cluster and assigns those clusters to different nodes of the computing
platform. Each node does a sequential simulation of a functional unit. Relatively independent func-
tional units are simulated in different nodes that progress in its own timestamp. Each node checkpoints
its state periodically to assist the synchronization process. Straggler Event in a cluster can cause two
types of rollbacks.

  – All LPs in the cluster rollback, called clustered rollback.

  – Rollback only the LP that received the straggler, called local rollback.
Chapter 3

Evolution of WARPED

3.1 Introduction

In 1995, researchers from the University of Cincinnati were working on projects Quest and Vast to build a simulation engine to run large VHDL models. A simulation kernel (WARPED) was developed as a part of this engine to accommodate memory and processing power requirements that was needed to run the large VHDL models [33]. So, WARPED was developed as an optimistically synchronised simulator that could run on a network of heterogeneous workstations (UNIX Machines). The first version of the kernel is called WARPED v1.x series.

3.2 Warped v1.x

WARPED acted as a generic Time Warp simulation kernel to test and compare new optimizations such as distributed termination, optimistic fossil collection and GVT computation algorithms that were developed as a part of optimization techniques for Time Warp mechanism [2] [34] [35] [36].

3.2.1 Modular Design

WARPED is developed in object oriented C++. The object oriented design provides the necessary abstraction need for modular design [37]. Modular design of WARPED makes it easier to modify, maintain and expand [38]. It also provides flexibility to develop and plug-in new optimizations such as fossil collec-
tion, lazy cancellation, GVT computation, event list and others. This flexibility makes WARPED a powerful experimentation tool for exploring the new horizons of PDES [15] [39] [38].

3.2.2 APIs

WARPED hides the complexity of the simulation kernel from developers by providing a modelling APIs [38]. Simulation model developers extends simulation objects, events, and states to realize the model in the simulation kernel. This allows developers to model a simulation without any working knowledge of the kernel [4].

WARPED realizes the abstraction of Logical Process (LP) by using one or more simulation objects. As shown in Figure 3.1, every simulation objects has an input queue, state queue and an output queue. LPs running in different nodes of the cluster communicate through a message passing interface (MPI). Simulation objects within the same LP uses shared memory for communication. Figure 3.2 shows the execution flow.

3.2.3 Limitations

As mentioned earlier, WARPED v1.x was used to test several Time Warp optimizations [40–44]. WARPED v1.x can be configured with these optimizations only at compile time. It uses *ifs* and *defs* to include the code that handles essential optimizations and excludes code that is not needed. This forces recompilation of the code every time a change is made to the configuration. As the kernel grew the complexity and compile
CHAPTER 3. EVOLUTION OF WARPED

3.2. WARPED V1.X

Figure 3.2: Execution Flow of WARPED v1.x and v2.x
time of WARPED increased. Recompilation adds complex compiler directives that makes maintenance of WARPED v1.x difficult. These limitations led to WARPED v2.x.

3.3 WARPED v2.x

This version of the kernel addresses the problem of compile time configuration and long compilation time by adding a runtime configuration file. This addition removes the need for compiler directives. It allows modules to be loaded at runtime based on parameters specified in the configuration file. It also combines all the individual optimizations that were previously small experimental setups in to a single executable [4]. Experiments can now be performed under different configurations simply by changing the parameters in the configuration file. For example, if the event list type is Multi-Set in the configuration file, the kernel reads the configuration file and loads the Multi-Set dynamically at runtime. Figure 4.7 shows the execution flow of WARPED v2.x.

WARPED v2.x includes three types of simulations.

- **Sequential Simulation**: A single thread executes events in the entire simulation.

- **Time Warp Simulation**: A single heavy weight process executes events at every node of the cluster. Every node synchronizes with other nodes using Time Warp protocol

- **Threaded Time Warp**: It is similar to Time Warp simulation with multiple threads executing events at every node of the cluster [6]

3.3.1 Limitations

Heavy weight Time Warp Simulation was originally designed for single core Beowulf cluster. WARPED v2.x attempts to parallelize the simulation within every node of the cluster. Although this method achieves concurrent execution of objects, it locks the data structures of the executing objects, preventing other threads from accessing it. This blocking results significant in amount of CPU time being spent on waiting for acquiring locks. Also, the waiting time increases exponentially with increase in thread count. This limitation is the primary motivation for this research.
3.4 WARPED v3.x

The new threaded version of WARPED removes the waiting time by keeping the input queue, output queue and state queues of a scheduled object open, thereby allows other objects to access it. This led to major changes in event list management, straggler and negative message handling. The implementation details of WARPED v3.x’s are discussed in detail in Chapter 5.
Chapter 4

WARPED Overview

4.1 Introduction

The primary component of WARPED v2.x is the *simulation manager*. It is the key element of the system that coordinates all other modules associated with the kernel. As described below WARPED v2.x provides three types of simulation managers: Sequential, Time Warp, and Threaded Time Warp. One of these managers can be selected dynamically at runtime by choosing appropriate configuration parameters. A sample configuration file is shown in Appendix A. The kernel implements these simulation managers by inheriting from an abstract class `SimulationManager`. The WARPED kernel can be divided in to two major functional parts:

1. Set of functions associated with core simulation engine that is responsible for event list management and Time Warp synchronization. This part of the kernel is called WARPED *Kernel Interface*.

2. Set of classes and functions associated with specific simulation model that is responsible for event execution and generation. This part of the kernel is called WARPED *Application Interface*.

4.2 WARPED Kernel Interface

The kernel interface is an integral part of code that is executed in all nodes of the cluster. Besides the simulation manager, every node in the cluster also maintains an event list manager, an output manager, a
communication manager, and a GVT manager. These components work together and act as the simulation engine of WARPED. Figure 4.1 shows an abstract view of software components running an LP in WARPED.

4.2.1 Simulation Manager

The simulation manager creates, executes, and destroys events of all the simulation objects in the WARPED kernel. It initializes the simulation model and different modules of the kernel based on the application and kernel configuration parameters. It organizes and maintains events that are generated within and received from other simulation managers, and also coordinates different modules of the kernel and takes the simulation forward. The simulate() function of the manager handles the core functionality of event execution within the kernel. Every manager implements simulate() in a different way. Other important functions of the simulation manager are:

- configure(): This reads the configuration file, instantiates and allocates memory for different components such as event list, GVT Manager, Output Manager at runtime.
• `handleEventReceiver()` : This sends events to a destination simulation object either through the event list manager (local events) or through the communication manager (remote events). The simulation object calls this function every time it creates an event.

• `getEvent()` : This gets the next available event for execution from the event list. It returns the reference to the top event in the event list.

• `rollback()` : The manager calls this on receiving a straggler event. It rollbacks an object’s state queue and input queue.

• `cancelEvents()` : This receives anti-messages and cancels the corresponding positive events in the input queue.

• `fossilCollect()` : This initiates memory reclamation by calling its corresponding modules from other components such as the state manager. The kernel calls this method after computing GVT.

In the kernel version v2.x, every simulation manager class overrides and reimplements these functions. It also provides methods like `getGVTManager()` that gives handle to other components of the kernel.

### 4.2.2 Event List Management

The `EventList` data structure saves and manages unprocessed events in an increasing timestamped order. Efficiency of event management is critical for the performance of the simulator. In WARPED, the `EventSet` class abstracts the `Event List` data structures and adds methods for inserting, removing and sorting events into the list.

`SequentialSimulationManager` class instantiates an event list using either the `SinglyLinkedList` or the `SplayTree` class depending on the specified configuration. These classes implement the event list either as linked list or splay tree. The `SinglyLinkedList` class maintains a sorted list of unprocessed events. It removes events for execution at a constant time complexity as the list is sorted. Insertion complexity is propositional to the size of the list because insertion involves searching the right spot in the list. `SplayTree` is a self balancing binary search tree with an additional property that supports quick access of recently inserted elements. It has logarithmic insertion and removal complexity.

The `TimeWarpSimulationManager` supports two types of event list:
• Default Event List: It is similar to SinglyLinkedList of the Sequential Simulation Manager. It sorts the list event every time an event is inserted and removal of event takes place at constant time.

• Multi Set: It uses a self balancing tree called red-black tree, which is binary search tree that balances its height with addition and deletion of nodes. By balancing the left and right sub-tree, the red-black tree data structure performs operations such as insertion, deletion, and search in logarithmic time.

The kernel maintains either a multi set or an event list for every object in the simulation. WARPED extends the Eventset abstract class into multi set or default list based on the kernel configuration file parameters. Some important methods of EventSet are:

• peekEvent(): This function returns the event from the top of the sorted event list without actually removing it from the list.

• getEvent(): This function removes the event from top of sorted list.

• insertEvent(): This function inserts an event in to the event list.

4.2.3 Output Manager

The WARPED stores sent events (output events) for each simulation object in a unique queue called the Output Queue. While handling rollback, the kernel uses these output events to generate anti-messages. WARPED v2.x realizes this output queue using a simple vector from the Standard Template Library (STL). Since simulation objects generate events in increasing timestamp, the output queue is always sorted. The OutputManager is an abstract base class that is responsible for managing the output queue and its associated functionalities. The WARPED simulation kernel has three cancellation strategies that are implemented by classes derived from the OutputManager class. These classes are TimeWarpAggresiveOutputManager, TimeWarpLazyOutputManager, and TimeWarpDynamicOutputManager. These classes respectively implements aggressive cancellation, lazy cancellation, and dynamic cancellation.

4.2.4 Anti-Message Transmission

The OutputManager manages the transmission of anti-messages to both local and remote LPs. When multiple anti-messages are to be sent to the same LP, the Output manager methods will aggregate the anti-
messages together into a single message for transmission to the remote host. This reduces network traffic by avoiding multiple network packets routed between LPs. NegativeEvent class aggregates negative events as a message along with interface to add and remove negative events. Negative event carries a pointer to an array of negative events to the destination LP. Figure 4.2 shows how negative events are aggregated as a message.

4.2.5 State Manager

When a simulation object processes an event, it will generally update its state and generate one or more new events. It also saves its old state in a separate queue before executing new events. When an object receives a straggler, it restores itself from the saved states. The StateManager class manages these queues and functions to save and restore state using saveState() and restoreState() functions.

WARPED v2.x uses self balancing binary tree as a state queue that provides logarithmic insertion and deletion complexity. State saving involves usage of both CPU cycle and memory. WARPED has two optimizations for state saving:

- **Periodic State Saving**: This method saves the state of a simulation object only after execution of every 'n' events, where 'n' is the period of state saving. It reduces the number of saved states by a factor of 'n'. Large state period reduces the number of saved states and hence reduces the CPU cycles and memory allocated. On the other hand, fewer saved states increase the length of the rollback process by adding more events to reprocess during the coast forward period. The optimal period for state saving depends on the nature of the simulation model.
Adaptive State Saving: This is an extension of periodic state savings where each simulation object determines its checkpoint frequency based on observations of its past rollbacks behaviour. Simulation objects with higher occurrences of rollback are assigned a smaller checkpoint period and those with lower occurrences of rollback will increase (to some limit) their checkpoint interval.

4.2.6 Communication Manager

In the WARPED kernel, simulation objects communicate with each other through a Message Passing Interface (MPI). The CommunicationManager object is a collection of functions that help the LPs in sending messages and events across the nodes of the cluster. In addition to normal events and negative messages, WARPED supports GVT messages, termination messages and fossil collection messages for implementing the Time Warp synchronization. The CommunicationManager class abstracts the implementation of different communication protocol and provides a set of APIs for the simulation manager to send and receive messages. The sendMessage() function sends events to a destination object. Different modules of the kernel registers with communication manager by calling the registerWithCommunicationManager() function with the message type. WARPED supports the following type of messages:

- InitializationMessage
- StartMessage
- EventMessage
- NegativeEventMessage
- CheckIdleMessage
- AbortSimulationMessage
- GVTUpdateMessage
- RestoreCheckPointMessage

Currently, WARPED is programmed to use either of the following two protocols for sending messages:
1. TCP/IP, and


4.2.7 Termination Manager

The LPs running on one node of the cluster have no knowledge of the state of LPs running on the other nodes. This makes identification of termination in a distributed system a non-trivial problem. WARPED uses a token based asynchronous termination detection mechanism based on Mattern’s termination algorithm [9]. This algorithm uses the ”sticky flags” paradigm [9]. The algorithm is described below.

Token Passing Termination Algorithm

The termination detection algorithm circulates a token between LPs that maintains a sticky status. In WARPED, every simulation manager maintains a Simulation_Manager_Status that is Active when the manager processes events or has events to process, and Passive when there are no events to process. Termination manager module circulates tokens among other termination managers of the simulation. It also maintains a sticky Terminator_Status flag. The sticky status takes one of the following states:

- **SLAVE**: This denotes the termination manager is a slave and it cannot initiate or announce termination.
- **TERMINATOR_PROCESSING**: In this status, the LP acts as a terminator and circulates a termination token among the LPs.
- **TERMINATOR_IDLE**: Under this status, the LP still acts as a terminator, but it is not circulating termination token.

The system circulates the token in two cycles, FIRST and SECOND. After receiving the token, termination manager changes the state into one of the above three states. It decides the next status based on the current state, token value and the simulation manager status. SLAVE termination manager cannot initiate a terminate token. While beginning the simulation, WARPED initializes the master termination manager (LP) as the terminator and other LPs as slaves. It also initializes status of all simulation managers as active. An LP becomes a terminator when it changes its state from SLAVE to TERMINATOR_PROCESSING. All
non SLAVE termination managers circulates the termination token when the simulation manager’s status is passive. Termination manager relinquishes the TERMINATOR status and changes to SLAVE, when the simulation manager is active. It also taints the received token by changing its status to TAINTED. Token passing algorithm circulates the token twice to all LPs to make sure the simulation manager is idle through both passes of the token. The execution of the algorithm is as follows:

Token handling at a non SLAVE LP

1. When the token cycle = FIRST,
   - If simulation manager is passive, then create SECOND cycle and circulate the token.
   - If simulation manager is active, then change termination manager status to TERMINATOR_IDLE.

2. When the token cycle = SECOND,
   - If simulation manager is passive, then announce termination.
   - If simulation manager is active, then change termination manager status to TERMINATOR_IDLE.

3. When the token is TAINTED,
   - Change termination manager status to SLAVE and send BECOME_TERMINATOR token.
Token handling at a SLAVE LP

1. When the token cycle = FIRST or SECOND,
   If simulation manager is passive, then TAINT token back to sender.
   If simulation manager is active, then change termination manager status to TERMINATOR_IDLE.

2. When the token is BECOME_TERMINATOR,
   Change the status to TERMINATOR_IDLE.

Figure 4.3 captures a snapshot of WARPED termination manager. Here, LP3 acts as the terminator and passes FIRST cycle token to LP0. LP0, LP1 and LP2 are slave termination managers. LP0 is passive, LP1 and LP2 are actively executing events. In this scenario only LP3 can initiate and announce termination.

Figure 4.4 shows status of LPs just before deciding termination. All LPs are passive and the system is circulating the SECOND_CYCLE token among the LPs. LP1 is the terminator now. If this token circulates and returns back to LP1, it will announce the global termination to all the LPs.
4.2.8 GVT Manager

GVT acts as a synchronization parameter between LPs. It also helps fossil collection by providing a safe timestamp reference for reclaiming memory of events and saved states in WARPED [45] [46]. A number of GVT estimation algorithms have been developed over the years [24, 25, 47, 48]. WARPED uses Mattern’s GVT estimation algorithm for estimating the GVT.

4.2.9 Mattern’s GVT Estimation Algorithm

Mattern GVT estimation algorithm operates by finding global consistent states of the system using two deep cuts such as those shown in the time diagram of the simulation of Figure 4.5. The time diagram represents the states of the system over a linear line of simulation time from start to the end. Mattern uses a color coding of messages to ensure that all messages in transit are received before calculating the global state (GVT) [45].

In Mattern’s algorithm, GVT Token has three values:

1. Minimum Local Virtual Time of all LPs,
2. Minimum timestamp of all red messages sent by all LPs, and
3. Total white message count in the system.

Every simulation manager maintains a white message count and the minimum timestamp of red messages. The master simulation manager initiates a GVT estimation process by circulating the GVT token to all LPs. After receiving the token, all the simulation managers update their local VT information, local white message count, and minimum red messages’ timestamp of the token. LPs reset their white message count to zero after receiving the token.

Finally, the token returns back to the master. A white message count of zero indicates that there are no messages in transit. When the white message count is zero, the LVT carried by the token becomes the GVT of the simulation. Otherwise, the system re-circulates the token with an updated white message count and red messages’ timestamp. This process will be repeated until the white message count becomes zero. Finally after computing GVT, the master sends an GVT update token to all LPs.
Figure 4.5: Mattern’s GVT Algorithm

Figure 4.6: Execution Flow of Mattern’s GVT Algorithm
4.3 Preparing WARPED for Multi-core Cluster

The primary functionality of the simulation engine is to execute events. Figure 4.7 represents the execution flow of WARPED v2.x. The kernel v2.x and its optimizations are designed for a cluster of single core computers. In every node, a single heavy weight process executes events, communicates with other nodes, and does other housekeeping functions. Recent multi-core and many-core machines support concurrent execution of two or more instruction streams. Since the design of the kernel v2.x inherently supports a single thread of execution, it fails to exploit concurrent execution in multi-core processors. There have been two efforts to add multi-threaded execution to WARPED. The first attempt is described below. The second is the topic of this thesis.
4.3.1 First Threaded version

This method spawns multiple threads and attempts to execute multiple simulation objects concurrently. It was developed as a wrapper over the heavy weight version (v2.x). It reuses many kernel data structures from the non-threaded version. It protects concurrent access of kernel data structures by locking all associated data structures of scheduled objects. Figure 4.8 shows components that are added to enable the parallel threaded execution of simulation objects.

This method attempts a top-down approach of parallelizing WARPED by reusing modules from the non-threaded version. A scheduled object locks its data structures for the entire time that event is being executed. This makes the threads to access the shared data frequently which in turn increase the contention between threads. For example, when object A sends an event to a scheduled object B, the thread executing object A waits till object B completes execution. An analysis of the system shows that the time lost in waiting to acquire locks is significant. When the kernel executes simulation objects concurrently, waiting time of the threads increase. Simulation models with more dependent objects result in significant contention for locks.

The new Threaded WARPED addresses this problem with a bottom-up design approach. The new kernel concurrently executes simulation objects without blocking the data structures from other threads. Warped 3.x achieves this strategy by building thread safe kernel data structures. Chapter 4 discusses the new design in detail.
Chapter 5

Threaded WARPED

5.1 Introduction

This chapter explains the design and implementation of the new threaded version (v3.x) of WARPED called Threaded WARPED. It starts with the functional view of threaded design and explains how multiple threads effectively replace a single heavy weight process of the kernel. It explains the implementation of threaded design, thread-safe data structures and various locking mechanisms. It also explains how straggler handling, state saving, and negative messages works in WARPED v3.x. Finally, it enumerates the challenges in realizing the threaded design and steps taken to overcome those challenges.

5.2 Design Overview

The primary goal of this research is to parallelize the execution of LPs within every single node of a cluster by spawning multiple threads. Both distributed memory and shared memory simulators are collection of LPs with standard Time Warp functions. A threaded version must merge shared memory concurrent threads of LPs with distributed memory processing of LPs on cluster of multi-core processors.

Threaded WARPED uses the master-worker paradigm of parallel programming [4] where one master coordinates 'n' worker threads to expedite the work. The master thread initializes LPs, performs housekeeping functions, handles communication, and manages worker threads. Worker threads schedule and executes events of the simulation objects. The idea behind master-worker categorization is to remove the house-
keeping functions from the critical path of the simulation. The master thread streamlines communication between LPs by reducing possible contention that occurs when multiple threads communicate to a remote LP.

The key difference from the previous attempt to parallelize WARPED is that now anytime an object is scheduled for execution, that object’s data structures are not locked. Therefore, scheduled objects can simultaneously execute events and receive messages.

The main similarity to the previous version of WARPED (v2.x) is that both implementations have a single process performing all communication, thus the communication manager can remain unchanged. Another main difference is the manager thread is responsible for creating, terminating, and resuming the worker threads. With the manager thread handling most of the Time Warp processing functions, the worker threads are free to actually get actual simulation work done.

In addition to the default kernel data structures, the threaded version has a Schedule Queue, Send Buffer, LVT Queue and Object Lock Array. These are the thread safe data structures that allow the worker threads to communicate with a scheduled object’s input queue, and allows the master thread to concurrently perform housekeeping functions such as fossil collection and GVT estimation. This strategy substantially reduces the wait time of threads.

5.3 Implementation Details

This section describes the various data structures used and how they function together in realizing the threaded design, in detail.

5.3.1 POSIX Threads

Threaded WARPED uses the Portable Operating System Interface (POSIX) thread library (pthread library). The pthreads library is the most commonly used thread library in UNIX that defines a set of APIs for creating and manipulating threads. It is also a portable standard across different operating systems. The pthreads library provides simple and clean interface for creating, managing, and destroying threads.
Atomic Locks

Mutex protects critical region of the code from concurrent execution by multiple threads. Though it avoids waiting time, acquiring and releasing a mutex involves context switch between user and kernel threads (process). Spin lock waits over the value of a variable to acquire the access to a critical region. It does not involve any context switches. In threaded design spin locks are preferred over mutex to avoid frequent context switch of threads while trying to access a protected data.

Threaded WARPED uses atomic instructions to realize spin locks. Each entity that needs to be thread safe has a lock associated with it. These locks are implemented as an integer value which reads -1 when the object is in the unlocked state. For a thread to lock an object, the thread must be able to atomically compare and swap its own thread id with this object’s lock. It should be noted that the swap part of this command does not actually imply that the workers ID is swapped with the object’s locked bit, it merely states that if the command succeeds, the workers ID is copied to the object it is locking. If the atomic compare and swap happens to fail, the reason would be the locked bit for that object is no longer -1. Using this method over a simple true/false flag ensures no thread can unlock an object unless its identification number matches the integer stored in the object’s lock.
The AtomicState class shown in 5.1 illustrates the lock functionality for WARPED. It has these following functions:

- **setLock()**: This tries to atomically replace the Lock Owner if its value is -1
- **releaseLock()**: This atomically replaces Lock Owner with -1 if the it is equal to thread id.

WARPED uses *Atomic Locks* to protect data structures such as the event set, state queue and output queue.

### 5.4 Threaded Data Structures

#### 5.4.1 LTSF Queue

In general, the Time Warp protocol can be configured so that the simulation can follow the critical path of event timestamp by scheduling the least timestamped event of the LP for execution. Threaded WARPED maintains a minimum timestamped event of every object in an LP in a separate sorted queue called Least Time Stamp First (LTSF) queue. So, at any instant of the simulation, the top of LTSF queue has the least timestamped event for that LP. The threaded kernel enforces the simulation to follow the critical path by scheduling events from the top of LTSF queue for execution. Since the worker thread schedule events from

![Figure 5.2: Schedule Queue of WARPED v3.x](image)
5.4. THREADED DATA STRUCTURES

LTSF queue, it is also called as Scheduled Queue. Every time a worker selects an event for execution, it removes the corresponding simulation object from the schedule queue. In this way, the kernel avoids multiple workers from scheduling the same simulation object. At the end of execution, the worker returns the object back to LTSF queue, and hence the object is a candidate for scheduling once again.

Since LTSF queue helps the threaded scheduling manage, the LTSF queue is the heart of the threaded WARPED simulator. The LTSF queue can have only one event from every object. So, the maximum length of LTSF queue is equal to the number of simulation objects in that LP. An atomic lock protects the LTSF queue from concurrent access by the threads.

Figure 5.2 shows a state of Schedule Queue before and after scheduling. After object 3 is scheduled, the event corresponding to object 3 is removed from the scheduler.

5.4.2 Threaded Event Set

Every simulation object maintains a separate Input queue that includes an unprocessed queue and a processed queue. In threaded WARPED, the Input queue of a scheduled object is open for other threads to insert
and delete events. For example, the master thread can access a scheduled object’s unprocessed queue for the following purpose:

- to insert a received event from a remote LP, and
- to delete events as a part of fossil collection.

Multiple worker threads and the master threads concurrently access the unprocessed queue. Therefore, the system protects the unprocessed queue of each object by an atomic lock. Similarly, a worker and master simultaneously access the processed queue of a scheduled object. Thus, processed queues are also protected with atomic locks. Figure 5.3 shows a expanded view of threaded event set.

5.4.3 Output Queue

Every simulation object maintains a separate output queue to save sent events. Only a single worker thread and the master thread can simultaneously access the output queue. Hence, output queues of all objects are protected by atomic locks.
5.4.4 Straggler Messages

In the threaded version, worker threads execute multiple objects concurrently with their input queues open. So, a scheduled object can potentially receive a straggler event while executing a normal event. The straggler can be from a worker in same LP or from a remote LP.

The threaded kernel inserts straggler events like a normal event into the input queue. Later, when the object is scheduled, the simulation manager handles the straggler event first and then executes normal events. Since the straggler is the lowest timestamped event in the input queue, no extra computation or data structure is needed. If a simulation object receives more than one straggler message, WARPED inserts all of them into the input queue. When a worker schedules the object, it picks the straggler with lowest timestamp and rollbacks the object. Since the object rollbacks to the time of the first straggler, it makes other stragglers in

![Diagram showing straggler handling in threaded WARPED](image-url)
the input queue as normal events. In this way, the threaded kernel handles stragglers without interrupting the scheduled objects.

Figure 5.5 shows a scenario, where a straggler of timestamp 33 arrives when a scheduled object is executing an event 40. In this example, the LVT of the simulation object is 40. Upon receipt of the straggler message, the master thread simply inserts the straggler with timestamp 33. When the object is scheduled again, it identifies the first event as a straggler and performs a rollback. Figure 5.4 shows a comparison of flow of execution involved in straggler handling between single and threaded WARPED.

Figure 5.6 shows a scheduled object receiving three stragglers of timestamp 14, 27, and 33, while it is executing an event of timestamp 40. The kernel inserts all stragglers into the input queue and when the simulation object is scheduled, it identifies 14 as straggler and rollbacks the simulation object. This makes all the three straggler as positive events.

### 5.4.5 Anti-Messages

Anti-message from a remote LP carries multiple negative events for one or more simulation objects in the receiving LP. In WARSED v2.x, a single process handles anti-messages. The simulation manager receives an anti-message and decodes it into individual negative events and cancels the equivalent positive events of all objects. In the threaded version, a scheduled object can potentially receive a negative message while it is executing a normal event.
The master thread breaks the negative message and groups negative events of each object into a separate event called LocalNegativeEvent. The master inserts these new LocalNegativeEvents into the corresponding object’s unprocessed queue. Objects’ sending negative messages to another local object in the same LP directly creates a LocalNegativeEvent and inserts it into the corresponding object’s unprocessed queue.

When the LocalNegativeEvent is scheduled for execution, the kernel identifies and handles the associated negative events. In case of a negative straggler event, the kernel first rollbacks the object and then handles the negative event. Figure 5.7 shows execution flow of the simulation manager while receiving and handling a negative message. In this way, threaded kernel handles negative messages without stopping the scheduled objects.

Figure 5.8 shows an input queue with negative messages. The simulation object rollback to 30 before handing the negative message. It also shows a simulation object with two normal negative messages 54 and 63 inserted to the input queue.
5.4.6 Send Buffer Object

The `SendBuffer` is a shared memory container that uses `pthread_spinlock` to avoid concurrent access. It exists to hold global messages that the worker threads want to have transmitted to the other nodes. The reason for this is so the master thread can handle all of the remote transmissions and allow the communication manager to remain simple and unchanged from version 2.x.

5.4.7 Threaded State Manager

The State queue of an object is a list of states indexed by timestamp of the event that is being executed by the object. The kernel uses saved states to restore an object to a past state. When an object executes multiple events with the same timestamp, it saves multiple states with the same timestamp. Having two or more states of same timestamp creates ambiguity while restoring the state. This problem can be addressed in two ways:

1. By saving one state per timestamp.

![Diagram](image)

Figure 5.8: Straggler handing in Threaded WARPED
2. By indexing the state queue with an additional parameter to distinguish states with same timestamp.

WARPED v2.x uses the first method i.e., saving one state per timestamp and executing all the events with the same timestamp. On receiving more events of the same timestamp, the object rollbacks to the state that timestamp and executes them together.

For example, Figure 5.9 shows an object that executes four events of the same timestamp 15, but saves just one state. Later, it receives four more events with the same timestamp 15. So, the object rollbacks to a previous state of 10 and executes all eight events (15) together. This is the reason for WARPED v2.x to rollback an object with multiple events of same timestamp.

Since objects are executed concurrently in WARPED v3.x, following the above state saving approach results in frequent rollback of simulation objects. So, threaded version uses method two i.e., by indexing states with \texttt{eventId} and \texttt{senderId} along with timestamp. Figure 5.10 shows state queue indexed with a three tuple value.
CHAPTER 5. THREADED WARPED

5.5. SIMULATION ENGINE

Figure 5.11 shows an functional view of the threaded design.

5.5.1 The Master Thread

The Master thread is the main process that starts the simulation in every node. It is identified by thread id ‘0’. It reads the configuration of the kernel and initializes various components such as event set, GVT man-

Figure 5.10: Three Tuple State Saving Index in Threaded WARPED

Figure 5.11: Architecture of Threaded WARPED
Chapter 5. ThreaDeD Warped  

5.5. Simulation Engine

Manager, and communication manager. The master initializes the simulation model by creating and allocating simulation objects as per the model configuration file. As a part of initialization, the master thread spawns worker threads based on WorkerThreadCount parameter in the configuration file. It assigns a unique identifier to each worker called threadId. There should be a minimum of one worker for the simulation to run.

After initialization, master performs a repeated set of operations until the simulation terminates. The simulate() function implements the master thread’s core functionality. It reads messages remote messages from the physical layer. It initiates and computes GVT and uses it as a reference for reclaiming events and saved states. It sends pending messages in the message buffer to remote simulation managers through the communication manager. It also manages worker threads by suspending and waking up workers based on the number of events pending. Finally, the master ends simulation by joining and terminating the worker threads. Figure 5.12 shows the execution flow of the master thread.

Figure 5.12: Execution Flow : Master Thread
5.5.2 Worker Threads

Worker threads execute events and help simulation objects within an LP to communicate with each other. Each worker removes the event from the top of the LTSF queue for execution. The simulation object picked by a worker is said to be *scheduled* by the worker for execution. The worker removes the scheduled object from the LTSF queue to avoid multiple threads scheduling the same simulation object. This strategy avoids contention for objects at the LTSF queue.

Based on the first element of the scheduled object, the worker thread performs one of the following three steps:

1. The worker thread checks and handles the input queue of the scheduled object for stragglers event. It locks the Input queue of the scheduled object during rollback because insertion of an event while coast forwarding results in loss of the event.

Figure 5.13 shows a simulation object at 41, rollbacks to 22 after receiving a straggler event 39. As a part of rollback, events 34 and 41 are moved back to the unprocessed queue. Before coast forwarding, the object receives an event 29 into the input queue. Now, when the object coast forwards from the restored time (22) to rollback time (39), it coast forwards the event 29 along with 22 and 34. To avoid
this undesirable condition, the worker locks the input queue while performing rollback.

2. If the top of the input queue has a negative event, the worker thread handles negative message by calling `handleNegativeMessageFromStraggler()` function. In case of negative straggler, the worker thread handles the straggler before handling the negative message.

3. If the worker finds a normal event at the top of the queue, it executes the event normally.

At the end of the execution, the worker thread returns the object back to the LTSF queue and calls the scheduling manager to choose the next available object. With the help of LTSF scheduler and worker threads, threaded WARPed achieves concurrent execution of simulation objects in every node of the cluster. Figure 5.14 shows the execution flow of worker thread in Threaded WARPed.

Figure 5.14: Execution Flow of Master Thread
5.6 GVT Estimation

Threaded WARPED (v3.x) uses Mattern’s algorithm to compute GVT. The main challenge in this version is to compute LVT of a simulation manager (LP). Version v2.x computes LVT directly from the local event set. Since threaded WARPED execute objects concurrently computing LVT of an LP is non-trivial. It uses an adapted version of shared memory GVT computation algorithm developed by GTW [12].

LVT is the lower bound of all unprocessed events and partially processed events of all objects in an LP. Every worker thread executing an object logs its timestamp and the master thread finds the minimum of the logged time and declares that value as the LVT. Data structures used in computation of LVT are: a global flag (LVTFlag), an array of local times (LVTArray), an array of status updates (computeLVTStatus) for each worker and a variable (LVT) to store the computed LVT.

When a simulation manager receives a GVT token, it sets the LVTFlag to the number of threads in the

![Diagram of LVT Computation In Threaded WARPED](image)

Figure 5.15: LVT Computation In Threaded WARPED

46
LP. On reading a non-zero value of LVTFlag, each worker thread logs the minimum timestamped event involved with the currently scheduled object in the LVTArray and decrements the LVTFlag atomically by one. Similarly, each worker thread picks an object and updates LVTArray and decrements the LVTFlag by one. The last thread updates LVTArray and sets the LVTFlag to zero. When LVTFlag becomes zero, the master thread computes the minimum value from the LVTArray and sets it as the new LVT of the simulation manager. The system restricts duplicate contribution of the worker threads by setting a compute status (0/1) flag. Every worker checks this status before updating the LVTArray.

In Mattern’s algorithm, when an LP receives a GVT token, it sets a flag called `GVTTokenPending`. This flag indicates that the LP is currently working on LVT computation. After computing LVT, the master updates the GVT token with new LVT and circulates the token as per Mattern’s algorithm.

![Flowchart](image)

Figure 5.16: Execution Flow of LVT Computation In Threaded `WARPED`
class handles GVT calculation in Threaded warped. The following functions are implemented in the simulation manager to assist with LVT calculation:

- **initiateLocalGVT()**: This initiates the LVT calculation by setting the LVTFlag and resetting other shared data structures.

- **updateLVTArray()**: This updates the LVTArray. It returns 0, when computation is not complete.

- **updateLVTfromArray()**: This function finds the minimum time from the LVTArray.

Figure 5.15 captures state of an LP while computing LVT. The LP has 5 worker threads with each executing a simulation object. `computeLVTStatus` array shows the computation status of each thread. Worker threads 2 and 3 have already contributed for LVT and their status is updated to 1. After worker threads 1, 4 and 5 contribute, the master thread will compute LVT from the LVTArray. Figure 5.16 shows the execution flow of master and worker threads involved in LVT computation.
Chapter 6

Performance Analysis

This chapter provides a brief overview of the four different simulation models available in WARPED. It explains the experimental setup used for analyzing the performance. It also presents a detailed analysis of the performance of Threaded WARPED and associated algorithms such as GVT calculation and periodic state saving. It also compares the performance of threaded systems with that of heavy weight WARPED system.

6.1 Simulation Models

The four simulation models available to test the kernel are: PingPong, RAID (Redundant Array of Inexpensive Disks), SMMP (Symmetric Multi-Processor) and PHOLD (Parallel HOLD).

6.1.1 Pingpong

Pingpong is a primitive simulation model developed to test the correctness of the kernel. In this model, PingObject represents players who pass a ball around in a circular pattern. An event (PingEvent) represents passing of a ball between the players. One of the PingObjects is designated as a master before the simulation starts. When the ball is circulated and returned back to the master, the master will start a new ball. This process continues until the master completes circulation of a specified number of balls.

For example, the following configuration represents a Pingpong simulation with four PingObjects passing a total of 10000 balls. The simulation has a maximum number of 1000 balls circulated at any instant of time.
Since pingpong is a primitive model with no real processing, it is not used in the performance analysis. Figure 6.1 shows a sample pingpong with four PingObjects sending balls between them.
6.1.2 RAID

A RAID5 model distributes the parity bit to all disks to mitigate the risk of parity disk failure. It has the following simulation objects:

- **RaidProcess**: This represents a process that reads and writes data into an array of disks by using read and write events.

- **RaidFork**: This represents the RAID controller that receives read and write requests from the RaidProcess object. It forwards these requests to the appropriate disk.

- **RaidDisk**: This represents an actual disk of RAID. It receives redirected requests from RaidFork and sends the response back to the source. It maintains disk block size, head position and spindle speed to imitate the operation of a modern magnetic disk.

Figure 6.2 represents a typical RAID5 configuration with two sources, two forks and fours disks. Simulation objects are connected as specified in the configuration file and simulates the specified number of read and write requests.

Table 6.1 shows the configuration file for the model shown in Figure 6.2. Four FUJITSU disks are connected to two sources using two controllers. Source 1 simulates 6005 requests and Source 2 simulates 9005 requests.
Figure 6.3: Simple SMMP Model with Four Processors Sharing a Common Memory
6.1.3 SMMP

SMMP is a synthetic model that represents a shared memory multiprocessor. It models the memory request of multiple processors. Each processor in SMMP model has a local cache. They share a common memory with other processors. Processors make memory requests that can either be satisfied by the local cache (hit) or by the shared memory (miss). The HitRatio is specified in the model configuration file. The time delay incurred in reading and writing data to cache and memory is also configurable. A default delay of 10 units for cache and 100 units for memory are used in the current setup. Figure 6.3 shows a simple SMMP model with four processors.

Every processor in SMMP is represented by a set of six simulation objects. They are SMMPSource, SMMPJoin0, SMMPQueue, SMMPPFork, SMMPServer, and SMMPJoin1 object. The shared memory is modeled using four simulation objects. They are MemJoin, MemQueue, MemServer, and MemRouter objects. Since six components of a processor represents a single physical entity in real world, they reside in the same simulation manager. Similarly all four components of the shared memory are partitioned to be in the same simulation manager. Communication between different simulation managers happens only when there is a cache miss.
6.1.4 Phold

Parallel HOLD is a variation of Hold simulation model that is used to test queue implementations in network communication systems such as routers and switches. In Warped, PHOLD is represented by a set of simulation objects passing events between them. The total number of events inside the simulation at any given time is constant. The process starts the simulation by sending same number of events. A process selects the destination of the event and the time-stamp increment by using random variables. The user can configure:

- Process state size.
- Event computational grain.
- Event process destination, and
- the random probability distribution.

Figure 6.4 shows an example of a small PHOLD configuration containing four objects connected to each other. The rollback behavior of a PHOLD simulation depends on the configuration created. The rollback count increases with increase in the degree of connections between the processes.

### 6.2 Experimental Setup

The performance of ThreadedTimewarp is analysed under three different computing platforms.

1. Beowulf Cluster is a cluster of Linux nodes with each node powered by the configuration shown in Table 6.2

<table>
<thead>
<tr>
<th>Specification</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel(R) Xeon(R) CPU E5410 @ 2.33GHz</td>
</tr>
<tr>
<td>Sockets</td>
<td>Dual</td>
</tr>
<tr>
<td>#Cores</td>
<td>4</td>
</tr>
<tr>
<td>#Threads</td>
<td>8</td>
</tr>
<tr>
<td>Memory</td>
<td>4 GB</td>
</tr>
<tr>
<td>Operating System</td>
<td>GNU/Linux</td>
</tr>
<tr>
<td>MPI</td>
<td>MPICH2-1.4</td>
</tr>
</tbody>
</table>
Table 6.3: Configuration of the Intel i7 Machine

<table>
<thead>
<tr>
<th>Specification</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel(R) Core(TM) i7 CPU 920 @ 2.67GHz</td>
</tr>
<tr>
<td>#Cores</td>
<td>4</td>
</tr>
<tr>
<td>#Threads</td>
<td>8</td>
</tr>
<tr>
<td>Memory</td>
<td>22 GB</td>
</tr>
<tr>
<td>Operating System</td>
<td>GNU/Linux</td>
</tr>
<tr>
<td>MPI</td>
<td>MPICH2-1.4</td>
</tr>
</tbody>
</table>

Table 6.4: Configuration of the AMD’s Magny-cours Machine

<table>
<thead>
<tr>
<th>Specification</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>AMD Opteron(tm) Processor 6168</td>
</tr>
<tr>
<td>Sockets</td>
<td>Quad</td>
</tr>
<tr>
<td>#Cores</td>
<td>48</td>
</tr>
<tr>
<td>#Threads</td>
<td>48</td>
</tr>
<tr>
<td>Memory</td>
<td>64 GB</td>
</tr>
<tr>
<td>Operating System</td>
<td>Ubuntu</td>
</tr>
<tr>
<td>MPI</td>
<td>MPICH2-1.3</td>
</tr>
</tbody>
</table>

2. Intel i7 Machine is a quad core machine with the configuration as shown in Table 6.3

3. AMD’s Magny-cours Machine is an 48-core experimental machine built with four 12-core processors connected together in a board. Table 6.4 shows this machine’s configuration.

6.2.1 Model Configurations

Configurations of each simulation model are:

- PHOLD model having 200 processes with each starting 20 events. Each process has a state size of 1024 bytes and performs 100 floating point divisions for each event granularity. Every process is connected to 20 other processes. PHOLD simulation is set to terminate after GVT reaches 5000.

- RAID simulation has 20 Disks that serves 5 source processors through 5 controllers. Each process generates 95000 read and write requests.

- SMMP simulation has 128 symmetric processor modules with 10000 requests, out of which 95% are satisfied in the local cache of the processor. Access time of the cache and main memory are modelled as 10 units and 100 units respectively.
6.3 Experiments

The following experiments are conducted to understand the behaviour and performance of THREADED WARPED (v3.x) system and various optimizations.

- Performance of Threaded WARPED with increase in worker thread count.
- Performance of Threaded GVT manager.
- Performance comparison between WARPED (v2.x) and THREADED WARPED (v3.x).

Simulations were repeated 10 times and average values are used for analysis.

6.4 On Beowulf Cluster

6.4.1 Simulation Time Vs Worker Thread Count

Figure 6.5, Figure 6.6 and Figure 6.7 shows that the simulation time of all three models decreases with increase in the number of worker threads. After a certain worker thread count, the simulation time gradually increases. This is due to the kernel spending more time on rollbacks and other Time Warp housekeeping functions with the increased number of worker threads. In addition, the rollback count increases with increase in worker thread count. Initially, the increase in rollback count is gradual. After a certain worker count, the rollback increases rapidly, which in turn increases the simulation time.

PHOLD (Figure 6.7) shows a different rollback behaviour because of the independent nature of the simulation objects. In PHOLD model, worker threads can execute more process simultaneously and still produces fewer rollbacks. The same simulation time and rollback pattern is observed in all simulation models running on both the cluster and i7 machine.

6.4.2 TimeWarp Vs ThreadedTimeWarp

This section compares the performance of heavy weight version WARPED v2.x and the threaded version (v3.x). Simulations were conducted on multiple cores for threaded version and multiple nodes for heavy weight version.
CHAPTER 6. PERFORMANCE ANALYSIS

6.4. ON BEOWULF CLUSTER

Figure 6.5: Performance of RAID with increase in worker thread count

Figure 6.6: Performance of SMMP with increase in worker thread count

Figure 6.7: Performance of PHOLD with increase in worker thread count
CHAPTER 6. PERFORMANCE ANALYSIS 6.4. ON BEOWULF CLUSTER

Figure 6.8(a) and Figure 6.10(a) shows that the threaded version performs better than the heavy weight version for both increasing number of nodes and number of threads. Since the threaded version has multiple workers closely following the critical path using LTSF queue, the simulation is faster than heavy weight version.

Figure 6.9(a) shows SMMP performing better with heavyweight version than the threaded version. In SMMP, simulation objects in each node has very less inherent parallelism to exploit and hence, heavy weight version of the kernel suits this model.

6.4.3 Performance of Threaded GVT Manager

This experiment is performed to understand the behavior and performance of threaded GVT manager with different GVT estimation periods. The GVT period is varied from 100 to 10000 units.

Figure 6.11, Figure 6.12 and Figure 6.13 shows increase in the simulation time with GVT period. Increasing the GVT period results in infrequent fossil collection. This leads to lengthier queues. Consequently, the performance deteriorates with part of the queues moved to the lower hierarchy of the memory.

The rollback count also increases with increase in the GVT estimation period. All three simulation models follow the same pattern with increase in GVT period.
Figure 6.8: Performance comparison between WARPED and THREADED WARPED for RAID.

Figure 6.9: Performance comparison between WARPED and THREADED WARPED for SMMP.

Figure 6.10: Performance comparison between WARPED and THREADED WARPED for PHOLD.
CHAPTER 6. PERFORMANCE ANALYSIS

6.4. ON BEOWULF CLUSTER

Figure 6.11: Performance of Threaded GVT Manager with RAID

(a) On Cluster

(b) On i7 Machine

Figure 6.12: Performance of Threaded GVT Manager with SMMP

(a) On Cluster

(b) On i7 Machine

Figure 6.13: Performance of Threaded GVT Manager with PHOLD

(a) On Cluster

(b) On i7 Machine
6.5 On Magny-Cours Machine

The main idea behind analyzing the performance of threaded warped in this machine is to test its scalability for many-core machines. Since this machine supports higher thread count, experiments are conducted with wide range of worker thread counts.

6.5.1 Simulation Time Vs Thread Count

Figure 6.14(a) and Figure 6.14(b) shows that the simulation time of RAID and SMMP initially decreases with increase in worker count. After a thread count of 12, the simulation time increases and the rollback count increases sharply.

Figure 6.14: Performance of The Threaded kernel with different Worker Count on Magny-cours Machine.
The reason for deterioration in performance with higher thread count is due to the hotspot created in the LTSF queue. Execution cycle of every worker and the master involves acquiring the LTSF queue multiple times which in turn increases the contention. This is a potential area of future research.

### 6.5.2 Heavy weight Vs Threaded Warped

In this section, the scalability of the heavy weight (v2.x) and the threaded version (v3.x) WARPED are compared. This experimentation involves creating multiple instances of heavy weight WARPED on the same Magny-cours machine while the threaded version with multiple worker threads.

Figure 6.15(b) shows that the threaded version performs better than multiple instances of heavy weight WARPED.

![Performance Comparison](image)

Figure 6.15: Performance Comparison between WARPED and THREADED WARPED on Magny-cours Machine.
version. It can be observed that the simulation time initially decreases with increase in thread count. After thread count of eight, the simulation time starts increasing in the threaded version while that of heavy weight version keeps decreasing and eventually beats the threaded version. This is because of the hotspot in the LTSF queue with higher worker count.

Figure 6.15(a) shows, in RAID, the threaded version out performs the heavy weight version for all configurations. Heavy weight’s rollback count increases exponentially with node count which in turn results in higher execution time. Since threaded version follows the critical path closely with multiple threads, it produces less rollbacks than heavy weight.

Figure 6.15(c) shows PHOLD follows the same pattern in both heavy weight and threaded version and heavy weight performs slightly better than threaded version.

### 6.6 Summary

Experiments conducted shows that THREADEDTIMEWARP utilizes all cores of an CPU effectively and completes the simulation faster with more worker threads. Threaded kernel achieves this by closely following the critical path of the simulation using LTSF scheduling. It also shows that the threaded kernel scales effectively for many core machines when compared to the heavy weight kernel. Results also shows that the performance of threaded WARPED is based on the inherent parallelism in the simulation model.
Chapter 7

Conclusions and Suggestions for Future Research

7.1 Detailed Conclusions

WARPED 3.x a distributed PDES kernel is now developed for multi-core and many-core Beowulf cluster. This thesis explored the possibilities and challenges in combining the shared and distributed PDES that led to the evolution of to THREADEDWARPED. In addition to the simulation engine, all Time Warp optimizations are also migrated to the new version of the kernel. Threaded WARPED has attempted to achieve parallelism between nodes and within every node of the multi-core Beowulf cluster. Both distributed and shared memory algorithms are working seamlessly in achieving parallelism at different levels of the kernel. Analysis shows the capability of the kernel to scale up for multi-core and many core machines.

7.2 Suggestions for Future Work

Profiling and performance analysis of threaded WARPED shows that there is a lot of scope for improvement. This section enumerates the possible future work that could be explored to improve the threaded kernel’s performance.
7.2.1 Alternate Multi-Threaded Design

In Threaded WARPed, worker threads perform similar activities and hence, symmetric. A symmetric threaded design can achieve speedup when minimum number shares limited data.

A symmetric design with more shared data result in threads frequently running into each other. This leads to more waiting time to acquire the locks. An alternate asymmetric design can be considered for future version of WARPed. Asymmetric design partitions a task in to a set of smaller independent tasks and assigns a thread of each task. In this way, shared memory can be replaced by message passing between threads. One of the most popular browsers, Chromium use a similar parallel programming strategy [49].

7.2.2 Configurable LTSF Queue

Analysis of threaded WARPed on experimental Magny-cours machine shows that the threaded kernel efficiently scales for increasing values of worker threads. After a certain worker count, the system fails to scale and there is no performance gain. Profiling shows hotspots in a LTSF queue with large number of worker threads. Since every thread accesses the LTSF queue at least once in its execution cycle, increase in workers directly impact the performance of LTSF queue. This can be rectified by creating multiple LTSF queues and assigning a group of threads to each LTSF queue. In this method, the workers poll only from the assigned LTSF queue to distribute the load and avoid hotspots. The number of LTSF queues can also be made as a configurable parameter.

7.2.3 Dynamic Partitioning / Load Balancing

The WARPed system statically partitions simulation objects different simulation managers of the kernel. This might result in unbalanced distribution of active simulation objects between managers that lead to uneven loading of nodes. This could be avoided by dynamically migrating simulation objects between nodes. A load balancing system can be designed that records load history of each node and initiates migrating of objects.
CHAPTER 7. CONCLUSIONS & FUTURE RESEARCH  7.2. SUGGESTIONS FOR FUTURE WORK

7.2.4 Real Simulation Models

PingPong and SMMP are synthetic models that were developed to test the correctness of the kernel. These synthetic models are less helpful in characterizing and analyzing the performance and scalability of the kernel. In particular, when the kernel is extended to multi-core and many core architectures, these models fail to test the parallelism added to the kernel. Real world complex systems like weather and logic gates can be captured as simulation models that could help testing the threaded kernel.

7.2.5 Atomic Data Structures

Efficiency of Time Warp data structures determine the performance of WARPed and its optimizations. Atomic instructions leads to waiting CPU cycles and mutex involves context switches. Frequent waiting and context switch affects the performance which deteriorates with higher thread count. Waiting CPU cycle and context switches could be reduced by building atomic data structures [50]. Transaction memory [51] based programming could be explored for lock and contention free data structures.
Appendix A

WARPED Configuration File

# FileName: parallel.config
# Notes : Empty lines are ignored; a comment is indicated by a "#".
# Choices:
# [a] True or False
Debug : true
ParallelDebug
#To Use: set true and
#Kick off a simulation, all sim managers will be sitting there spinning,
#Attach on every CPU with gdb, and then break them out and you could watch
#every CPU in gdb simultaneously.
SpinBeforeSimulationStart :False

# Simulation Options:
# Time Warp, (Single thread Time Warp simulation)
# ThreadedTime Warp (Time Warp with multiple threads executing objects)
# Sequential (Single thread sequential simulation)
Simulation : ThreadedTime Warp
#Time Warp Scope
APPENDIX A. WARPED CONFIGURATION FILE

Time Warp
#
# ThreadControl:
# WorkerThreadCount :
ThreadControl
WorkerThreadCount : 2

# Type Options:
# ThreadedTimewarp has only Multiset Scheduler
Scheduler
Type : MultiSet

# EventList:
# Type Options:
# Default, MultiSet, Threaded
# ThreadedTimewarp has only Multiset EventList
EventList
Type : MultiSet

# CommunicationManager:
# PhysicalLayer Options:
# MPI requires (--enable-timewarp)
# TCPSelect requires (--enable-timewarp)
# UDPSelect requires (--enable-timewarp), not implemented yet
# Type Options:
# Default
CommunicationManager
PhysicalLayer: MPI
APPENDIX A. WARPED CONFIGURATION FILE

Type: Default

# StateManager:
# Type Options:
# Periodic, Adaptive
# Period Format:
StateManager
Type: Periodic
Period: 10

# OutputManager:
# Type Options:
# AntiMessages Options:
# Default
# One
# FilterDepth, AggrToLazyRatio, LazyToAggrRatio & ThirdThreshold
# are used for Dynamic Cancellation
OutputManager
Type: Aggressive
AntiMessages: Default
FilterDepth: 16
AggrToLazyRatio: 0.5
LazyToAggrRatio: 0.2
ThirdThreshold: 0.1

# GVTManager
# Type Options:
# Mattern, AtomicMattern
APPENDIX A. WARPED CONFIGURATION FILE

# Period format:
# Must be an integer value

GVTManager
Type : Mattern
Period : 1000

# Type Options:
# None, Cheby

OptFossilCollManager
Type : None
CheckpointTime : 10000
MinimumSamples : 64
MaximumSamples : 100
DefaultLength : 2000
AcceptableRisk : 0.99
Bibliography


