The Integration of LlamaOS for Fine-Grained Parallel Simulation

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 Electronic and Computing Systems
of the College of Engineering and Applied Sciences

July 19, 2013

by

John Gideon
BSEE, University of Cincinnati, 2013

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

Abstract

LlamaOS is a custom operating system that provides much of the basic functionality needed for low latency applications. It is designed to run in a Xen-based virtual machine on a Beowulf cluster of multi/many-core processors. The software architecture of llamaOS is decomposed into two main components, namely: the *llamaNET* driver and *llamaApps*. The llamaNET driver contains Ethernet drivers and manages all node-to-node communications between user application programs that are contained within a llamaApp instance. Typically, each node of the Beowulf cluster will run one instance of the llamaNET driver with one or more llamaApps bound to parallel applicaitons.

These capabilities provide a solid foundation for the deployment of MPI applications as evidenced by our initial benchmarks and case studies. However, a message passing standard still needed to be either ported or implemented in llamaOS. To minimize latency, llamaMPI was developed as a new implementation of the Message Passing Interface (*MPI*), which is compliant with the core MPI functionality. This provides a standardized and easy way to develop for this new system.

Performance assessment of llamaMPI was achieved using both standard parallel computing benchmarks and a locally (but independently) developed program that executes parallel discrete event-driven simulations. In particular, the *NAS Parallel Benchmarks* are used to show the performance characteristics of llamaMPI. In the experiments, most of the *NAS Parallel Benchmarks* ran faster than, or equal to their native performance. The benefit of llamaMPI was also shown with the fine-grained parallel application WARPED. The order of magnitude lower communication latency in llamaMPI greatly reduced the amount of time that the simulation spent in rollbacks. This resulted in an overall faster and more efficient computation, because less time was spent off the critical path due to causality errors.

To my parents, my sister Rachel, my love Emily,

and all my friends that have changed my life forever

at the University of Cincinnati

Acknowledgments

I would first like to thank my advisor, Dr. Philip Wilsey, for his tremendous support with the writing and editing of my thesis, as well as throughout my time at the University of Cincinnati. He is one of the main people that originally encouraged me to engage in research at the higher level. In addition, I would like to thank Dr. Carla Purdy and Dr. Fred Beyette for being members of my committee and providing helpful insights throughout college. Also, I would like to thank other UC faculty members, such as Rob Montjoy for his help in the lab, Julie Muenchen for her assistance with the graduate program, and Teresa Hamad and Anita Todd for their guidance.

The students inside Dr. Wilsey's lab also provided a large amount of support, especially my research partner William Magato. He is the creator of llamaOS, the foundation upon which my research is based. In addition, he stayed up many long nights to help me fix bugs and further stabilize and add features to his operating system.

I would also like to thank Tom, Tori, Tyler, Sarah, Gabe, Peter, Kodi, and my many other friends who supported me throughout my time at the University of Cincinnati. They managed to keep me sane while completing this project. Finally, I would like to thank my mother Beth, my father Terry, my sister Rachel, and my loving girlfriend Emily for their unwavering support. I love you all very much and greatly appreciate everything you have done for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

High performance computing (HPC) systems are usually constructed from application specific and expensive hardware. Generally the nodes of an HPC system are interconnected using high performance communication systems, such as InfiniBand. While this may be cost-effective for large corporations with computational needs that require and justify the expense, a high cost HPC system is generally too expensive and inflexible for smaller research groups. *Beowulf* clusters provide an inexpensive alternative that provide smaller groups suitable performance of the more advanced systems with the flexibility of loosely-coupled hardware.

Unfortunately, most Beowulf clusters simply contain consumer desktop hardware with average speed Ethernet connections and general purpose desktop or server operating systems [1]. While this provides the users all the resources of a parallel computing system, it comes at the cost of inefficient drivers that may not be well suited to parallel computing. This is especially the case for *fine-grained* parallel computing, which greatly emphasises low driver latency. For example, the implemented network stack may emphasize bandwidth over latency and may be scheduled to execute only every one hundred milliseconds. While some attempts have been made at directly improving conventional operating systems for better parallel computational performance, these changes are often difficult to make and maintain.

Virtualization is a method that has been explored to decrease the difficulty in deploying and customizing the operation of a cluster [2]. This possibility has been increasingly utilized with the integration of hardware virtualization capabilities into the x86/x64 instruction set [3]. Virtual cluster systems can now be constructed with little noticeable overhead [4]. However, virtual systems are required to share all of the I/O resources of the physical machine and in cluster systems networking performance can take a hit. Many solutions have been explored to improve the sharing of these devices [5], including directly assigning certain resources to selected virtual machines. However, most of these projects seek to improve bandwidth performance, rather than latency.

One alternative which has been proposed by University of Cincinnati researchers is the Low-LAtency Minimal Appliance Operating System, called *llamaOS* [6]. The goal of llamaOS is to remove this virtualization overhead to allow for better I/O and parallel computational performance. This is achieved by allocating a virtual machine for the *llamaNET* driver, which has complete control over one of the network interface controllers (*NIC*). The driver has access to shared memory between virtual machine and communicates with the *llamaApps* in virtual machines on the same node. Whenever a message is received by the llamaNet driver it is immediately placed into the receive buffer in shared memory for low latency use in the application. Transmits are handled in a similar manner. Several benchmarks have been run to show that not only does llamaNET match the latency of a native networking application, but it actually reduces latency by an order of magnitude for message sizes below 1kB.

1.1 Motivation

The capabilities of llamaOS provide only the very basic foundation for the deployment of parallel applications that rely on low latency. The system is in the early development stage and requires standardization of the message passing functionality in order to be scaled to larger clusters and use pre-existing applications. In addition, it is necessary to develop a standard method of deploying the new system into a full sized virtualized Beowulf cluster. Finally, to date only simple ping-pong tests have been run on the system to prove lower latency. It is still necessary to perform further benchmarks and case studies in order to prove the system has realistic merit, despite costing the user one processing core per node.

To overcome these issues, a version of the Message Passing Interface (*MPI*) [7–9] has been developed for llamaOS, called *llamaMPI*. This provides a low overhead method of facilitating communication in the virtual Beowulf cluster and meets the standards for all basic and even some advanced MPI functionality [8]. This allows for almost any previously written MPI program to be seamlessly ported into llamaOS. Furthermore, the creation of llamaMPI also provides a standard method of deploying the virtual machine nodes throughout the cluster.

Finally, because of the ease of porting previously written MPI applications into our system, many new benchmarks and case studies have now been performed. One of the first successfully executed suites was the *NAS* benchmarks, which range from computationally heavy to communicationally heavy processes [10]. Nearly all benchmarks, especially those that relied heavily on communication, performed at native or slightly better on our system. This was expected, as the NAS benchmarks mainly rely on high network throughput, which llamaNET matches equally versus native. Finally, a Time Warp synchronized parallel simulation called WARPED was built into the system to demonstrate the improved performance of fine-grained parallel applications that mainly rely on low latency. The results show an *average drop in execution time of around thirty or forty percent, dependant on the simulation performed.* While this was not as dramatic as hoped, further planned driver changes and the addition of threads to llamaOS should provide substantial boosts to performance.

1.2 Thesis Overview

The remainder of this thesis is organized as follows:

Chapter 2 provides background information on the key technologies that provide a foundation for this research. The use of virtualization and the Xen Type-I hypervisor is explained, as well as the concept of application specific operating systems. In addition, the central concepts behind llamaOS are described. The Message Passing Interface (*MPI*) is then briefly explained as a method of normalizing and simplifying network interfaces. Next, an important set of tools called the *NAS Parallel Benchmarks* [10] are described as a method of testing the effectiveness of new parallel systems. Finally, the parallel discrete simulation WARPED is shown as a high performance fine-grained program that should test the limits of the new system.

Chapter 3 explores related works to the area of dedicated operating systems for parallel discrete event simulation. The Time Warp Operating System [11] was one of the first attempts at achieving speed improvements to Time Warp at the device driver level. Years later, the micro-kernel Musik [12] provided greater flexibility by supporting a wide range of parallel discrete simulation techniques with the ability to mix and adapt. Finally, both of these are compared with the new system combining llamaMPI and WARPED [13].

Chapter 4 describes the implementation of llamaMPI, a new message passing standard for llamaOS. While llamaOS provided an excellent low latency system for the development of fine-grained parallel applications, it was necessary to standardize and simplify the interface. This chapter explains why the MPI standard was chosen, as well as the development of its core functionality. Concepts such as groups, communicators, point-to-point communication, and collective communication and their tuning are described. The chapter ends with a comparison of llamaMPI to the more conventional MPICH implementation, used later in native tests.

Chapter 5 explains the process of implementing two applications into llamaMPI, namely the *NAS Parallel Benchmarks* and WARPED. Because both of these applications are based on MPI they did not require significant modification to get working in the system. However, each required a certain amount of tuning or even the creation of additional features in llamaOS to achieve optimal performance.

Chapter 6 provides the results taken from the *NAS Parallel Benchmarks*, as well as from the WARPED parallel discrete event simulator case study. Furthermore, this chapter provides information on the two hardware test setups and the required llamaMPI parameters to get the applications running optimally. Each results section also provides details on the parameters used for each individual NAS or WARPED test. The results are divided between the two major applications and the two hardware setups on which they were run.

Finally, Chapter 7 will summarize the results from the entire research and will make conclusions about their implications. The chapter will also provide suggestions for future work, based off of those discoveries that warranted more work.

Chapter 2

Background

2.1 Introduction

LlamaMPI is standard messaging interface that combines together the concepts of many previously existing work. This chapter presents information about llamaOS, the operating system upon which llamaMPI is constructed, and information about the virtualized environment in which it will run. In addition, the MPI standard is explained in detail. Furthermore, the *NAS Parallel Benchmarks*, which are used to evaluate the performance of llamaOS and llamaMPI, are described. Finally, the parallel discrete event simulation WARPED system is explained, as it is used as the case study for the completed project.

2.2 Virtualization

Virtualization provides a considerable amount of flexibility in system configuration, allowing for new guest operating systems to be created, rebooted, and modified on the fly. While this previously came with a computational penalty, modern virtualized systems have been made more efficient through hardware assistance. Hardware virtualization has been built into both the x86 and x64 instruction sets, allowing for all but a few of the more important calls to be performed directly as usual [3]. The virtual machine monitor (*VMM*), also called a *hypervisor*, is the central part of a virtualized system which instantiates the guest operating systems and manages their resource and time allocation. Essentially, it can be viewed as a operating system of completely independent operating systems, which is important to isolating sections of shared clusters.

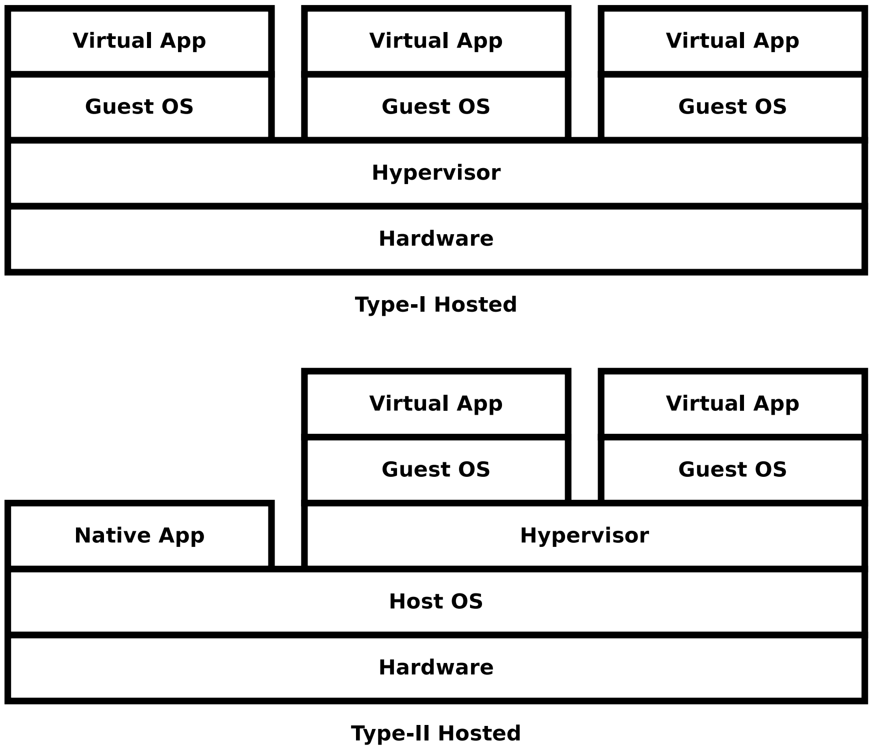

Figure 2.1: Hypervisor Classification

Most modern hypervisors can be classified into one of two types, described in Figure 2.1. *Type-I* (Bare-metal) hypervisors are implemented directly in the hardware and allow for direct access to the system resources. An example of a Type-I hypervisor, and the hypervisor used in this research is *Xen* [14]. Xen is installed directly onto the hardware of the system and booted during the rest of the machine's regular boot order. Therefore, the hypervisor is at the absolute base above the hardware. In order to keep the hypervisor's code as simple and reliable as possible, much of the conventional operating system responsibilities are actually borrowed from one of the guest operating systems. This special guest is given elevated permissions and the title `dom∅` [15]. While the core hypervisor controls memory and scheduling, the `dom∅` controls much of the peripheral drivers and I/O. Because the functionality of the `dom∅` is often different than that of a conventional OS, a few modifications are generally required for it to properly share its drivers.

In contrast, *Type-II* (Hosted) hypervisors are installed on top of a conventional operating system, such as Linux. One commonly used Type-II hypervisor is *KVM* [16], which leverages a host OS to perform both the core system functionality, as well as I/O. This allows for KVM installed as any other kernel module. This comes at the cost of not having the low level hardware interactions that a Type-I hypervisor can perform. However, for many virtualization applications this may be sufficient.

Regardless of which type of hypervisor is used, a process without much I/O will run at nearly the same rate in a virtual guest operating system as it will in a natively hosted operating system. This is because with hardware assisted virtualization, the guest operating systems will run directly on the CPUs without any interruption [3]. While virtualization provides the opportunity for multiple operating systems to share the same hardware, this often comes at the cost of higher latency. This is especially true with the concept of *full virtualization*, which allows for the guests to run unmodified with their original hardware drivers. This requires the host to emulate the hardware devices for the virtual machine and then map the I/O requests to the actual hardware drivers, increasing latency. However, *paravirtualization* allows for the guest OS to be aware of and take advantage of its virtualized state. For example, it can directly communicate with the underlying hypervisor to try to improve memory and CPU management. In addition, it can often request direct access to a certain peripheral, skipping the latency of the hypervisor.

Even though the overall design of llamaOS is hypervisor independent, Xen was chosen as the initial hypervisor to allow for easier development. Choosing a Type-I hypervisor allows for more isolated guests, direct access to the hardware, and lower latency. Furthermore, Xen is well documented and provides a useful system of paravirtual calls that allow the guest operating systems to interact with the hypervisor. *XenStore* is part of this functionality, and provides a shared memory space between guests to allow for communication. This ease of producing shared memory proved crucial in the development of the networking functionality of llamaOS.

Before moving on, it is important to fully understand how networking calls conventionally take place in Xen. Even with paravirtualized networking calls, as shown in Figure 2.2, the message must first completely be processed by the `dom∅` driver's protocol stack. Once the driver determines that the message is destined for another guest it is sent through through the `netback` and shared ring buffers to the `netfront` of the destination guest. Then the guest OS has to process the message through its own protocol stack, increasing the latency. While paravirtualization eliminates the need to completely simulate the network driver on the guest, there is still much added inefficiency because `dom∅` must route all messages. Although this method provides compatibility for almost any network, it greatly increases overhead [17].

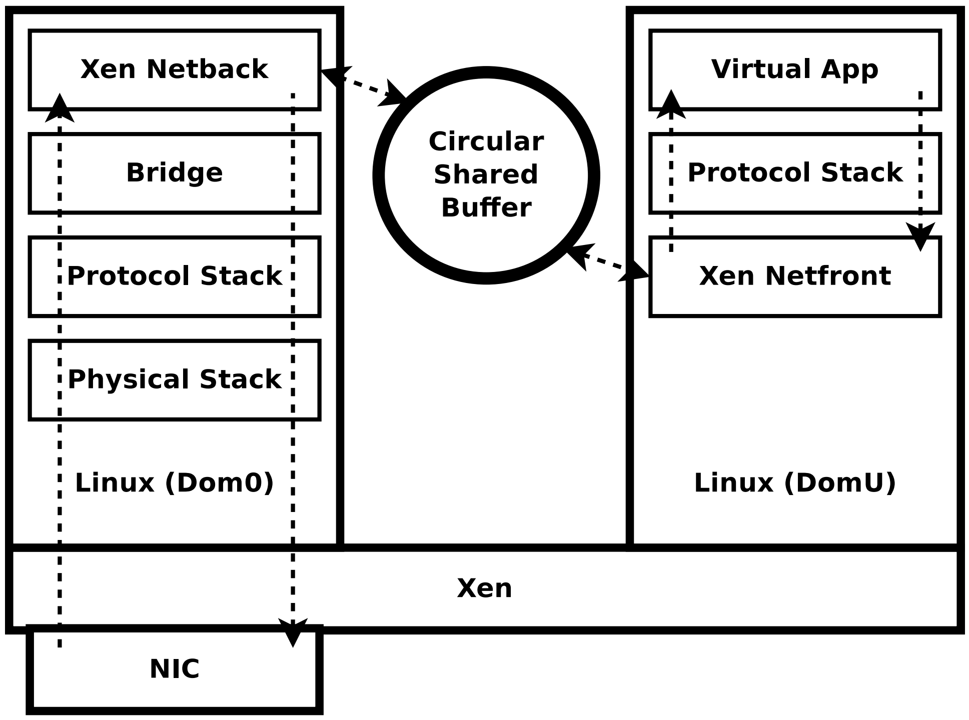

Figure 2.2: Xen Paravirtual Networking

One potential alternative to this problem is direct device access, or *PCI pass-through*. This became possible with the development of the I/O memory management unit (*IOMMU*) [18] as part Intel's VT-d [19]. This allows for the hypervisor to alter the memory mapping of peripherals and ultimately which guest is able to access them. The guest OS is then allowed complete and direct control over the peripheral, providing close to native latency. Due to the potential performance increases this provided, it was soon added to the Xen standard [20]. When the peripheral is a network interface controller, or *NIC*, the network latency is greatly reduced and becomes comparable to that of a native system. However, this is not scalable as all other guest operating systems lose access to that NIC. So in order for all guests to have access to the network they must either each have their own NIC at greater hardware costs or communicate with each other through shared memory, possibly increasing latency again. Despite this, studies have shown virtualized networks implementing direct access devices benefiting from the technology [21]. This has spurred the development of a new standard for network hardware, SR-IOV [22], that helps multiple operating systems safely alternate having direct access to a single NIC.

2.3 LlamaOS

This section introduces a novel system called *llamaOS*, the **L**ow-**LA**tency **M**inimal **A**ppliance **O**perating **S**ystem. LlamaOS attempts to take many of the positive features of virtualized environments without sacrificing the latency, price, or scalability that they usually entail. It provides all of the necessary features for developers to easily release HPC applications with network performance (both latency and bandwidth) that can meet and even exceed native TCP/IP performance.

LlamaOS is essentially a light-weight operating system into which all of the application code is directly built. This is similar to the idea of a *virtual appliance* [23], which has the application code pre-installed into the OS image, allowing for easier distribution of the application without configuration. All of the system services logic, as well as the application code is directly built into this one image. It also borrows some ideas of a *microkernel* [24], in that it provides only the absolutely minimal system services, such as memory management and scheduling, while sharing common resources. Lastly, it is also similar to an *exokernel* [25] by providing programs with direct and efficient hardware access. The key benefit of llamaOS is that because it runs in a standard commonly used virtual machine, it can be deployed and peacefully co-exist on a Beowulf cluster shared by multiple, distinct user communities.

The idea behind llamaOS is that each machine would launch several llamaOS nodes and coordinate their communication with shared resources. Different nodes can take on different roles as primary computation nodes or external networking intensive nodes. Rather than relying on modifications to existing operating systems, llamaOS was built from scratch to be absolutely minimal. It provides the necessary system services at a minimum overhead. Finally, it provides to developers many of the necessary tools to even make system driver development simple.

Even though llamaOS was designed to be hypervisor independent, the current implementation runs only on Xen, as it has easily provided many of the tools necessary for development and is proven in HPC applications [26]. Most of the Xen specific calls that are made in the code should be easily portable to other hypervisors if necessary. The system architecture of llamaOS on top of Xen is depicted in Figure 2.3. This shows some of the modules that are available and typically deployed with a llamaOS guest image. All images must at least contain the llamaOS-Xen module, which interfaces with Xen and is hypervisor specific. However, because such modules as the c-runtime environment are available, it is possible for

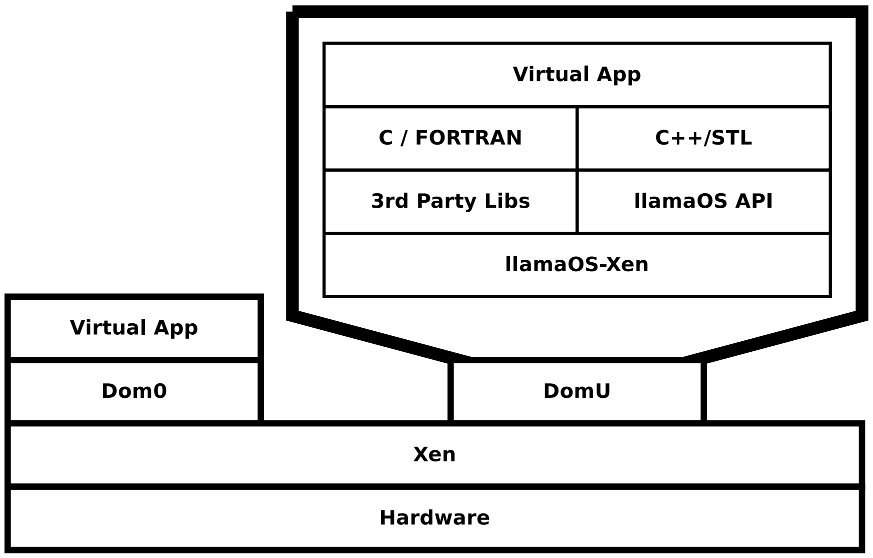

Figure 2.3: LlamaOS Xen System Architecture

an application developer to never have to interface with this module. In addition, system calls are simply routed through function pointers, which allows for the absolute minimum overhead when performing system services. Finally, much of the GNU C, C++, and FORTRAN libraries have been ported to llamaOS to allow for software engineers to easily build applications for the system. As with any typical application development environment, the application program simply begins with the *main* function.

Built into llamaOS, is the networking API *llamaNET*. As with the rest of the project, llamaNET was constructed to have low overhead and latency. It is based off of Ethernet and modeled after *GAMMA* [27], which is the open source version of the *Active Message API*. Active Messaging reduces the latency of conventional TCP/IP messages by replacing the standard software stack, with one more suitable for low latency messaging. LlamaNET builds upon this by taking full control of the hardware and providing a *zero-copy* data transfer. Each individual application may view and lock the data, but never exclusively owns the rights to it. Different networking hardware drivers can be easily written to allow for llamaNET to work on different systems. This is because within llamaOS users are given access to such tool sets as glibc, C++, and the STL. Because the drivers can be tailored to the specific hardware, this further improves the performance of the system.

Figure 2.4 fully describes the components in llamaNET. One llamaOS instance, designated the *driver node*, takes control of one of the NICs, providing access to the network. While it could eventually have added

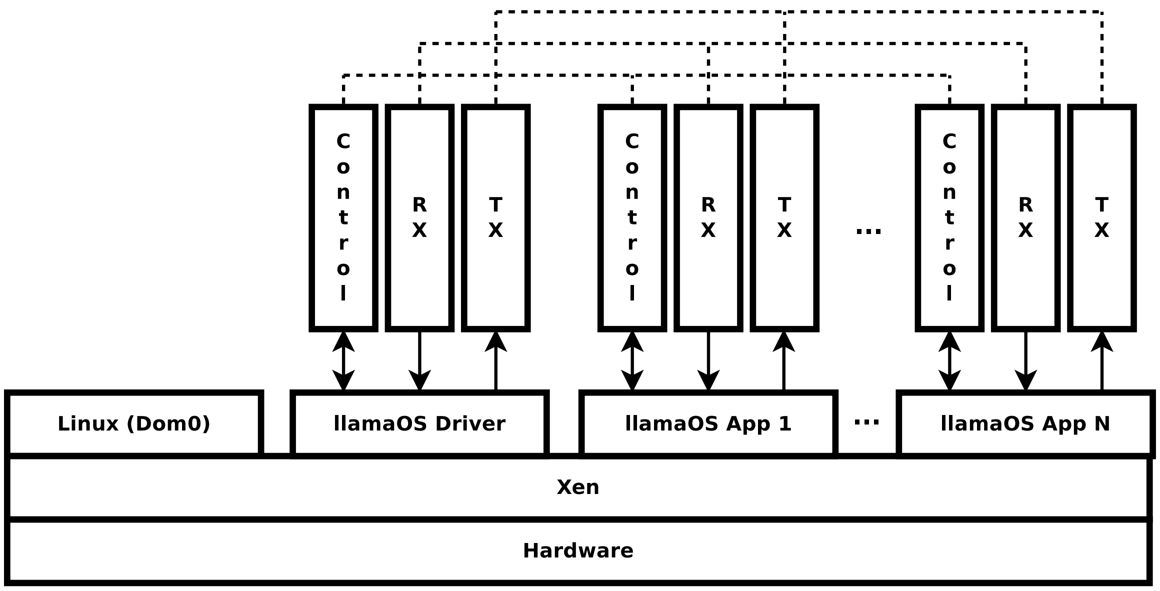

Figure 2.4: LlamaNET Block Diagram

functionality, all it currently does is simply wait for messages to be received and then place them in shared memory. If it detects a transmit message request in shared memory it sends it out over the NIC. Additional llamaOS nodes can be instantiated to actually implement the application, performing communication with the driver node over shared memory. These nodes are called *application nodes* or *llamaApps*. LlamaNET is not necessarily the first system to take this approach, and other systems further reinforce the idea of sharing network data between the guest operating systems [28]. However, these systems have not realized the performance increase of combining a minimal kernel with communication utilizing direct NIC access.

The llamaNet API uses a simple C++ interface to make all network calls, following the basic RAII design [29]. The *receive* function waits for a message to appear in the shared buffer and then passes the buffer to the user for processing. Once the processing is finished the user calls the *release* method to free up the buffer for further data. Transmits are done by first calling the *reserve* method to provide the user with a blank buffer in shared memory for transmitting messages. After the buffer is filled, the *send* method is called to mark the message as ready for transmission. The driver node will see this and immediately send out the message over the network.

Because all nodes all share common send and receive buffers it was necessary to find some way to distinguish them from each other for addressing purposes. Each node within a machine can attach a unique ID to the message header. In addition, it can screen for a certain ID upon reception of messages. This

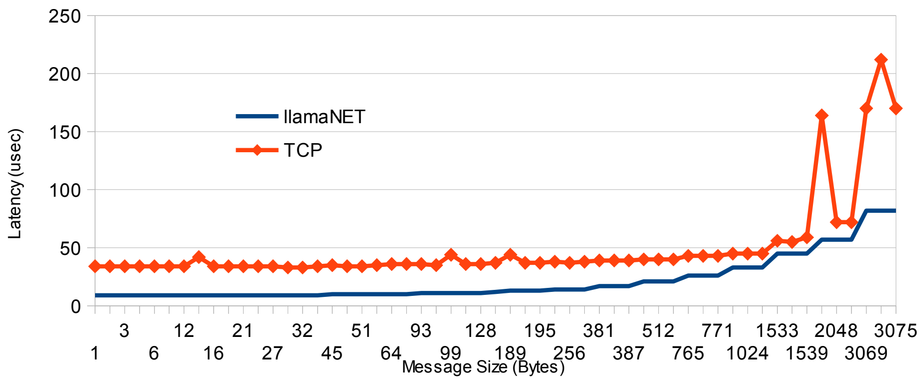

Figure 2.5: LlamaNET NetPIPE Latency Results

means that although each node sees all outgoing and incoming messages it can choose a unique ID for itself and then only filter out those messages. Because all outgoing messages are shared, intra-machine communication is simply accomplished by sending out a message to the other node's ID. The other node will match the ID on the way out and will retrieve it in the same as a typical receive.

As shown in Figures 2.5 and 2.6, initial testing has already shown the superiority of the llamaNET system to conventional virtual message passing and even native operating systems [6]. A simple ping-pong benchmark called NetPIPE was used to attain these results on both llamaOS and on a native system. The tool determines the latency and bandwidth of a system at different message data sizes, increasing in an exponential fashion. In all cases llamaNET performed better — providing lower latency and higher

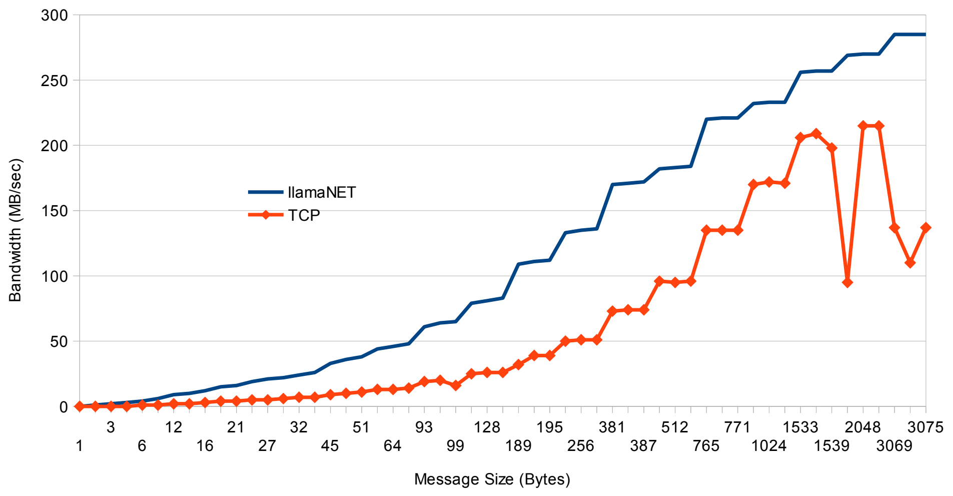

Figure 2.6: LlamaNET NetPIPE Bandwidth Results

bandwidth. In addition, llamaNET was much more predictable and the plots followed a fairly consistent trend. Some of the irregularity of the native plots may be explained by randomness in the scheduling of the TCP/IP software stack. This may demonstrate another possible benefit of using the llamaNET system — regularity. However, further tests were not yet performed and a communication standard needed to be adapted.

2.4 Message Passing Interface

The Message Passing Interface (*MPI*) is a language-independent communication standard that was developed to allow for easier development and portability of parallel applications [7–9]. It supports both simple point-to-point communications, as well as more complex collective communications, such as `MPI_Reduce`, which performs the second step of a MapReduce operation. MPI permits the programmer to worry only about the parallel algorithm and not the underlying hardware drivers or implementation. It consists of a set of function calls that can be made from C/C++ or FORTRAN that are automatically optimized for speed depending on the hardware setup. The following section outlines each of MPI's major components.

Two important concepts that easily allow for the portability of MPI code are *groups* and *communicators*. Groups contain a certain number of processes N that have continuous unique ranks from 0 to N-1. In addition, processes can be part of multiple groups. For example, a programmer can divide the world group into separate sub-groups by the machine on which they are running to optimize data transfers. However, it is important to keep in mind that groups are not globally synchronized and are kept local to each process. Groups can then be used to create communicators which are globally synchronized and allow for a set of processes to selectively communicate with one another. This isolates their set of messages and allows for reusable parallel libraries to be created. When an MPI program is initialized it contains only the global communicator, `MPI_COMM_WORLD`, which contains all processes. All other groups and communicators are created by subdividing this original communicator.

Once a communicator is established, it can be used for messaging. The simplest form of communication in MPI is the basic send and receive functions, `MPI_Send` and `MPI_Recv`. In order to send a message a specific communicator, destination rank, and message tag must be specified. While wildcards can be used on the source and tag parameters on a receive, it must always be performed within a specific communicator. Figure 2.7 provides an example detailing communication within two different communicators. The rightmost transaction demonstrates the use of wildcards. In addition, sends and receives can be blocking or non-blocking and and can be implemented in a multitude of ways. For example, one implementation could immediately send messages and store them all in a receive buffer. Another implementation could store them in a send buffer until requested by a receive. It is important to determine which version is best suited to an application to reduce overhead but also prevent buffer overflow and deadlock.

Not only does MPI provide simple point-to-point operations, but it also provides many collective operations to speed up parallel algorithm development [30]. All of these functions require communications across

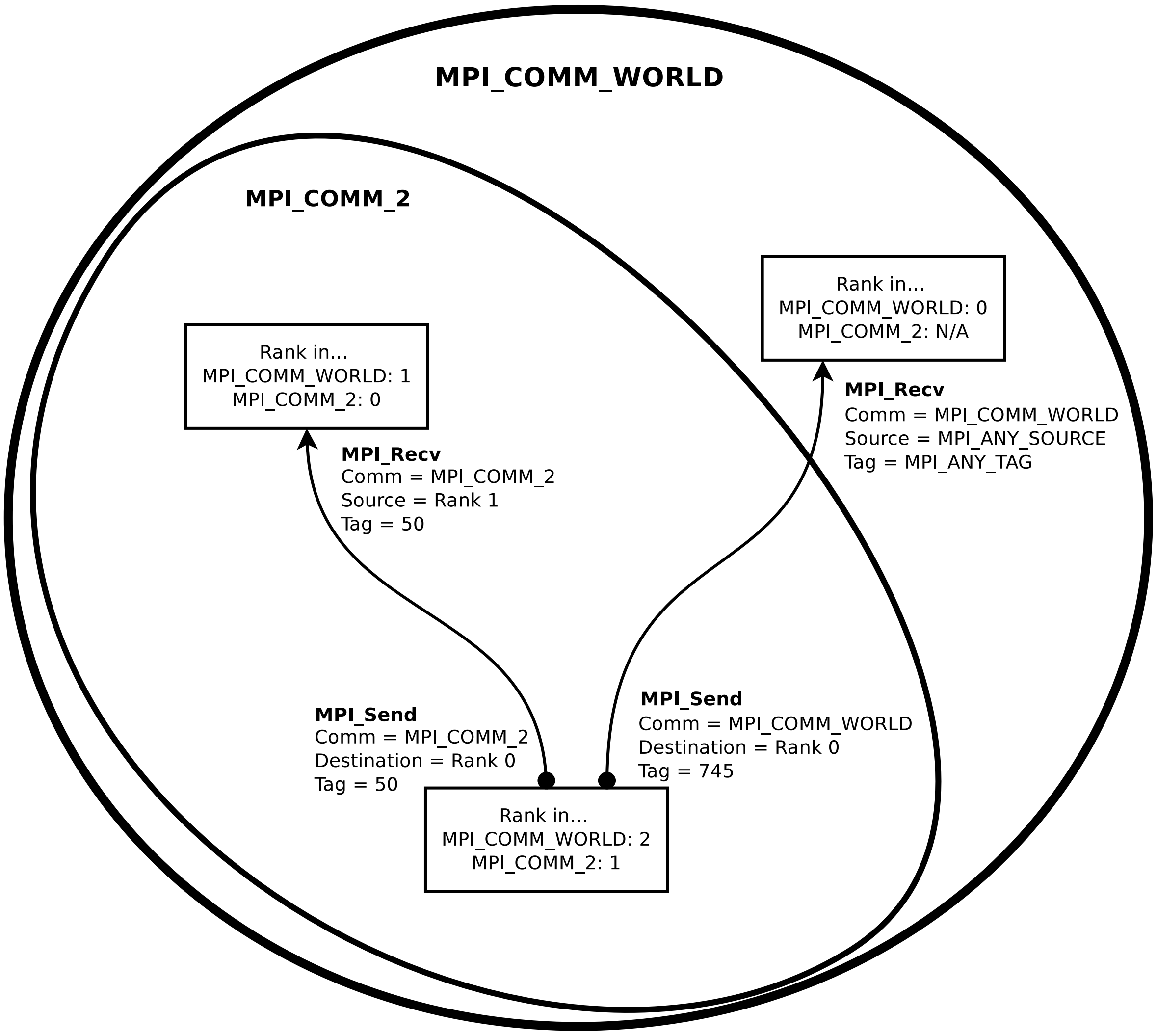
MPI_COMM_WORLD
MPI_COMM_2
Rank in... MPI_COMM_WORLD: 1 MPI_COMM_2: 0
Rank in... MPI_COMM_WORLD: 0 MPI_COMM_2: N/A
MPI_Recv Comm = MPI_COMM_2 Source = Rank 1 Tag = 50
MPI_Recv Comm = MPI_COMM_WORLD Source = MPI_ANY_SOURCE Tag = MPI_ANY_TAG
MPI_Send Comm = MPI_COMM_2 Destination = Rank 0 Tag = 50
MPI_Send Comm = MPI_COMM_WORLD Destination = Rank 0 Tag = 745
Rank in... MPI_COMM_WORLD: 2 MPI_COMM_2: 1

Figure 2.7: Example MPI Send/Receive

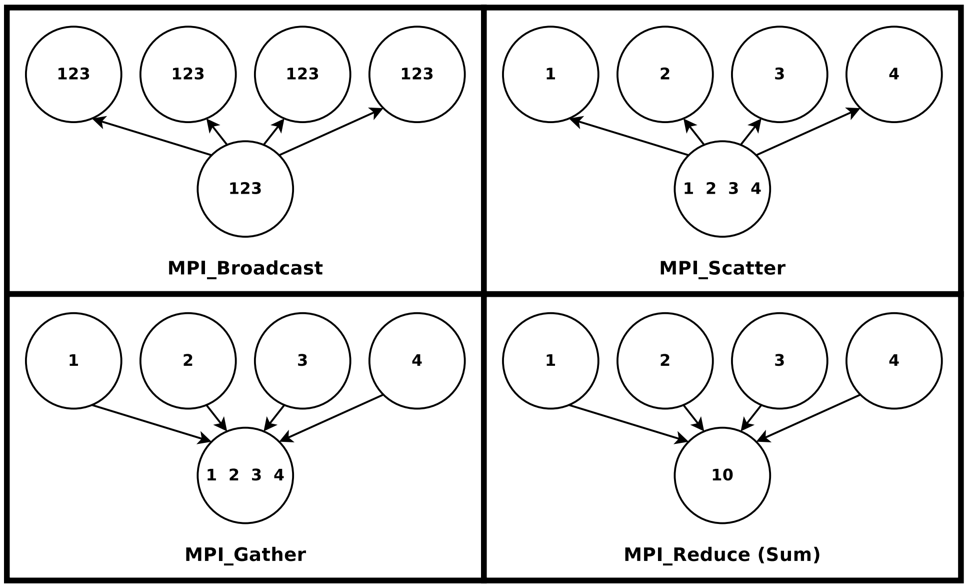

Figure 2.8: MPI Collective Functions

all members of a communicator. For example, the `MPI_Barrier` method syncs up all processes until they are at the same location in the code. Another common function is `MPI_Broadcast` which sends data from one process to all the other processes in the communicator group. The reverse of this is `MPI_Reduce` which pulls together data from the separate processes into one process while condensing it using some operation. Figure 2.8 depicts several of these operations.

Finally, another important concept within the MPI standard is the initialization and synchronization of the processes. However, this is largely specific to the implementation and is not defined by the standard. One commonly used implementation is *MPICH* [7], which is also used in the native MPI comparisons in this report. It uses a command line function called *mpiexec* which launches the processes on the separate machines across Secure Shell (SSH). In addition, each computational node is assigned a separate manager process to control startup and messaging. This process is the actual place where low level TCP/IP communication occurs between machines. Whenever the program makes a call to send or receive a message it just queries the manger process for transmit and receive buffers. Finally, each application is also required to call `MPI_Init` in order to start the interface and `MPI_Finalize` to close it. Section 4.8 provides further details on the initialization and implementation of MPICH. This concludes the basic overview of MPI functionality.

Benchmark	Full Name	Description
BT	Block Tridiagonal	Block triangular equations - high I/O
CG	Conjugate Gradient	Approximates eigenvalue of a sparse matrix
DT	Data Trafic	Black hole, white hole, and binary shuffle
EP	Embarrassingly Parallel	Solves integral through many random trials
FT	Fast Fourier Transfortm	Solves 3-D partial differential equation
IS	Integer Sort	Sorts a large number of integers
LU	Lower-Upper Symmetric	Solves a lower and upper triangular system
MG	MultiGrid	Implements simplified multigrid kernel
SP	Scalar Pentadiagonal	Block triangular equations

Table 2.1: NAS Benchmarks

2.5 NAS Parallel Benchmarks

The Numerical Aerodynamic Simulation (*NAS*) Program at NASA developed the *NAS Parallel Benchmarks* (NPB) to provide a suitable set of benchmarks for highly parallel computers [10]. At the time of their development the technology used to evaluate the quality of such systems had not advanced at the same rate as the systems themselves. The *NAS Parallel Benchmarks* effectively model many common computational aerodynamic problems using a ”paper and pencil” algorithmic approach to allow for verification and portability. In addition, the standard originally required that all implementations should be written in Fortran-77 or C code, as they were the most commonly used scientific languages at the time. Many different variations have been created for use with MPI, OpenMP, and single serial processes for comparison. The remainder of this thesis will focus specifically on the NPB3.2 MPI release, which contains the benchmarks listed in Table 2.1 [31].

Contained within the benchmark suite are many separate programs that try to reproduce important aerodynamic processes. They are mostly divided into kernel and simulated application benchmarks [10,31]. The kernel benchmarks FT, MG, CG, EP, and IS contain the small commonly run computations in aerodynamic applications. The simulated application benchmarks BT, SP, LU, and DT run computational fluid dynamic (CFD) problems. Within each of these divisions certain benchmarks may stress raw CPU power, while others may mainly test the communication interconnect. Taken together, this suite of benchmarks can effectively measure the total computational and communicational performance of a system. However, it should be noted that the communication-heavy benchmarks mainly emphasize high throughput over low latency. The list below enumerates the parts of a system that are especially tested by each benchmark [10,32,33]:

- **BT**: This is the only of the benchmarks to rely heavily on file system I/O. In addition, the benchmark seems to be most effected by the computational power, rather than the communicational efficiency.

- **CG**: The conjugate gradient benchmark contains a balance of computation and communication. The computations performed are unstructured matrix multiplications and use irregular long distance messages.

- **DT**: This program tests communication by simulating a network and passing messages around through nodes. At its simplest, the black hole simulation has every node send a very large message to one central node. The white hole simulation is the reverse, with one node spewing many messages to others. The number of nodes required increases with each class size.

- **EP**: Embarrassingly parallel mainly tests the computational speed of the nodes. It starts many processes at the beginning and has very little communication to keep them in sync. The speed of this benchmark is linearly proportional to the number of processors.

- **FT**: The computation of an FFT is communication bound. Increasing the number of processors in the system does not necessarily increase performance, and can actually often inhibit it.

- **IS**: The integer sort benchmark is mostly based on communicational efficiency. Increases in communication time dominate the run time of this process.

- **LU**: This benchmark is balanced between computational and communicational work. Increasing the efficiency of either should proportionally improve the run time.

- **MG**: The multigrid kernel is mostly computationally bound and is greatly effected by the number of processors in the system. The communication time accounts for a much smaller fraction of the time.

- **SP**: This benchmark is a reasonable balance between computation and communication. An improvement in either should bring a proportional increase in efficiency.

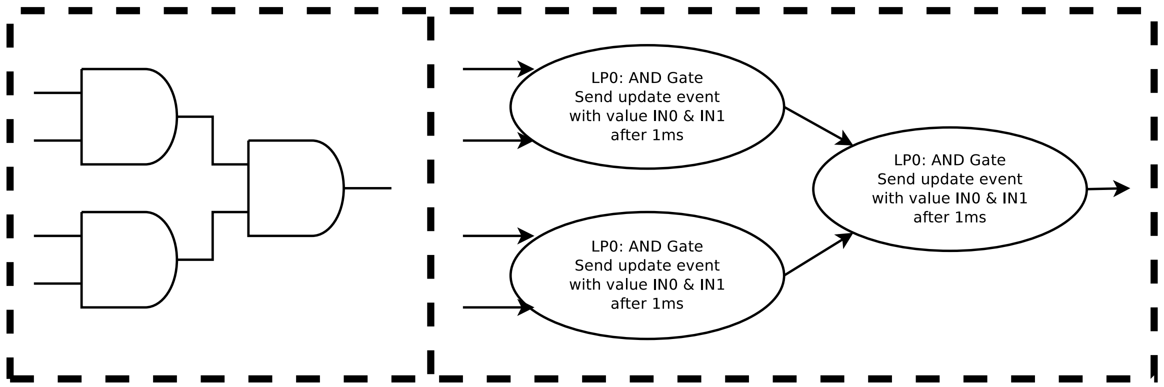

Figure 2.9: Example Logic Processes

2.6 The WARPED Parallel Discrete Event Simulator

While benchmarks are useful to determine system performance, a case study with a fine-grained application should highlight the effects of having lower latency. A discrete event simulation (*DES*) is one such application that models a system that occurs in many distinct time segments. The basic parts of a DES are as follows:

- **Global Clock**: This keeps track of the simulation time.
- **Event List**: This enumerates all the events that will be processed once the global clock reaches their time-stamp.
- **State Variables**: These contain all of the information regarding the system — the memory.

In order to effectively model a DES, the system is divided into a series of logical processes (*LPs*) that represent separate physical parts of a system [34]. For example, when modelling a logic circuit, the separate gates would be viewed as individual LPs. The signals travelling in between the gates would be represented as time-stamped events. Notice how this denotes all inputs as incoming events and outputs as outgoing events. Figure 2.9 demonstrates this derivation of LPs from physical processes. The inputs and intermediate signals between the gates would be relayed with time-stamped messages. This is a trivial simulation, as there are no state variables needed since the system has no memory.

The main limitation of conventional discrete event simulation is that all LPs must be gated through one central queue. Parallel discrete event simulation, or *PDES*, attempts to improve this system by concurrently processing the LPs [35]. In order to keep the output of the simulation valid it is important to follow the *local causality constraint*, which states that all events must be processed in increasing order [36]. This makes certain that all past events are factored into future decisions and future events do not affect the past. While this is easy to maintain within each local process, PDES suffers from synchronization issues globally when events need to be sent between LPs. If an event is scheduled out of order a *causality error* occurs and the output of the simulation is likely compromised. In order to overcome this issue one of two classes of techniques can be used. *Conservative* techniques have the goal of preventing causality errors from ever occurring [37, 38]. *Optimistic* techniques assume that causality errors occur relatively infrequently and instead tries to just detect and correct them [35, 39].

One particular optimistic synchronization technique is known as the *Time Warp Mechanism* [35, 40]. In Time Warp an LP computes all events as they come without any regard for potential causality errors. If an error occurs by an event arriving after it needed to be processed, called a *straggler event*, the LP simply restores to a previous time. This *rollback* not only involves restoring the LP to its previous state, but also cancelling the premature outgoing messages that have been sent. These message cancellations are known as *anti-messages*. The modified parts of a Time Warp simulation are listed below:

- **Input Queues**: Each LP needs to not only keep track of all events to be processed, but also all previously processed events for potential rollbacks.
- **Output Queues**: Each LP also requires a queue of previously output messages in order to know what anti-messages need to be sent in the case of a rollback.
- **State Queues**: Each LP needs to store previous states in order to be able to easily restore itself to a previous state on rollback.
- **Local Virtual Clocks**: Each LP keeps track of the previous time-stamp processed and considers this the local virtual time. If a new incoming message has a time-stamp before the local virtual time then a rollback must occur.
- **Global Virtual Time**: The simulation is required to approximate a global virtual time (GVT) for system synchronization [39]. This is equal to the minimum time-stamp of all unprocessed events, as no causality errors can occur before these successful events.

Because the Time Warp Mechanism requires previous states to be saved it can also result in much larger memory usage. The simulation can perform fossil collection, deleting all of the data dealing with events and states before the GVT. Many separate implementations of the Time Warp Mechanism have been developed to optimize these different facets of its operation.

One implementation designed by researchers at the University of Cincinnati is called WARPED [13]. The project began by taking the original VHDL analysers built in the QUEST [41] and VAST [42] projects and generalizing them into a new discrete event simulation kernel. This allowed for new Time Warp algorithms to be easily implemented and compared. In addition, it was built to provide time and space efficient algorithms that could easily be scaled in regards to communication and computational power. MPI was used for all message passing to allow for flexibility in the actual underlying hardware [7]. The actual code itself was written in C++ so that object oriented modules could be easily interchanged when needed. Furthermore, most of these configurations are able to be made at run time to avoid long compilations in between tests [43].

Parallel discrete event simulation and the Time Warp mechanism provide the ability for simulation to be executed in much closer to real time. This is important, as many things that are modelled using PDES, such as weather predictors, become irrelevant after too much passed time. It should be noted that parallel discrete event simulations almost always contain fine-grained event processing steps and thus they are greatly effected by communication latency. This is because slower messages imply a higher chance of causality error. For this reason, it is important to further research into methods to implement WARPED with lower communication latency and provide a simulation environment more suited to PDES.

Chapter 3

Related Work

3.1 Introduction

While llamaOS is flexible enough to potentially have a wide range of uses, the primary application explored in this research is for fine-grained parallel discrete event simulation. This is later shown through the case study of the WARPED system using llamaMPI running on llamaOS. The idea of constructing an operating system for PDES is not unique to this research and has been explored as early as 20 years ago. This chapter of the paper will explore two alternative operating systems designed to particularly support parallel discrete event simulation, namely: Musik [12] and the Time Warp Operating System [11].

3.2 The Time Warp Operating System

One of the first attempts at building a specialized system for PDES was the Time Warp Operating System (*TWOS*). The system was pioneered by NASA's Jet Propulsion Laboratory (JPL) as a means of specifically optimizing the lower level operating system service routines for Time Warp simulations [11]. The system completely implements the Time Warp functionality and sets itself apart from general purpose operating systems by synchronizing processes using a distributed process rollback mechanism. Many of the typical OS features had to be reworked to adjust to this system, such as scheduling, networking, and flow control. Unlike other alternative operating systems explained in this chapter, this implementation fully committed itself to the optimistic Time Warp methodology.

The Time Warp OS is a single user system that only supports one independent process at a time. However, this process is distributed between multiple machines using a communication protocol that keeps the global state in sync. In addition, it is not possible to spawn additional processes dynamically at run time. Problems are statically split into several process at startup and then left to run until completion. More recently, TWOS has implemented some successful static and dynamic load balancing algorithms [44]. However, beyond these differences, it still behaves, and is modularly built, much like a conventional operating system. It just utilizes a different set of algorithms to perform tasks specific to PDES.

The programming model for TWOS is fairly straightforward, utilizing a simple object oriented language [11]. Each process is assigned a globally unique name and is given several routines that run at different times during execution. There are the usual initialization and termination routines, as well as message received and message query routines. Whenever the process wants to send a message to another process, it simply provides the other process' name and the data to send. It is not necessary to open any type of port or pre-assign which processes can intercommunicate. This allows for the easiest flexibility of programming the simulation.

Individual processes have the ability to automatically roll back their own execution and messages when needed. However, this comes at the cost of having to meet the following two criteria for all programmed code:

- The process must be deterministic. This makes certain that the process will run identically when rolled back and executed again, except for potential differences in input messages.
- Dynamic allocation of variables, such as using `malloc`, must not take place. Heap storage for rollback is not enforced as it is difficult and slow. Instead, variables on each process stack are simply saved.

In order to allow for programs requiring some degree of randomness to execute, each process contains a seed which is stored on the stack and attached to rollbacks. Therefore, if a rollback occurs the pseudo-random number generator will return to the past and produce an identical sequence.

Once a simulation is programmed, it is loaded into the TWOS to execute following the special algorithms designed to work efficiently with Time Warp. For example, processes are scheduled solely based off of which has the lowest virtual time. The process runs continuously until either the computation moves the virtual time ahead or a rollback takes place producing a process with a lower virtual time. In addition, the OS optimizes message queueing by placing all received and sent messages in queues in order of time stamp, not receive or send order. It also checks for matching messages and anti-messages in the same queue to annihilate, freeing space. TWOS also has the functionality to initiate process rollbacks due to administrative tasks, such as the memory being full. This tool, called *message sendback*, allows for computations to stay in sync when one needs more time for storage processing. Finally, the OS has special routine for determining the GVT and committing the process states. This allows for memory to be freed up as soon as possible to prevent process rollbacks.

Initial benchmarking of the Time Warp OS showed reasonable speedup in the processing of an set of irregular simulations [11]. Additional work has been performed to determine its performance on a variety of different models. For example, a combat simulation, called STB88, was tested on an increasing amount of processors to determine scalability [45]. While it produced a speedup factor of 28.6 on a 60 processor system, the speedup only increased to 38.5 for a 128 processor system. Further research has been performed into limiting the amount of optimism in the system to prevent rollbacks [46]. However, these modifications saw only minor speedups and often accidental increases in run time due to incorrect settings. This could be likely due to the Time Warp Operating Systems' poor performance on riskfree or conservative simulations [47]. TWOS may provide a large amount of speedup for a certain number of cores as long as the simulation is favorable to the optimistic model.

3.3 Musik - A Micro-Kernel for PDES

While the Time Warp Operating System attempts to perform a certain type of simulation really well, the micro-kernel *Musik*, also known as μsik, provides a wider range of tools for PDES in general. Developed by Dr. Kalyan S. Perumalla, Musik strove to provide a large amount of possible PDES optimizations that were easily interchangeable [12]. The author states that this is similar to how great music is often composed with many cooperating instruments, rather than just one. Not only can different types of algorithms be easily interchanged for testing, but they can be all used on the same simulation in order to determine the best mix to solve a problem.

Unlike TWOS, Musik is a micro-kernel rather than a full operating system. This allows for the easier addition of system services as separate external modules. In addition, this allows for multiple simulation modules to be written and interchanged without redesigning the entire PDES core. The design was based on isolating the algorithms and processes that are common to all types of PDES and only placing that minimal amount of code in the micro-kernel. These core provided services are described as follows:

- **Naming:** The system presents a method of identifying processes to one another for communication.

- **Routing:** This provides the method for routing a message from a sending process to the receiving process. Communication is performed similarly if the processes are on different machines or sharing the same CPU. Successful message routing is guaranteed.

- **Scheduling:** The system determines how to allocate CPU time for each process to most efficiently suit the simulation. It also prevents livelock and deadlock.

In order to allow for a variety of algorithms to be used, including both conservative and optimistic approaches, the event structure and scheduler is kept general. The events proceed through a series of states throughout their lifetime with the state transitions determined by the type of simulation. In addition, these events can be sent using different messaging models, such as optimistic, lazy, and non-aggressive sends, to allow for the OS model to best match the simulation. Finally, the system provides a set of GVT estimation algorithms and efficient lists and queues to suit any type of computation.

Initial testing of the Musik system showed promising results with low overhead for large amounts of processes and events [12]. In addition, simulations ran well with both solely conservative and solely optimistic algorithms, although there was a larger amount of overhead for larger numbers of processors. Mixing conservative and optimistic processes also functioned correctly, although these results did not necessarily show an application that could benefit from this mixing. Later testing on a Blue Gene supercomputer showed the system's ability to scale with low overhead [48]. Conservative simulations were able to run with up to 16384 processors without degradation. However, the optimistic simulations topped out at 8192 processors before overhead became too large. Finally, this demonstrated the largest mixed mode computation to date with 1024 involved processors.

3.4 PDES Operating System Comparison

This thesis presents the creation of a communication interface for the llamaNET system, called llamaMPI, which provides efficient, low-latency messaging. For this reason, it is well suited to running fine-grained irregular applications, such as PDES. Chapter 5 of this thesis demonstrates the combined power of the WARPED simulator with the llamaMPI interface. Table 3.1 provides a comparison of llamaMPI combined with WARPED to two other operating systems, TWOS and Musik. The implementation and benchmarking of this new system will be given in later chapters.

	Time Warp OS	**Musik**	**llamaMPI + WARPED**
Developed By	Jet Propulsion Lab	Kalyan S. Perumalla	University of Cincinnati
Originally Developed	1987	2005	2012
OS Type	Full OS	Micro-Kernel	Micro/Exo-Kernel
Target Simulation	Time Warp	PDES	Time Warp
Supports Non-PDES Code	No	No	Yes
Runs in Virtual Machine	No	No	Yes
Interchangeable Optimizations	No	Yes	Yes
Supports Mixed Mode	No	Yes	No
Supports MPI	No	Yes	Yes
Time Warp at Driver Level	Yes	Yes	In Development

Table 3.1: PDES Operating System Comparison

Chapter 4

Design and Implementation of LlamaMPI

4.1 Introduction

LlamaMPI has been developed to make adapting parallel programs into llamaOS as easy as following the MPI standard. This makes most existing MPI programs compatible with llamaOS without any source code changes. This chapter explains the decisions that led to implementing MPI and why an existing version, such as MPICH was not simply ported to llamaNET. In addition, it details the design of the core functionality, such as groups, communicators, point-to-point messaging, and collective messaging. The startup process for the entire system and individual processes is also explained, as well as the tuning parameters for different applications. Finally, a comparison of llamaMPI to MPICH, a well known implementation of MPI, is given.

4.2 Initial Development

Early in the research it was determined that some standardized system of communication would be needed to make llamaOS practical for common applications. In addition, it was determined that the NAS benchmarks would provide a thorough method of benchmarking our system. Existing implementations of the NAS benchmarks were written for MPI, OpenMP, and Java networking. Finally, the ultimate end-goal of the research was to improve the latency of fine-grained parallel applications, such as the WARPED PDES system, which used MPI as one of its communication interfaces. Therefore, MPI was chosen to be implemented or ported, as it was best suited to our application goals.

Initially the idea of porting OpenMPI or MPICH was explored, as it is often better to use an existing solution than spend time implementing something completely new. However, it was eventually determined that porting would not be the best option for the following reasons:

- LlamaOS does not currently provide threading support. Mainstream versions rely on separate manager processes from the main process to actually control the message flow.
- The initialization code would need to be reimplemented to allow for processes to be started with an encapsulating virtual machine for each one.
- An additional module would need to be added to the networking protocol to support communication. This would require understanding how to add an additional device, which itself is an undertaking [49].
- Other implementations often sacrificed latency for bandwidth or functionality.

For these reasons it was appropriate to instead explore implementing a version of MPI specifically for llamaOS. That way it could also be optimized specifically for latency and operate in the same process as the main program. This new implementation of MPI, called llamaMPI, is detailed in the remainder of this chapter.

4.3 Groups and Communicators

One important concept within MPI is the ability to subdivide sets of nodes into smaller and smaller groups. In addition, when each of these groups send messages between each other over a communicator they should not interfere. This allows for the creation of reusable library functions, because they are guaranteed to not have any effect on the rest of the system. LlamaMPI fully provides this isolation and implements the core functionality of groups and communicators.

4.3.1 Groups

LlamaMPI provides all the basic functions of groups, as well as some more advanced ones. For example, it is possible to create groups through the union, intersection, or difference of two pre-existing groups. In addition, groups can be compared with each other and have their ranks translated. Unlike common

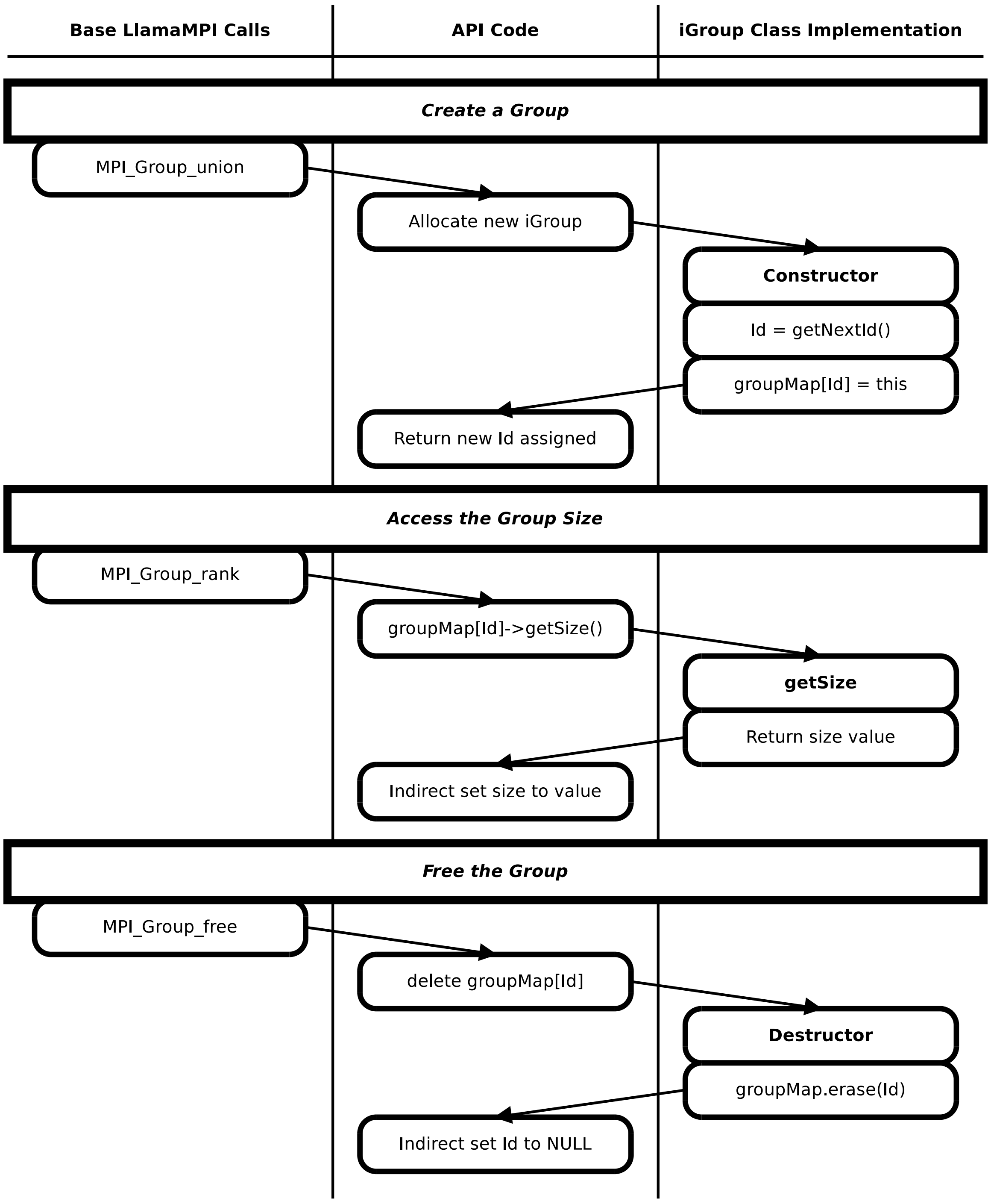

Figure 4.1: LlamaMPI Accessing Groups Via Tags

implementations of MPI, llamaMPI does not keep track of group garbage keeping automatically and requires one object for each instantiation. This simpler implementation should ideally keep overhead low.

Figure 4.1 demonstrates the process of creating, accessing and destroying a group. This begins at the application level with a call to `MPI_Group_union` which merges together two other groups. This process creates a new iGroup object, which contains all the functionality and data related to the new group. Next, it is necessary to map the iGroup object pointer to a unique integer ID to meet the MPI standard. Once an ID is determined, the key value pair is added to the global map and the ID is returned to the application level. From there forward, the program can access the group by simply using the ID key in the map and then dereferencing the pointer. This is demonstrated with accessing the group size. Finally, when it becomes necessary to free the group from memory, the key map pair can be dereferenced and deleted. The destructor will then erase the ID from the global map and change the returned ID to *MPI_GROUP_NULL*. This method of mapping an integer to an object pointer is common in the rest of the llamaMPI implementation.

4.3.2 Communicators

While groups provide llamaMPI with the necessary ability to divide nodes of a computation into separate sections, it is their attached communicators that allow them to send messages. The basic communicator, `MPI_COMM_WORLD`, allows for all nodes to communicate with one another. Upon initialization each node is assigned a *world rank* which defines its rank within `MPI_COMM_WORLD`. Any other communicator rank is defined from this initial rank. Each communicator consists of an internal group to keep track of which world ranks can communicate with which. Most of the MPI standard is fully implemented in llamaMPI, with the exception of intracommunicators. Only the intercommunicator type was implemented to provide simplicity and given time constraints.

An important property of communicators is that they can be accessed with a unique integer ID, that unlike group IDs, needs to be common between nodes within the communicator. This requires for there to be some form of consensus reached by involved nodes in the base communicator involved in creation. This process is outline in Figure 4.2 and uses the `MPI_Allreduce` function explained later in Section (4.5). Because a new communicator must be created from an existing one, the only nodes that must come to a consensus are those in the existing communicator. Furthermore, because this ID will be used later in the receive function to filter messages by communicator, it is necessary that the ID be unique within the existing communicator. This works by assigning all nodes a global `nextId` value for communicators. When determining an ID for a new communicator, the value will be chosen as the maximum value within the formative communicator. In addition, all nodes that join the new communicator will increase their `nextId` value by one. This should ensure that IDs remain unique within each communicator, but allow for the quick consensus to be reached.

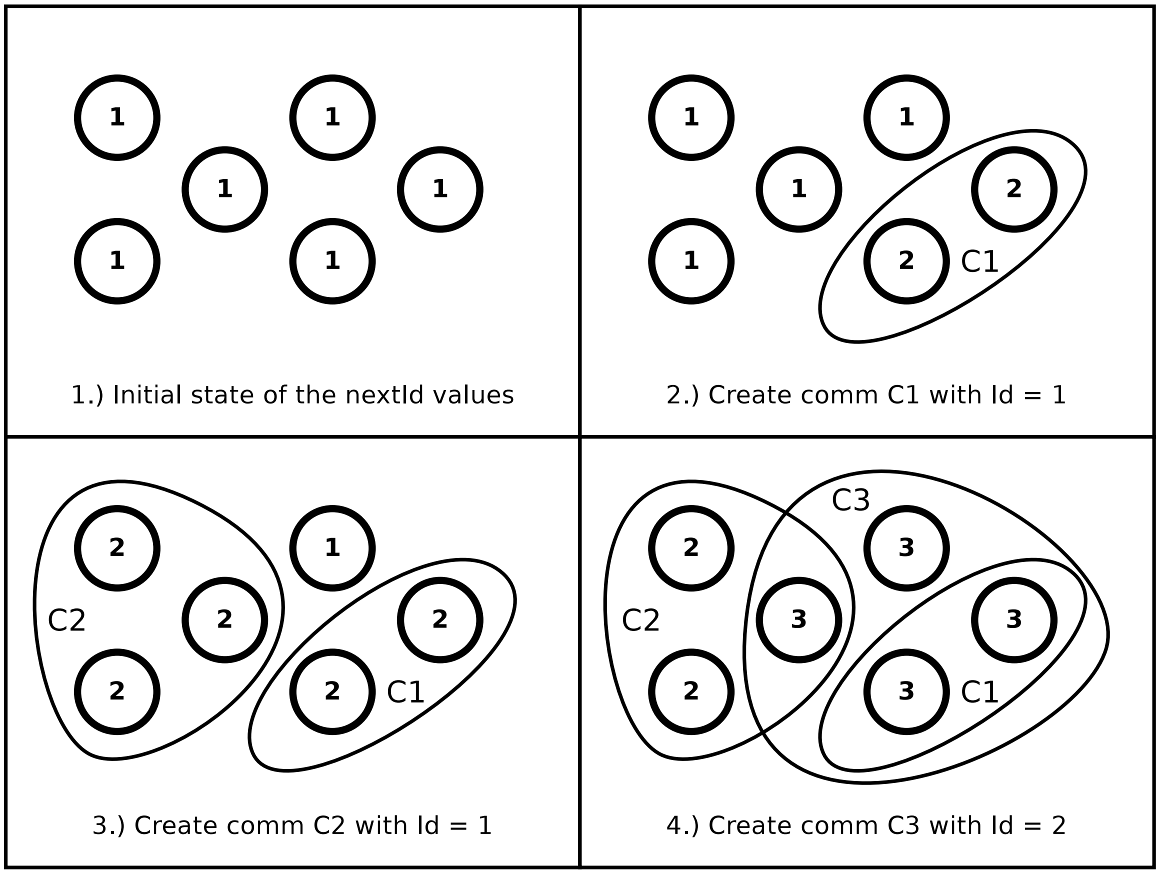
1.) Initial state of the nextId values
2.) Create comm C1 with Id = 1
3.) Create comm C2 with Id = 1
4.) Create comm C3 with Id = 2
C1
C2
C3

Figure 4.2: LlamaMPI Communicator ID Assignment

4.4 Point-to-Point Functionality

At the core of llamaMPI's library is a basic send and receive functionality. All higher level communication routines are based around these methods. It order to be compliant with the MPI standard, messages sent within separate communicators or with separate tags must be kept independent. Furthermore, messages read from separate sources should be able to be retrieved in any order without disrupting the other messages. In addition, the application needs a way of detecting messages without actually copying them into a buffer, called probing. The system also must have the ability to perform all these functions in a non-blocking manner using requests. Finally, llamaMPI must be able to break larger messages into smaller chunks to meet sizing constraints of the underlying hardware.

4.4.1 Sending, Receiving, and Probing

The first communication functionality implemented in llamaMPI was the basic send and blocking receive methods. Sends were implemented in the most simplistic way by just immediately calling the underlying llamaNET send function after requesting a shared buffer. The receive function was more complicated, as it required all messages to be stored until retrieval by a particular communicator, tag, source combination. In order to accomplish this, a set of mapped by communicator linked listed were created. The mapping completely ensured that messages from different communicators would be completely independent. In addition, the linked lists make sure that messages are retrieved relative to the same order they were received into the system. The following enumerates the steps associated with sending a message from one machine and receiving it on another, as depicted in Figure 4.3.

Sending a Message:

1. In order to send a message, first request a shared tx buffer space from the llamaNET driver.

2. Next, copy the message into the acquired buffer and finalize it, allowing it to send.

Receiving a Message:

1. In order to receive a message, first pull any already received parts of the matching message out of the receive queue and into the buffer.

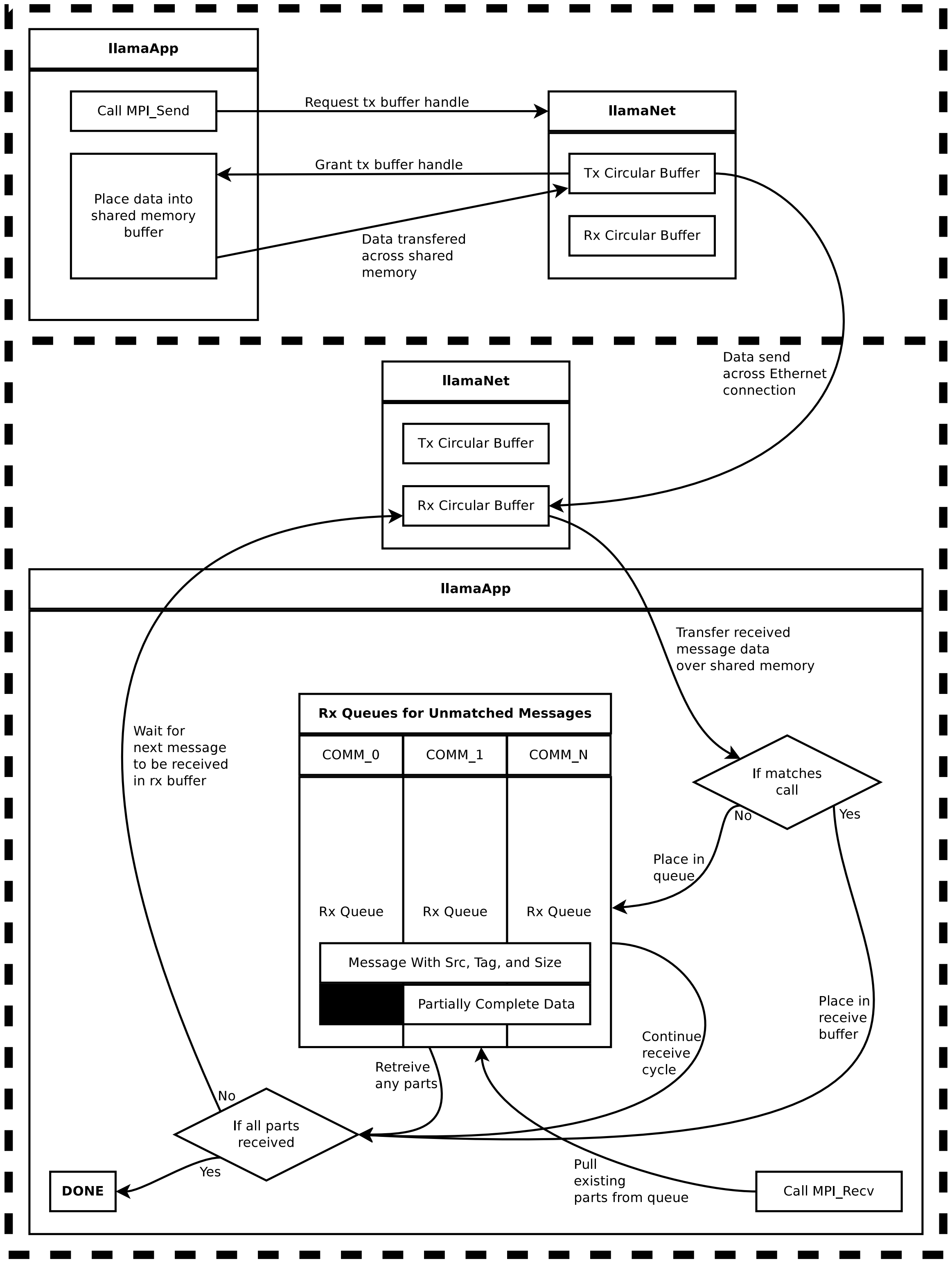

Figure 4.3: LlamaMPI Send/Receive Implementation

2. Next, check to see if all parts of the message have been received. If so, then exit.

3. Wait for a new message to appear in the shared rx buffer.

4. Check to see if the new message matches the desired message. If not, then place it in the matching queue and return to step 2. If the message matches, then place the new part in the receive buffer and return to step 2. Either way, return the utilized buffer to the llamaNET driver after use.

This method achieves the desired goal of message isolation, filtering by source and tag, and placing them in separate rx queues to be received out of order. In addition, messages can be received in many separate chunks and stored in the queues, allowing for llamaMPI to simulate whatever packet size is needed, even if it exceeds the hardware limitation.

However, this only demonstrates one of the four ways to receive or probe a message. Figure 4.5 shows all the possible permutations of the blocking/non-blocking receive/probe and shows their implemented algorithm as a flow chart. Figure 4.4 is the associated key for interpreting the flow charts. The main difference between a receive and a probe is that a probe never actually copies the message out into a buffer. Instead it stores all incoming messages into their matching queues and returns if/when a messages is fully received into the queue. This could allow the application to receive a message of unknown size by first waiting to determine its size before copying into a buffer. Blocking routines on the other hand do not return until the matching message is fully received or probed. Non-blocking messages will always immediately return with a flag which marks whether or not the message is complete.

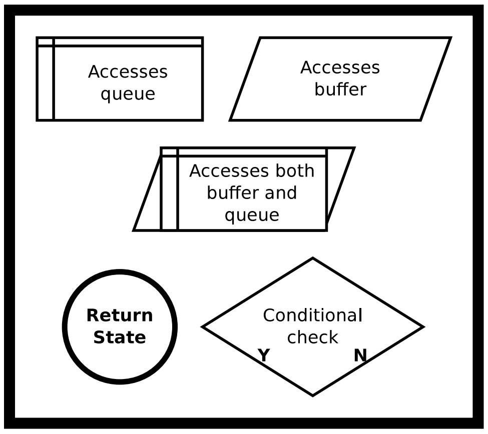

Figure 4.4: Key for Flow Chart in Figure 4.5

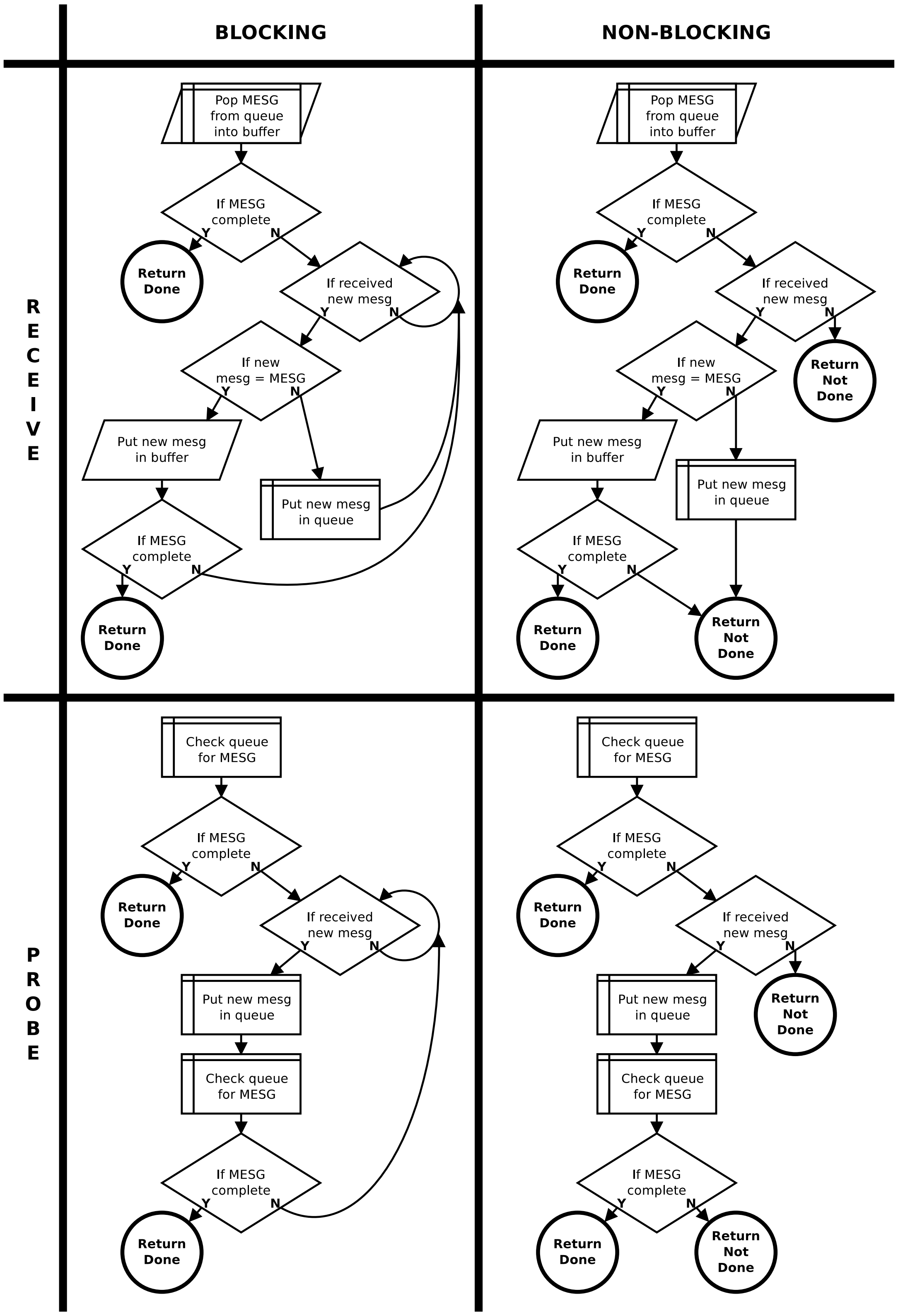

Figure 4.5: LlamaMPI Receive/Probe Blocking/Non-Blocking Flow Charts

4.4.2 Completing Requests with Wait and Test

LlamaMPI applications do not have direct control over the non-blocking receipt of messages. Instead, this is accomplished through the use of a system of requests, which are passed between routines. The process begins when the application calls either the `MPI_Isend` or `MPI_Irecv` function and a request is creates with a unique integer ID. Unlike the blocking send and receive functions, these non-blocking versions do not necessarily complete until a matching wait or test method is called with the request. Because sends are all non-blocking in the underlying implementation all messages are immediately sent, although it is still necessary to complete the request. However, non-blocking receives are not processed until completed. The generated request keeps track of all the related message information, such as receive buffer and message size. When the function `MPI_Wait` is called on a request, the program halts until the full message is received. This essentially reproduces the functionality of a blocking receive. On the other hand, the function `MPI_Test` will just check to see if a message is complete and return either way with a status flag.

In addition to the basic request completing functions `MPI_Wait` and `MPI_Test` there are a variety of other methods that allow for the completion of a group of messages. These functions wait or test that either all, any, or some of the request have completed. All possible combinations of these functions were implemented and are described in Table 4.1. For example, if an application is expecting a multitude of messages that will be received in an unpredictable order for which the program needs to block, the function `MPI_Waitall` would be of use. Another possibility would be for the program to block until just one of the requests is complete, using the `MPI_Waitany` function. A more general function, `MPI_Testsome`, simply checks the buffers of each pending request and returns those messages that are complete, if any. Refer to the table for a description of the other versions of the functions.

MPI Function	**No Requests Complete**	**Some Requests Complete**	**All Requests Complete**
MPI_Waitall	Block	Block	Return
MPI_Waitany	Block	Return lowest complete index	Return lowest index
MPI_Waitsome	Block	Return all complete indices	Return all indices
MPI_Testall	Return flag = FALSE	Return flag = FALSE	Return flag = TRUE
MPI_Testany	Return flag = FALSE	Return lowest complete index*	Return lowest index*
MPI_Testsome	Return flag = FALSE	Return all complete indices	Return all indices

* Also returns flag = TRUE

Table 4.1: LlamaMPI Wait and Test Calls

4.5 Collective Functionality

Another important facet of the MPI standard that builds upon simple point-to-point messaging is collective messaging. This category of functions deals with those that simultaneously involve every node in a communicator. They can be useful for abbreviating algorithms that perform commonly used methods, such as process synchronization. In addition, these pre-existing algorithms should shorten application development time. These algorithms are explained and displayed in the remainder of this section. Unlike common versions of MPI, the more complex functions ending in ”v”, such as `MPI_Gatherv`, were not implemented except for `MPI_Alltoallv`. This was mainly a factor of time, with implementing only those functions specifically needed for common applications. The remainder of this section will briefly describe the implementation of each collective function.

4.5.1 Broadcast

The broadcast function `MPI_Bcast`, depicted in Figure 4.6, sends a copy of a message to every node within a communicator. One central node is designated the root node that performs all of the sends. Due to time constraints and to make implementation simpler, the algorithm simply sends out the messages to each node in a linear fashion. While this could have been improved by having hardware aware code, it was instead chosen to implement the function entirely off of lower level send and receive functionality already in place. One common case optimization was implemented for a broadcast to all nodes within the `MPI_COMM_WORLD` communicator by simply using the hardware broadcast message. However, more care would need to be taken for broadcasts within subset communicators.

Figure 4.6: LlamaMPI Collective Function `MPI_Bcast`

4.5.2 Barrier

During parallel program execution it is often necessary to sync the separate nodes in order to correctly meet timing requirements. This can be accomplished in llamaMPI through use of the `MPI_Barrier` function, depicted in Figure 4.7, which waits until all nodes in a communicator calls it before proceeding. Because this method does not actually deal with passing data at the higher level, the values passed do not actually matter. The function works by waiting for all other nodes to send node 0 within a communicator a dummy value. At this point node 0 broadcasts a dummy value to all other nodes to tell them to proceed.

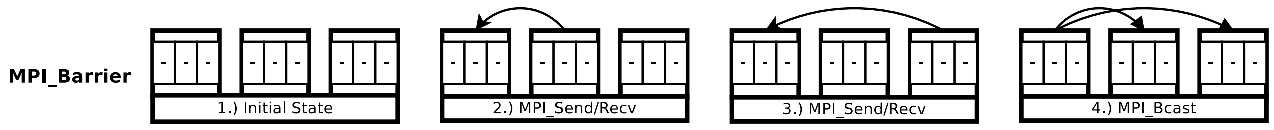

Figure 4.7: LlamaMPI Collective Function `MPI_Barrier`

4.5.3 Scatter and Gather

Another set of collective functions used in the splitting and rejoining of data for parallel processing are the scatter/gather routines, depicted in Figure 4.8. `MPI_Scatter` takes a buffer on a root node and splits it evenly throughout the communicator using individual send and receives. The function `MPI_Gather` performs the inverse by collecting individual messages from smaller buffers on each node in a communicator into a larger buffer on the root node. In both cases the larger buffer is equal to the size of the smaller buffer multiplied by the size of the communicator. Finally, the function `MPI_Gatherall` performs the same function as `MPI_Gather` but then broadcasts the resulting large buffer to all nodes.

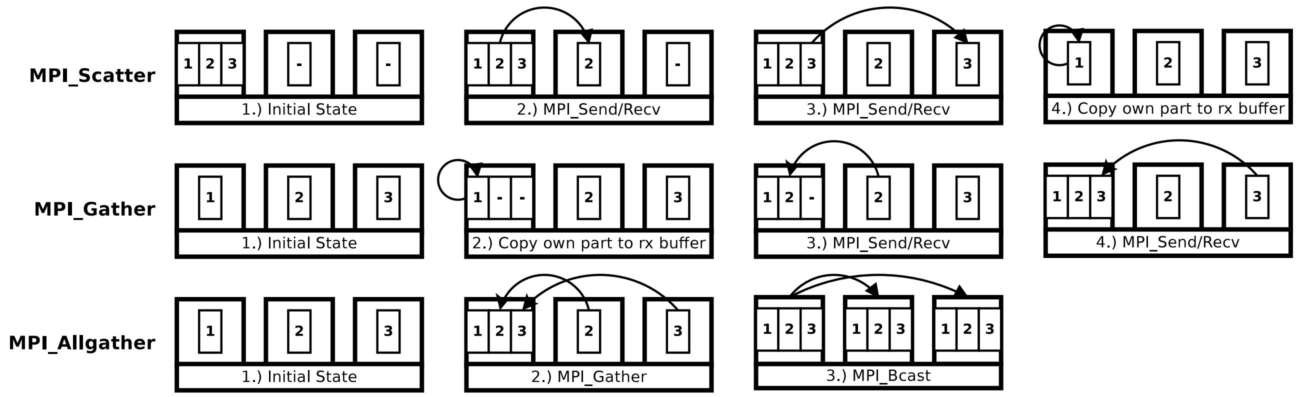

Figure 4.8: LlamaMPI Collective Functions `MPI_Scatter`, `MPI_Gather`, and `MPI_Gatherall`

4.5.4 Reduce

Often it is beneficial to have llamaMPI actually perform certain simple calculations on the data as it gathers back together. These are the reduce sets of functions, which can perform a variety of operations on the data while collecting together messages from all nodes. It is important to note that all implemented operations obey the commutative property, meaning that they can be performed in any order and the result can just be stored in a cumulative receive buffer. The function `MPI_Reduce` first begins by copying the data on the root node into the cumulative buffer. Following that, all nodes linearly send their buffers to the root node to be incremented into the buffer. All of these separate functions with separate data types is implemented through a series of templated files. Another function, `MPI_Allreduce`, is implemented by first calling `MPI_Reduce` and then broadcasting the results out to all nodes. Both these functions are shown in Figure 4.9 with a simple `SUM` operation. The variety of operations available and the data types supported by each are depicted in Table 4.2. Common versions of MPI can support the addition of custom operations. However, this seemed excessive to implement for llamaMPI, at this time.

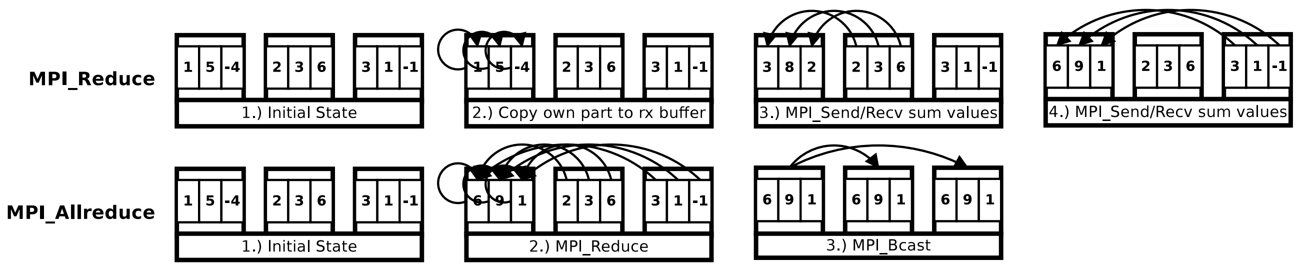
MPI_Reduce
1.) Initial State
2.) Copy own part to rx buffer
3.) MPI_Send/Recv sum values
4.) MPI_Send/Recv sum values
MPI_Allreduce
1.) Initial State
2.) MPI_Reduce
3.) MPI_Bcast

Figure 4.9: LlamaMPI Collective Functions `MPI_Reduce` and `MPI_Allreduce`

Reduce Operation	Supported C Data Types	Supported FORTRAN Data Types
MAX, MIN	Character, Byte, Integer, Float	Character, Byte, Integer, Float
SUM, PROD	Character, Byte, Integer, Float, Complex	Character, Byte, Integer, Float
LAND, LOR, LXOR	Character, Byte, Integer	Byte, Logical
BAND, BOR, BXOR	Character, Byte, Integer	Character, Byte, Integer
MAXLOC, MINLOC	Integer+Integer, Float+Integer	Integer+Integer, Float+Float

Table 4.2: LlamaMPI Reduce Operations and their Supported Types

4.5.5 All to All

The final collective function is `MPI_Alltoall`, which is depicted in Figure 4.10. This function essentially transposes the data buffers between all the nodes of a communicator. All nodes divide their tx buffers into portions in number equal to the size of the communicator. The first node then splits out its tx buffer sending each part to the first part of the appropriate node's rx buffer. The second node follows suit, splitting out its tx buffer and sending each part to the second part of the appropriate node's rx buffer. This continues until all nodes have split out their data in a similar manner. This function can be further complicated through the use of the `MPI_Alltoallv` method (not depicted), which varies the splitting locations for the tx and rx buffers. All of this functionality is efficiently accomplished through the use of low level send and receive calls.

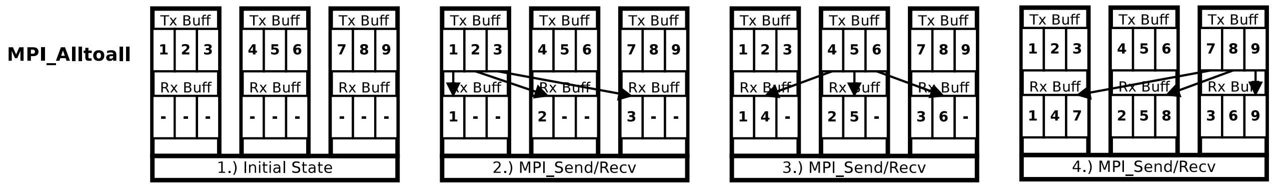

Figure 4.10: LlamaMPI Collective Function `MPI_Alltoall`

4.6 Process Initialization

Even though it is not present in the MPI standard, it is necessary to describe a method of initialization for the llamaMPI processes. This is first achieved by distributing the processes throughout the cluster using a set of Bash scripts. These launch the binaries using SSH and then return data from their output. Next, each node is passed the MAC addresses to all other machines and a unique rank. Finally, all nodes must fully initialize and synchronize before continuing with the application.

4.6.1 Scripts

In order to keep the startup routine simple and flexible, a series of Bash scripts are used to initialize all nodes. Figure 4.11 and the following list describes the necessary steps to prepare Xen for a system of llamaNET nodes operating llamaMPI:

1. It is first necessary to remove any previous instances of llamaNET drivers and llamaNET apps.

2. Next, initialize the llamaNET drivers on each separate machine using SSH.

3. Because llamaNET communication relies on MAC addresses, collect them from each machine using SSH.

4. Create configuration files that contain information on rank, assigned CPU, allocated memory size, and the MAC addresses to each machine. Distribute one of these unique configuration files to each machine node using SCP.

5. Using SSH, start all application nodes between node 0 and node N-2.

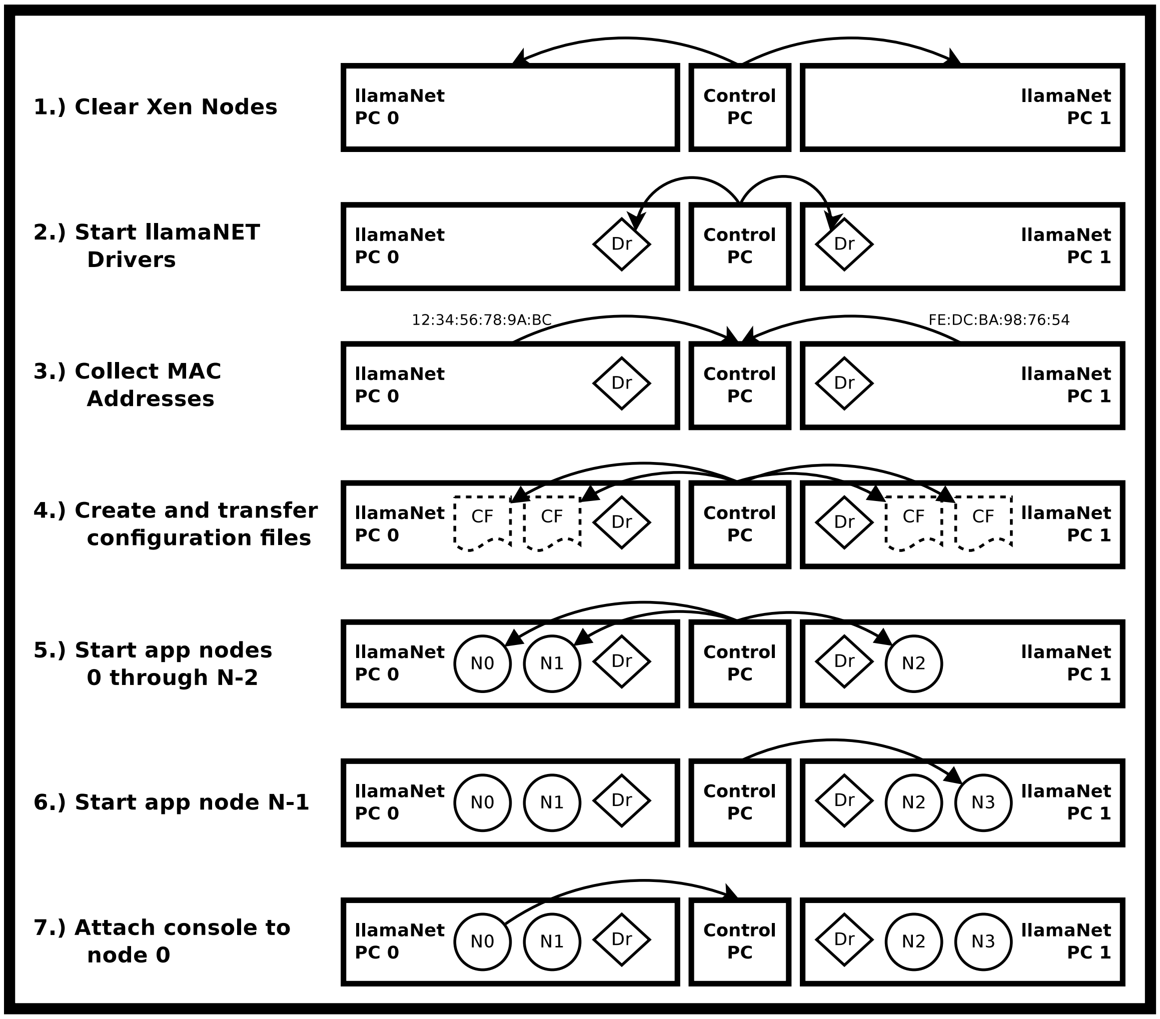
1.) Clear Xen Nodes
2.) Start llamaNET Drivers
3.) Collect MAC Addresses
4.) Create and transfer configuration files
5.) Start app nodes 0 through N-2
6.) Start app node N-1
7.) Attach console to node 0
llamaNet PC 0
Control PC
llamaNet PC 1
Dr
12:34:56:78:9A:BC
FE:DC:BA:98:76:54
CF
N0
N1
N2
N3

Figure 4.11: LlamaMPI Initialization Script

6. Using SSH, start the final application node after a small delay to make certain the other ones have been already initialized.

7. Finally, read the output from the root process node 0 over SHH by attaching to its console.

4.6.2 Startup Code

When an individual node begins it calls `MPI_Init` and executes a series of initialization functions to start the system. First, extract the process' world rank and all other node's MAC addresses from the configuration file. This will allow for the node to form a host table with a MAC address associated with each rank, used in send and receive functions. In addition, the code creates the default groups and communicators, including `MPI_COMM_WORLD`. Next, initialize the llamaNET library to allow for networking. Finally, make sure that all processes start their main code simultaneously, following the process illustrated in Figure 4.12. It begins by waiting for a broadcast from the final nodes. Root node 0 then executes a barrier to ensure that all nodes are synchronized.

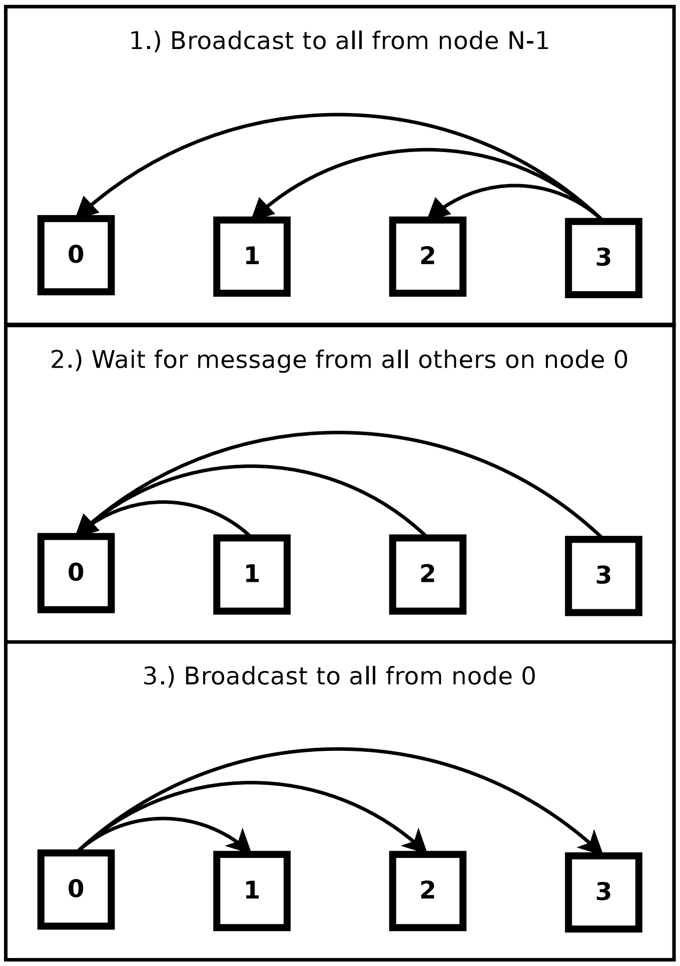
1.) Broadcast to all from node N-1
0
1
2
3
2.) Wait for message from all others on node 0
0
1
2
3
3.) Broadcast to all from node 0
0
1
2
3

Figure 4.12: LlamaMPI Initialization Syncronization

4.7 Additional Optimizations and Tuning

Throughout the development of llamaMPI a variety of features were added to the system in an attempt to both optimize applications and prevent deadlock. However, while some of these features could help under certain circumstances, other times they could hinder a different benchmark's performance. For this reason, the following options were allowed to be determined at compile time using a set of definitions in the global header:

- **Rx Before Tx:** Makes sure that all messages are received from the hardware buffer before attempting to send new ones. This decreases the occurrence of deadlock under some circumstances, especially when all nodes use collective operations to send large messages to one another before receiving. However, this can produce a decrease in performance, as it can cause the nodes to become unsynchronized.
- **Collective Barrier:** This option ensures that all collective operations wait until all nodes in a communicator are synchronized to begin their operations, potentially reducing the occurrence of deadlock. However, this can cause inefficiencies in the program execution, as all nodes are asked to wait for the slowest node to continue, wasting time.
- **Hardware Broadcast:** Enabling this option allows for the optimization of using hardware broadcasts for messages meant for all nodes. This option only has a slight drawback of a small amount of additional logic to detect the broadcast messages. It is exposed as an option in case the cluster on which the llamaMPI is running does not support hardware broadcast. This could be the case if the router blocks all of these types of messages.
- **Max Packet Size:** This option is used to express the largest packet size supported by the underlying communication implementation hardware. Bigger is nearly always better, as it allows for large messages to be broken into fewer chunks. One example of this option would be whether or not Ethernet jumbo frames are supported on a network.
- **Ethernet Flow Control:** This set of options contained in the llamaNET driver startup code directly affects the performance of llamaMPI. For example, it determines how long the hardware receive buffer waits before sending out a halt message to other nodes, telling them to temporarily stop sends.

In addition, these options determine how long these halts last. If the settings are too conservative the performance will suffer. However, a riskier approach could result in deadlock. Finding the correct balance of these settings can mean the difference between a program that cannot run at all and one that runs unbearably slow.

4.8 Comparison with MPICH

This final section explains the main differences between llamaMPI and MPICH, a currently popular implementation of MPI used on clusters running conventional operating systems. It is especially important to understand the differences in MPICH, as it is used for all of the native benchmark comparison tests given later in the results chapter.

4.8.1 Installation

MPICH is installed within a conventional operating system using the normal makefile methods [50]. It is often most convenient to do this across a shared file system so that binaries are accessible in the same location across a cluster. In contrast, llamaMPI is built directly into llamaOS as a supporting static library, depending on many of the other libraries created during the make process. It is intended to only be run inside virtual machine nodes as a llamaNET application.

4.8.2 Compilation

The special compiler commands mpicc, mpif77, and mpif90 are used to compile MPICH applications [51]. This easily allows for the program to build in all the necessary libraries and features needed for a certain environment. On the other hand, llamaMPI applications are built in the same makefile system as the llamaMPI library itself. Because they are statically linked, each application contains the entirety of the MPI code.

4.8.3 Initialization

In order to start a MPICH application the command mpirun or mpiexec is used to distribute the processes throughout the system [7]. This utilizes a machine file, designated by the user with the host names of each physical machine, to launch all processes over SSH. Launching processes in llamaMPI actually involves the creation and synchronization of many virtual machine nodes across physical machines, as outlined in Section 4.6.

4.8.4 Portability

A central goal of the MPICH project was to also create a highly portable implementation of MPI. For this reason, it was built in a modular manor, using the abstract device interface (*ADI*) to allow for the addition of new hardware dependant subsystems [7]. Taking the opposite approach, llamaMPI was specifically designed and optimized to be run on the llamaNET system. However, most of the hardware dependant files are isolated to a few files. For example, collective functions are hierarchically constructed to rely on pre-existing low level send and receive functions. Finally, as explained in Section 2.3, the development of additional network drivers is as easy as the development of application code.

4.8.5 Process Managers

MPICH relies on a series of external process managers and daemons running in separate threads to coordinate much of an application's activity [52]. For example, there is a job scheduler to determine which resources to use during a program's execution. The process manager also exposes a *parallel library* which facilitates communication between processes. Because llamaOS does not support separate threads, all functionality for llamaMPI is built directly into the application code. While this has the potential to decrease the potential latency from interacting with another thread, it raises some buffering issues. For example, a program does not actually pull data from the driver until the `MPI_Recv` function is called. If programs send a large amount of data before receiving anything the buffers may overflow and cause a program crash. This could eventually be fixed by implementing a background thread that pulls messages from the driver regularly.

4.8.6 Collective Operation Optimization

The implementation of MPICH has a multitude of optimizations for collective operations [30]. Studies have been performed to correctly balance the method of message distribution (linear vs. binary tree). In addition, because it utilizes separate resource manager threads, it has the ability to adapt collective algorithms to its needs over time. While llamaMPI also implements some of these optimizations, due to time constraints, further optimization could be warranted. Basic optimizations, such as a hardware broadcast, have already been introduced into the system.

4.8.7 MPI-2 Features

LlamaMPI was mainly built to be compatible with most MPI-1 functionality. However, MPICH has additional features taken from the more recent MPI-2 standard [53]. For example, it supports dynamic process allocation and management at run time. In addition, collective operations have been implemented in an asynchronous manor to allow for nodes to complete their part and not waste time waiting for others. Finally, real time systems are further supported with the addition of priority-based messaging. Despite these absent features, llamaMPI supports the wide range of programs that just use the base MPI standard.

Chapter 5

Applications

5.1 Introduction

When the research presented in this thesis began, the only benchmarking that had been performed on the llamaOS system was just a simple ping-pong latency and bandwidth test. While this showed promising numbers to encourage further development, much more testing was warranted. For this reason, an additional goal of this research was to build a set of applications for llamaOS to more thoroughly test the system. The *NAS Parallel Benchmarks* proved to have enough variety to test both the computational and communicational capabilities of the system. Finally, with the decrease in networking latency, it was predicted that the performance of fine-grained PDES computations, such as WARPED would perform well in the system. This chapter deals with the process of integrating both of these applications into the llamaOS and llamaMPI systems.

5.2 LlamaOS Improvements

While llamaOS provided a great foundation upon which to construct a low latency parallel programming interface, a few changes needed to be made at the driver level to allow applications to run optimally. For example, additional buffering needed to be added to the network driver to allow for larger messages to be sent. In addition, the programs needed an efficient method to import data from an existing file system. This section shows these two main changes to the llamaNET driver and the file system management.

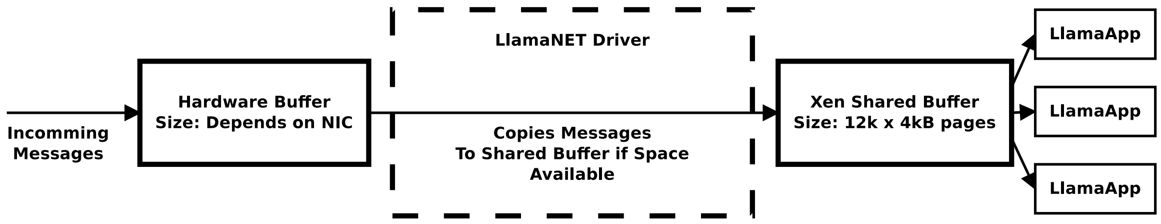
LlamaNET Driver
Incomming Messages
Hardware Buffer Size: Depends on NIC
Copies Messages To Shared Buffer if Space Available
Xen Shared Buffer Size: 12k x 4kB pages
LlamaApp
LlamaApp
LlamaApp

Figure 5.1: Original LlamaNET Driver Buffering Method

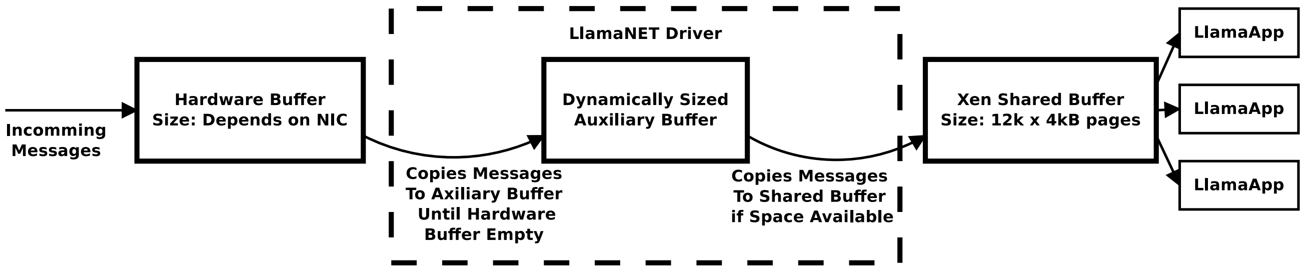
LlamaNET Driver
Incomming Messages
Hardware Buffer Size: Depends on NIC
Dynamically Sized Auxiliary Buffer
Copies Messages To Axiliary Buffer Until Hardware Buffer Empty
Copies Messages To Shared Buffer if Space Available
Xen Shared Buffer Size: 12k x 4kB pages
LlamaApp
LlamaApp
LlamaApp

Figure 5.2: New LlamaNET Driver Auxiliary Buffering Method

5.2.1 LlamaNET Driver

When initial testing of llamaNET began with the NetPIPE benchmark, only small messages were sent in a very regular ping-pong pattern. This required only minimal buffering at the hardware and software levels. However, when expanding to more irregular applications with larger messages, such as those featured in this chapter, it became necessary to improve the buffering method used in the driver. Initial tests of the *NAS Parallel Benchmarks* would not even complete because they dropped messages in transit to other nodes when the buffers overflowed. This was somewhat fixed when a hardware feature of Ethernet was implemented, essentially sending out a broadcast distress signal from a machine whose buffer was almost full. This stopped all other machines from sending messages over the network to allow time for the almost full machine to process its messages. However, this did not necessary prevent messages from being dropped, as messages already in the send queue of a NIC were sent, despite the distress message. In addition, the time when the distress message was in effect greatly wasted network latency and bandwidth by blocking all messages from being sent.

Due to this inefficiency, research was then moved to expanding the llamaNET software shared receive buffers to allow for more time before the hardware ones filled to the point of distress. This original driver model is depicted in Figure 5.1. However, Xen limited the amount of shared 4kB pages of memory to around twelve thousand, which was still not enough for some of the larger collective messages in the applications. For this reason, an additional auxiliary buffer was dynamically allocated inside the llamaNET driver program space, as depicted in Figure 5.2. While this slightly increased the latency with an additional copy, it allowed for almost any reasonably sized application to complete. In addition, this latency increase is rather small because almost all memory copies are now accelerated in hardware using DMA. Finally, because the auxiliary buffer is dynamically allocated, it can be expanded on an application by application basis by increasing the amount of memory allocated to the driver by Xen.

5.2.2 LlamaOS File Management

Another feature that needed to be added to llamaOS to run more advanced applications was some sort of file system. When development began on llamaMPI there was no file management present in the operating system. This limited application development, as all configuration details either had to be passed to the program through the command line parameters or built directly into the binary. In addition, there was a size limit on the number of characters able to be passed to a program over the command line arguments, meaning that the host table passed to llamaMPI was limited to only about four machines.

Because it would likely take quite an effort to implement a full file system, a temporary solution was developed to allow for configuration files to be easily passed to the applications. To achieve this, a simple single file system was created that allowed one file to be placed per virtual disk. The file would then just be accessed by its name using the standard library file I/O interface. These files were enumerated and assigned to unique disks in the Xen startup config files. Even though this provided a basic interface to allow for the importing of configuration files, it had the following limitations:

- All files must be statically declared and are loaded into memory at startup.
- Files are read-only and new ones cannot be created.
- It is a completely flat file system without directories.
- Files have to be padded to 512 byte segments.

5.3 NAS Parallel Benchmarks

The *NAS Parallel Benchmarks* were chosen to test a variety of different parts of the system. As explained in Section 2.5, the suite contains several benchmarks ranging from those that are computationally heavy, to those that require heavy communication bandwidth. It should be noted, however, that none of the benchmarks particularly benefit from having lower network latency. Instead, they test a wide range of MPI collective and point-to-point communications.

The version of the benchmarks used for the tests found in this thesis was NPB3.2-MPI. This allowed for easy compilation with the existing llamaMPI system. The only slight alteration that had to be made was in the makefile system in order to build each of the benchmarks individually. In addition, a parameter file was included to allow for different sizes and classes of the benchmarks to be interchanged equally. Each build of the llamaOS system just created one size and class version of each benchmark.

Of the nine potential benchmarks, seven were chosen to be used in tests. The first one that was chosen to be excluded was the BT benchmark, as it required a large amount of file system I/O to run correctly. Because the new file system could only statically read files during the program startup, this benchmark was determined to not be suitable for our system. In addition, the DT benchmark solely tested bulk data transfer and required an increasing number of nodes for each class test. These large data transfers are still not suitable for our system, even after the driver update, and can result in the benchmark crashing. Furthermore, DT has a completely different test pattern than the other benchmarks, as the number of nodes is completely determined by the class. Finally, it was determined that the more hybrid communication and computation tests would provide more information about the system than a singularly communicational one.

5.4 WARPED

The WARPED parallel discrete simulator was chosen as a case study for llamaMPI to determine the effectiveness of the decreased system latency. Because WARPED often contains simulations with many fine-grained processing steps, the lower latency of the networking system in llamaNET should greatly improve performance. In addition, because llamaOS contains bare-minimum optimized operating system calls, some of the computational sections of the code should be made even more efficient.

In order to perform the experiments shown in this thesis, the most recent version of WARPED was cloned from its Git repository, at `github.com/wilseypa/pdes`. The compilation of the applications again proved straightforward, as one of the supported communication protocols was MPI. However, it was necessary to remove some currently unsupported features in llamaOS from the WARPED source code, such as threads, TCP/IP communication, and the logging of progress through file I/O. In addition, it was necessary to use the new single file system to import the WARPED and simulation configurations. Finally, the makefile system had to be altered to support the building of three separate binaries for each simulation, namely *PHOLD*, *RAID*, and *SMMP*, explained in the remainder of this section.

5.4.1 PHOLD

The PHOLD simulation model contains a predetermined number of processes that send events to one another. It is a purely abstract model used to test PDES systems that does not really have a physical equivalent. Once the simulation begins, the processes begin sending messages to each other with messages having randomly assigned destinations and timestamps during which to be sent, as depicted in Figure 5.3. The user has the ability to completely configure the random distribution, the number of the messages to be sent, and the connectedness of the processes. An increased connectivity results in a larger amount of rollbacks.

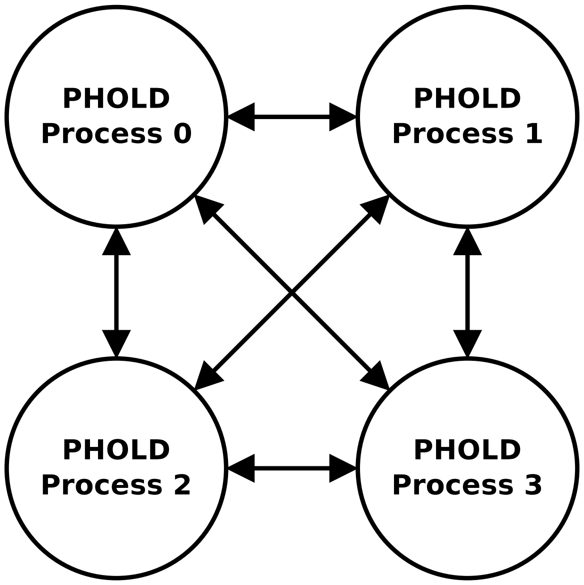

Figure 5.3: PHOLD Simulation Example Setup

5.4.2 RAID

Redundant Array of Independent Disks (RAID) is used to connect a set of hard drives together to increase the security and speed of the drives through shared data. The RAID model defined in WARPED simulates a level 5 RAID system. The PDES logic processes are divided between the request source processes, RAID controlled, and the hard disk drives, as depicted in Figure 5.4. The sources create request messages that are sent to a particular RAID controller, which then forwards the request to a specific disk. The disk then returns a data message back to the original source. Because RAID controllers can have multiple sources and destination disks, and disks can have multiple data destination files, this results in a highly connected simulation. This results in a simulation containing many rollbacks and very much effected by latency.

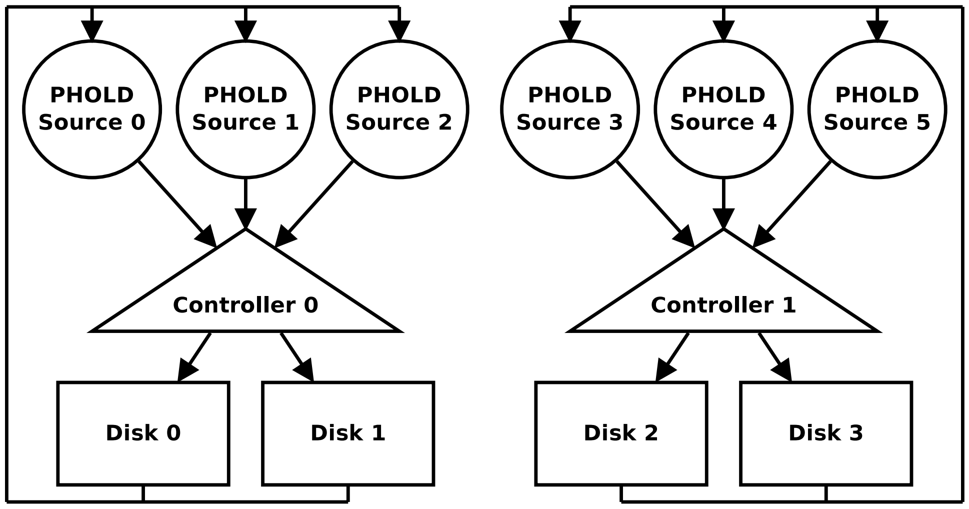

Figure 5.4: RAID Simulation Example Setup

5.4.3 SMMP

The final simulation, SMMP, models a shared memory multiprocessor system. Like most modern multiprocessors, each CPU contains its own individual cache memory, as well as having access to a central shared main memory, as depicted in Figure 5.5. The processor makes a request to the memory system that is either fulfilled by the cache or results in a cache miss and has to go to the main memory. The user is allowed to configure the number of memory requests to be made, the percent of time that a cache miss occurs, and the number of processors and caches. Most memory requests can be fulfilled without actually using the communication system. However, during all cache misses, it is necessary to communicate with the main memory, which is connected to all processes and may result in a rollback. Because of this, the amount of rollbacks is directly related to the percent of cache misses chosen by the user.

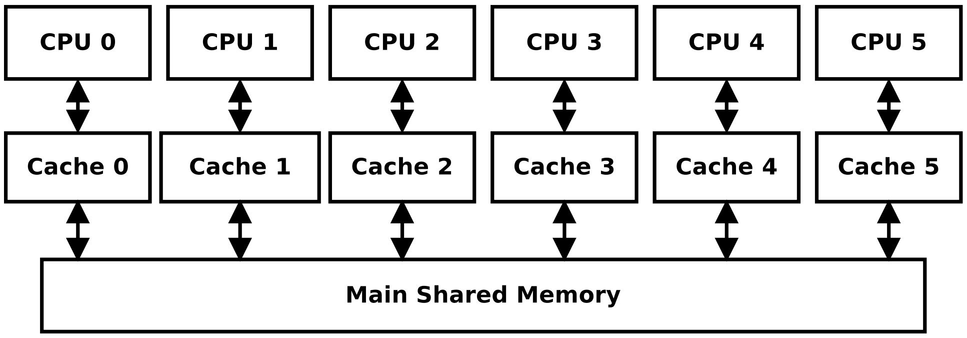

Figure 5.5: SMMP Simulation Example Setup

Chapter 6

Performance Analysis

6.1 Introduction

This chapter gives the analysis performed on the new llamaMPI system through the *NAS Parallel Benchmarks* and the WARPED parallel discrete event simulator. It describes the particular llamaMPI parameters used for every simulation, as well as the hardware test setups on which they were performed. The results sections are split between the two major applications and then further divided by the hardware on which they were performed. It should be noted that scripts were designed and used to run all test sets to reduce possible user error.

6.2 Hardware Setups

The experiments detailed in this chapter were run on two different hardware platforms. Setup 1, shown in Table 6.1, contains only two nodes. They both contain extremely strong specifications, with eight core processors and 16GB of memory. On the other hand, Setup 2 is a larger Beowulf cluster with 20 nodes available for testing, as shown in Table 6.2. However, each individual machine has much lower specifications than in Setup 1, with two separate dual core processors running at lower clock rates and 4GB of memory.

The use of two different setups for the tests provides three advantages. Firstly, it shows the ease with which the llamaMPI system can be ported between different hardware platforms. In addition, it allows for the results to demonstrate that llamaMPI can run successfully on a variety of hardware platforms. Finally,

Processor	AMD FX-8120 eight cores at 3.10GHz
Motherboard	Gigbabyte GA-990FXA-UD5
Memory	16GB RAM
Network Card	Intel Gigabit PCIe 82574
Network Switch	Netgear Prosafe GS108T
`Dom∅` Operating System	Gentoo Linux 3.8.13
Xen Version	4.2.0

Table 6.1: Setup 1: 2 Nodes with Large RAM and High CPU Performance

Processor	(x2) Intel Xeon CPU 5130 dual cores at 2.00GHz
Motherboard	ASUSTek DSBV-DX/SAS
Memory	4GB RAM
Network Card	Intel Gigabit 80003ES2LAN
Network Switch	HP ProCurve Switch 2800 Series
`Dom∅` Operating System	Ubuntu Linux 12.10
Xen Version	4.2.0

Table 6.2: Setup 2: 20 Node Beowulf Cluster

it is desirable to run tests that both utilize many processes on a single machine and use many processes overall. Each application node takes up a decent amount of memory, so it is difficult to run more than one node per physical machine in Setup 2, due to its restrictive memory size. However, Setup 1 is the opposite with a restrictive number of machines with high memory. Because of this, tests aimed at using multiple nodes per machine are performed on Setup 1, while tests using individual processes across many machines are performed on Setup 2.

6.3 LlamaMPI Parameters

The llamaMPI parameters, as defined in Section 4.7, are given below in Table 6.3. These were used universally for all tests and only differed in packet size between hardware setups. Overall, the settings were chosen to minimize latency to the hardware's best ability. This was the main reason for turning off the *Rx Before Tx* and *Collective Barrier* options, as they increase latency. *Hardware Broadcast* was left off as it was not supported on Setup 2 and Setup 1 was kept consistent. Jumbo Ethernet frames were not supported on Setup two, so a smaller packet size was used. The Ethernet flow control was chosen to not be too conservative to affect latency, but to allow for the benchmarks to complete. Ideally the new auxiliary buffer in the driver should prevent the distress signal from occurring too frequently, if ever. Finally, a large amount of memory was allocated to the driver to support the auxiliary buffer, leaving enough RAM to allow for all types of applications to run alongside it.

Parameter	**Value**
Rx Before Tx	Off
Collective Barrier	Off
Hardware Broadcast	Off
Max Packet Size (**Setup 1**)	4032 Bytes
Max Packet Size (**Setup 2**)	1436 Bytes
Ethernet Flow Control	20kB Threshold
Driver Memory	1 Gigabyte
App Memory	2 Gigabytes

Table 6.3: LlamaMPI Test Parameters

6.4 NAS Parallel Benchmarks

As explained in previous chapters, the *NAS Parallel Benchmarks* were chosen to test a variety of features in the llamaMPI system. Some of the benchmarks depend heavily on network performance, while others rely more on computational speed. In addition, the more network heavy ones can vary between using point-to-point communication or collective functions. However, the messages sent are often very large in size and therefore often depend more on bandwidth than latency. Because llamaNET only performs at native bandwidth levels, it is expected that the results will show about equal performance against native. These benchmarks are still beneficial to perform, as they perform a self-check at the end to verify their results. This allows for the NAS Benchmarks to essentially be used as a way to verify the correctness of llamaMPI.

6.4.1 Benchmark Parameters

The *NAS Parallel Benchmarks* are required to be recompiled for every different size and class variation. This means that configuration of the benchmarks is entirely performed at compile time and relies on the makefile parameter definition file. Although it is also possible to load each benchmark with a certain set of data, it was instead determined that the compiled in defaults would be used. This seemed to provide a more standard and easy to perform test, given the early absence of a file system in llamaOS. The selected sizes and classes for each benchmark are clearly shown next to every test in the results tables.

6.4.2 Results from Setup 1

As predicted, the NAS Benchmarks mostly performed at close to native times, as shown in Table 6.4. The tests shown were run on setup 1, which allowed for multiple processes to be run on each physical machine. Because the benchmarks all send large amounts of data, the total number of processes was kept fairly small (2 or 4). For each benchmark and size combination, an attempt was made to increase the class size all the way to size C. While this was possible for a few, many benchmarks were not able to run with the great increase in message sizes, causing buffers to overflow and the program to run out of memory. All tests evenly divide the number of nodes between the two machines.

Name	Class	Size	LlamaMPI Time (s)	LlamaMPI Mop/s	Native Time (s)	Native Mop/s	LlamaMPI / Native Time	LlamaMPI / Native Mop/s
cg	A	2	2.72	549.3	2.79	535.51	97.5%	102.6%
cg	A	4	1.58	949.26	2.11	707.59	74.9%	134.2%
cg	B	2	110.3	495.98	114.8	476.57	96.1%	104.1%
cg	B	4	62.76	871.64	76.35	716.56	82.2%	121.6%
cg	C	4	160.71	891.97	189.18	757.71	85.0%	117.7%
ep	A	2	16.53	32.47	17.18	31.25	96.2%	103.9%
ep	A	4	8.32	64.51	8.67	61.93	96.0%	104.2%
ep	B	2	66.12	32.48	66.26	32.41	99.8%	100.2%
ep	B	4	33.28	64.53	33.81	63.51	98.4%	101.6%
ep	C	2	264.52	32.47	270.81	31.72	97.7%	102.4%
ft	W	2	0.83	447.8	0.94	395.72	88.3%	113.2%
ft	W	4	0.54	695.45	0.54	689.25	100.0%	100.9%
ft	A	2	16.1	443.38	15.08	473.38	106.8%	93.7%
is	W	2	0.16	67.06	0.16	64.2	100.0%	104.5%
is	W	4	0.14	74.83	0.13	83.68	107.7%	89.4%
is	A	2	1.28	65.79	1.31	64.02	97.7%	102.8%
is	B	2	7.14	47.02	5.41	62.01	132.0%	75.8%
lu	W	2	28.49	633.89	30.89	584.76	92.2%	108.4%
lu	W	4	16.22	1113.39	16.77	1077.21	96.7%	103.4%
lu	A	2	182.76	652.75	184.94	645.06	98.8%	101.2%
mg	A	2	6.92	562.86	7.28	534.33	95.1%	105.3%
mg	A	4	3.87	1004.9	4.1	948.36	94.4%	106.0%
mg	B	2	32.52	598.54	34.14	570.05	95.3%	105.0%
mg	B	4	18.32	1062.1	19.38	1004.4	94.5%	105.7%
sp	W	4	23.26	609.49	22.98	616.7	101.2%	98.8%
sp	A	4	122.7	692.85	126.12	674.03	97.3%	102.8%
sp	B	4	545.59	650.69	560.35	633.55	97.4%	102.7%

Table 6.4: NAS Parallel Benchmarks Results on Setup 1

A detailed explanation of each benchmark result is given below:

- **CG**: This benchmark experienced the largest amount of speedup when ported to llamaMPI, with `cg.C.4` running with 117.1% native Mop/s. In addition, the system was able to handle class C with 4 nodes and class B with 2 nodes. This is likely due to the communicational structure of the cg benchmark, which relied on irregular long distance communication that benefits from lower latency.
- **EP**: The embarrassingly parallel benchmark had a slight amount of improvement, with the `ep.C.2` test completing in 97.7% the time of native. In addition, it was able to reach class C with 2 nodes and class B with 4. As this test purely tested computational power, it was expected to see little improvement. The slight runtime decrease is likely due to the more efficiently optimized operating system calls.
- **FT**: The Fourier transform benchmark mainly relied of bulk data transfers and performed slightly worse on llamaMPI, with 93.7% the Mop/s in the `ft.2.A` test. This was likely due to not fully optimized collective functions and would likely improve with algorithm changes. In addition, it was only able to complete the test at class A. This is reflective of the issues of bulk data transfers still present in the system.
- **IS**: The integer sort benchmark also relied heavily on large collective data transfers and saw the worst performance. The `is.B.2` test resulted in a run time that was 132.0% that of native. As with the FT benchmark, this was likely due to poor collective functionality and the handling of bulk transfers. It was also only able to reach class W with 4 processes, but class B with 2 processes. This likely points to the large amount of data tha needed to be passed between the nodes, often overflowing the buffers.
- **LU**: This benchmark experienced results close to native, with `lu.A.2` completing in 98.8% the amount of time. This is most likely due to the balanced nature of the benchmark, relying equally on computational and communicational power. However, it also seemed to utilize large bulk transfers, again resulting in a maximum class size of A for 2 nodes and B for 4 nodes.
- **MG**: The MultiGrid benchmark performed fairly well on the llamaMPI system, with the `mg.B.4` test experiencing 105.7% the amount of Mop/s as native. In addition, both node sizes of 2 and 4 were able

to run with the class size B. This increase was mostly attributable to improvements in the operating system calls, as it is primarily computationally bound.

- **SP**: The final benchmark, scalar pentadiagonal, could only be run with 4 nodes, as the algorithm only supports square numbers of processes. It performed essentially close to native, with the `sp.B.4` test completing in 97.4% the time. In addition, the test successfully ran for the longest time on class B with nine minutes of constant operation. This was likely due to a fair balance between computation and communication that never caused the buffers to overflow.

Overall, the majority of the *NAS Parallel Benchmarks* performed adequately, with only the FT and IS benchmarks under-performing native. As expected, most of the benchmarks did not see excessive increases in performance, as they mainly relied on bandwidth rather than the improved latency. Nevertheless, all of the benchmarks that completed were able to verify successfully, proving the stability of the llamaMPI system. Further improvements to the collective operations and buffering scheme should eventually allow for more benchmarks to complete and operate with better performance.

6.5 WARPED

WARPED was selected as an appropriate case study application before llamaMPI had even begun development. When the initial results came back showing llamaNET as having an order of magnitude lower latency than native it was determined that WARPED would benefit greatly from being ported to the system. Most parallel discrete event simulations contain fine-grained computational steps and rely on lower latency to limit the number of causality errors and subsequent rollbacks. The WARPED communication interface is implemented solely as MPI non-blocking point-to-point sends and receives. Ideally, this means that there should be little overhead present from the llamaMPI implementation and the application should benefit from the lower latency numbers shown in initial testing.

6.5.1 Simulation Parameters

Unlike the *NAS Parallel Benchmarks*, WARPED requires a set of two configuration files to be passed to each simulation at launch. The first configuration file is universally used in all the different types of simulations to determine which features of the WARPED simulator should be used. To keep tests simple and consistent, the same settings were used between all simulations and native and llamaMPI tests. These configurations are detailed below in Table 6.5.

Parameter	**Value**
Simulation	TimeWarp
Scheduler Type	MultiSet
Scheduler ScheduleQScheme	MULTISET
Scheduler CausalityType	STRICT
Scheduler ScheduleQCount	1
EventList Type	MultiSet
CommunicationManager PhysicalLayer	MPI
CommunicationManager Type	Default
StateManager Type	Periodic
StateManager Period	10
OutputManager Type	Aggressive
OutputManager AntiMessages	Default
OutputManager FilterDepth	16
OutputManager AggrToLazyRatio	0.5
OutputManager LazyToAggrRatio	0.2
OutputManager ThirdThreshold	0.1
GVTManager Type	Mattern
GVTManager Period	1000
DVFS Manager Type	None

Table 6.5: WARPED Configuration

The second needed configuration file defines the setup of the individual simulation. This file can define options ranging from the number of processes to the probability distribution used and requires a unique parameter set for each simulation type. Again, to keep tests simple and consistent, the three configuration files used for PHOLD, RAID, and SMMP were kept identical between different simulations on both native and llamaMPI. Table 6.6 gives the parameters used for the PHOLD simulation, providing a large number of processes and exponential distribution, sure to result in many rollbacks. Table 6.7 defines a large number of highly connected disks, controllers, and sources for the RAID simulation. Each controller, or fork, is connected to 12 source inputs and 8 disk outputs, which will likely result in many rollbacks. Finally, the SMMP configuration is given in Table 6.8, simulating eight processors all competing for main memory with a 0.85 cache hit ratio. Because this hit ratio is not very high, this should result in many shared main memory accesses and rollbacks.

Parameter	Value
Number of Processes	200
Message Density	20
Distribution	Exponential
Seed	1.0
State Size	1024
Computation Grain	1 second
Number of Outputs	20

Table 6.6: PHOLD Configuration

Parameter	Value
Number of Disks	32
Disk Type	FUJITSU
Number of Processes (Sources)	96
Max Number of Requests	20000
Number of Forks (Controllers)	8
Number of Process Inputs	12 each
Number of Fork Output Disks	4 each
Number of Start Disk	1

Table 6.7: RAID Configuration

Parameter	Value
Number of Processors	8
Speed of Cache	10
Cache Hit Ratio	0.85
Speed of Main Memory	100
Number of Requests Per Processor	400000

Table 6.8: SMMP Configuration

6.5.2 Results from Setup 1

As expected, all experiments on setup 1 showed a performance boost when using llamaMPI versus native, as shown in Table 6.9. All three WARPED simulations were run with both 2 and 4 processes evenly distributed across the two machines. In addition, a reasonable end time was chosen for each simulation to allow for the quick testing of different configurations while providing enough timing resolution to accurately determine the difference between llamaMPI and native. It should be noted that the number of rollbacks that occurred did not directly correlate with the amount of time a simulation took. This is because the time is solely determined by the speed of the critical path. Having many deviations of the critical path may not be detrimental, if the deviants are rectified quickly. It should be noted that the times and number of rollbacks presented in this and the following sections are relative to node zero. An analysis of each simulation's results is given below:

- **PHOLD:** This simulation experienced a greater number of rollbacks, but much better performance under llamaMPI. With 2 nodes it ran in 85.2% native time and with 4 nodes it ran in 84.8% native time.
- **RAID:** The RAID simulation had asimilar number of rollbacks and also increased performance significantly. With 2 nodes it ran in 81.7% native time and with 4 nodes it ran in 86.9% native time.
- **SMMP:** This simulation had a much smaller number of rollbacks than native and achieved the most dramatic speedup on setup 1 with 4 nodes completing in 75.8% native time. With 2 nodes it completed in 87.8% native time.

Name	Size	End Time	LlamaMPI Time (s)	LlamaMPI Rollbacks	Native Time (s)	Native Rollbacks	LlamaOS / Native Time
phold	2	1000	23.0	5596	27.0	353	85.2%
phold	4	1000	13.5	606	15.9	497	84.8%
raid	2	100000	38.8	51033	47.5	18936	81.7%
raid	4	100000	18.1	22625	20.8	22754	86.9%
smmp	2	None	38.2	292074	43.8	337923	87.3%
smmp	4	None	29.9	276110	39.4	412854	75.8%

Table 6.9: WARPED Case Study Results on Setup 1

6.5.3 Results from Setup 2

The experiments on setup 2, the Beowulf cluster, attempted to increase the number of nodes to a higher value. Due to the low memory constraints of the system, it was decided that only one application node would be run per physical machine. In addition, only the PHOLD and SMMP simulations were performed, as the RAID model would not complete with larger than two nodes on the cluster. This was potentially caused by either the small memory limit or overflowing of the buffers, due to the large amount of communication in the RAID model. Overall, both simulations achieved around a 60% - 70% of native execution time.

Table 6.10 gives the results for the PHOLD simulation, run between 2 and 10 processes. In nearly all cases the llamaMPI run time was around 60% that of native. The most dramatic improvement of all experiments in this thesis occurred when running PHOLD on 5 cluster nodes, resulting in the llamaMPI version completing in 55.1% of native time. The worst case occurred when running on two nodes, which only resulted in a 66.9% native time execution. It should also be noted that the number of rollbacks did not correlate with either the percent native time or the communication implementation. Instead, they seem to appear randomly in each simulation. A more telling value to use would have been the total duration of rollbacks, but this was not available. As predicted by Amdahl's Law, both the native and llamaMPI speedups seemed to decrease in an exponential curve, as seen in Figure 6.1. This is because the serial part of the simulation was beginning to dominate, and adding additional nodes did not help. Figure 6.2 depicts the percentage of native run time, in which each number of nodes resulted.

Size	**End Time**	**LlamaMPI Time (s)**	**LlamaMPI Rollbacks**	**Native Time (s)**	**Native Rollbacks**	**LlamaOS / Native Time**
2	2000	51.2	1705	76.6	5456	66.9%
3	2000	36.1	549	62.0	3191	58.3%
4	2000	28.4	1430	50.3	7588	56.5%
5	2000	23.6	1097	42.8	2586	55.1%
6	2000	20.7	950	36.0	1354	57.4%
7	2000	19.1	1780	33.6	191	56.8%
8	2000	22.5	7463	34.0	5559	66.1%
9	2000	20.0	6202	30.5	4087	65.6%
10	2000	18.5	6865	31.6	5962	58.3%

Table 6.10: WARPED PHOLD Results on Setup 2

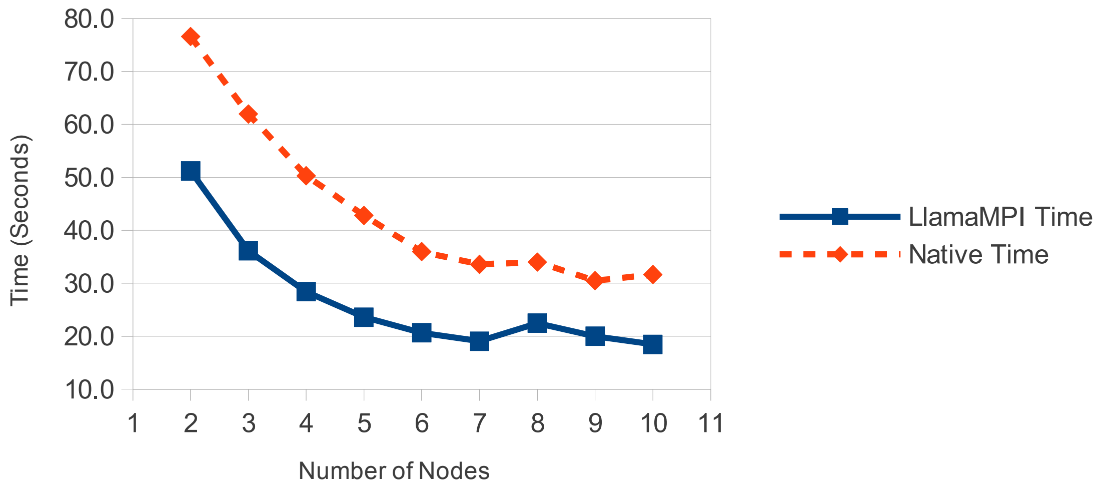

Figure 6.1: Plot of WARPED PHOLD Times from Setup 2

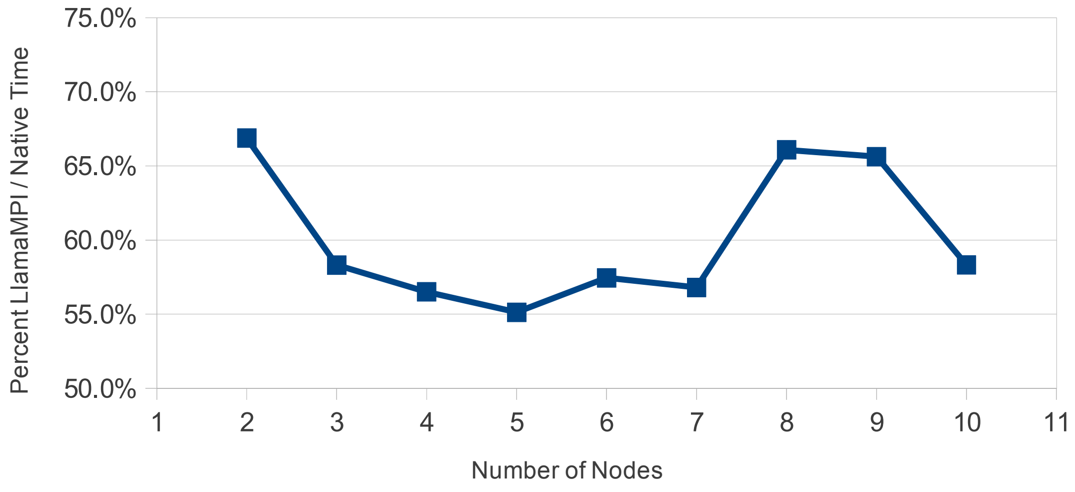

Figure 6.2: Plot of WARPED PHOLD Time Percentages from Setup 2

The results for the SMMP simulations on setup 2 are outlined in Table 6.11, run between 2 and 10 processes. In all the cases presented the percent native time was reduced to around 70%. The minimum achieved percent native time was 66.2%, found when running on 7 processes. As in the PHOLD simulation, the number of rollbacks seemed to fluctuate randomly during each simulation. Furthermore, as predicted, both the native and llamaMPI speedups decreased in an exponential curve, as seen in Figure 6.3. Figure 6.4 shows the percent of native runtime for each process setup. Overall, the percent native plot fluctuates only +/- 5% from the 70% average, showing that the advantage of using llamaMPI depends more on the simulation model used than the number of cores.

Size	**End Time**	**LlamaMPI Time (s)**	**LlamaMPI Rollbacks**	**Native Time (s)**	**Native Rollbacks**	**LlamaOS / Native Time**
2	None	41.7	199873	59.0	228612	70.7%
3	None	36.9	8992	51.8	5267	71.3%
4	None	33.8	245218	50.5	251306	67.0%
5	None	34.8	25247	48.8	14511	71.4%
6	None	33.9	19168	47.8	10461	70.9%
7	None	32.5	14625	49.1	7051	66.2%
8	None	31.7	307044	46.5	278828	68.2%
9	None	31.8	77987	43.7	46495	72.9%
10	None	31.9	78213	43.4	46647	73.5%

Table 6.11: WARPED SMMP Results on Setup 2

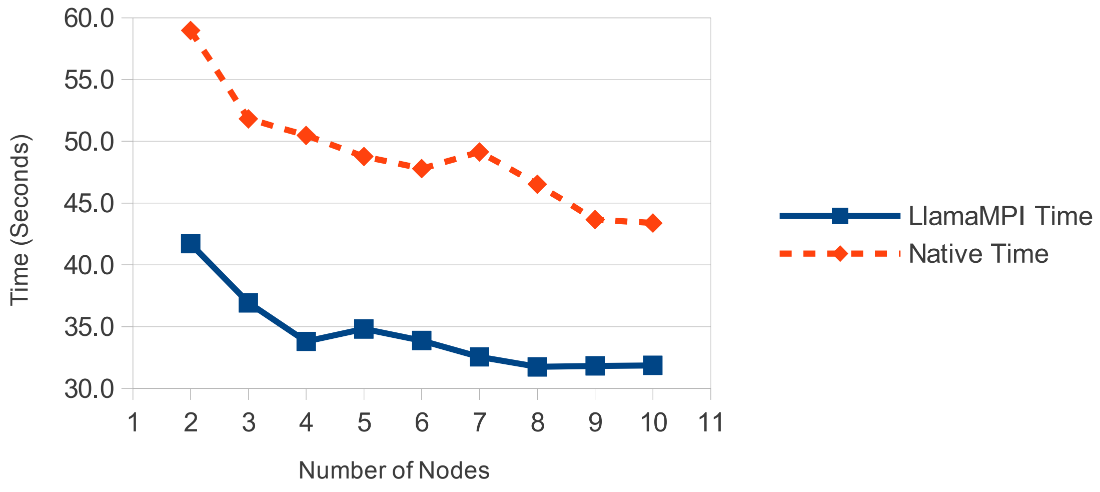

Figure 6.3: Plot of WARPED SMMP Times from Setup 2

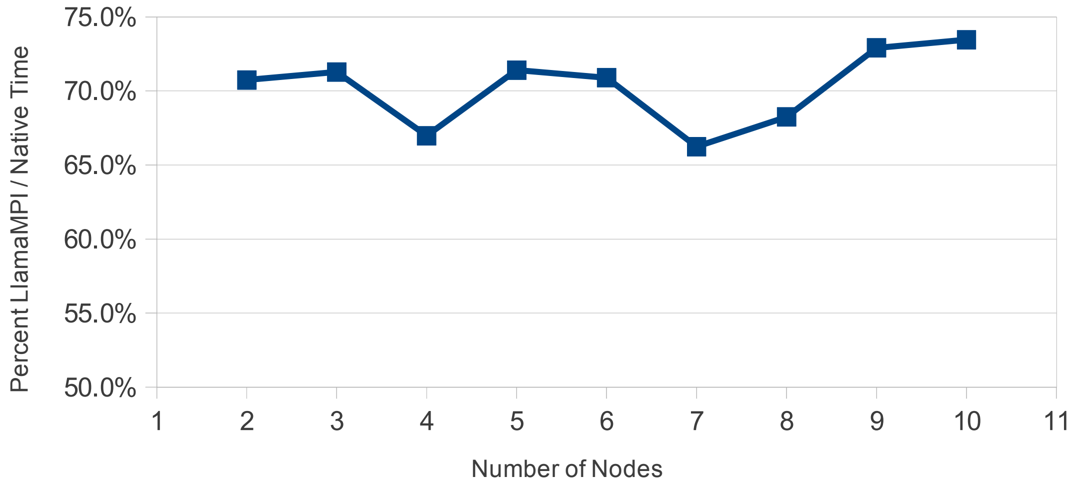

Figure 6.4: Plot of WARPED SMMP Time Percentages from Setup 2

6.6 Conclusion

The results in this chapter show that the llamaMPI system is performing well in almost all cases of the benchmarking tests and case study. Nearly all of the *NAS Parallel Benchmarks* ran at speeds equivalent to a native system, as expected. This also served to essentially verify most operations available in llamaMPI. In addition, the implementation of the WARPED simulator into the system showed significant gains in performance with the decrease in network latency and optimized operating system calls. While the improvements are fairly striking, there is much room for improvement in the flexible llamaMPI system, as explained in the concluding Chapter 7.

Chapter 7

Conclusions and Suggestions for Future Research

7.1 Summary of Findings

While llamaOS provided the basic foundation for a new flexible, low-latency operating system, llamaMPI now provides an efficient networking standard to promote future development on the system. After realizing that porting was not a possibility, llamaMPI was designed from the ground up to provide a messaging system with low overhead, without sacrificing the core functionality. The new system has been verified using the *NAS Parallel Benchmarks* and has been shown to run at native or better speeds when bandwidth is the main limiting factor. The benchmarks that run slower or do not complete show the room for development in collective functionality and buffer flow design. Finally, the WARPED case study illustrates the main benefit of llamaMPI, namely its order of magnitude lower latency than native TCP/IP. Because PDES simulations usually rely heavily on small fine-grained computational steps, a decrease in latency can drastically reduce the amount of time spent off the critical path caused by causality errors. This is evidenced in the results with an average decrease of 30% to 40%, when compared with native.

7.2 Detailed Conclusions

The research presented in this thesis produced the originally expected results; the WARPED system saw significantly improved performance with an overall decrease in latency, due to llamaMPI. This verifies the idea that the performance of fine-grained applications, such as WARPED, have much to benefit from greatly decreased latency. However, other factors, such as network buffering performance and process scheduling were also key contributors. For example, before adding in an extra auxiliary buffer to the driver, the performance of llamaMPI suffered and became even worse than native. In addition, the discrepancy between inter-machine and intra-machine communication latency likely still caused many performance losses from communication. Improvements may need to be made in these other areas to further minimize the time spent off of the critical path in simulations.

Even though the results mostly attained the desired goal of lowering latency, there are a few facets of the research process that could have been altered to potentially improve performance and minimize development time. Firstly, while the *NAS Parallel Benchmarks* were useful in the verification of the system, they were poorly matched to the ultimate goal of decreasing latency for fine-grained applications. They were closer to large scientific calculations and required large amounts of bandwidth and buffering, that if nothing else, uncovered several issues in the system that would have been difficult to find later. Another set of benchmarks could have been potentially chosen to both discover these errors and provide a more meaningful set of results. However, researchers in the HPC community seem particularly interested in the NAS benchmarks for comparison, ultimately making them the chosen candidate.

Another change that could have potentially boosted the performance of the system would have been to implement the MPI communication interface in a background thread or the llamaNET driver. Even though implementing llamaMPI directly alongside the parallel applications likely lowered the overhead of the system, it created many complications for message buffering. For example, if an application collectively sent a large amount of data before receiving anything, all of the information had to be stored in the driver buffer. This resulted in buffer overflows and a difficulty in determining the memory distribution between driver and applications. By receiving data in the background this issue could have been mostly alleviated.

The overall lack of debugging tools in the llamaOS system also proved to hamper the development of new programs. Because of the absence of threads in the system, this meant that most debugging had to be directly compiled in to the main program. This often resulted in `cout` messages being the most common inefficient method of determining the cause of errors. The only method of debugging crashes was through a cryptic crash report generated by Xen and a hardware debugger that was difficult to use if not in the presence of the physical machines. Spending more time developing debugging tools at the beginning would have likely decreased the overall development time.

Despite these difficulties in development, llamaMPI appears as an excellent system for hosting future parallel discrete event simulations. Still, some may argue that it would be more effective to use new cloud computing environments to perform some of the applications presented in this text. However, there will likely always be a need for the small, low-cost Beowulf cluster environment for the development of cutting edge, operating system dependant code. At least as of now, most cloud computing seems more suited to scientific applications, and request-parallel computing. Finally, there is nothing preventing llamaOS from being part of the backend running on the cloud environment. Because it works in a fully virtualized environment, it has the potential to run alongside any other type of warehouse computing code, producing an environment more suited to fine-grained programs. LlamaOS could potentially provide that compromise that can allow for high performance fine-grained development, while fulfilling the desire to consolidate computing power.

7.3 Suggestions for Future Work

As explained in the previous section, it would beneficial to add additional improvements to llamaMPI to further increase reliability and performance. Moving the receive functionality of llamaMPI to a background thread could provide a better method of buffering more suitable to scaling to larger sizes. However, the llamaNET model of instant message transfers could suffer greatly increased latency if thread scheduling ever conflicted with the receipt of a message. For this reason, an alternate method could be to move all of the message send and receive logic to the driver to allow for smarter buffering there. This could result in a difficulty in determining how to allocate memory resources between the driver and application nodes, though. In addition, llamaMPI needs more improvements into the collective messaging functions. As the *NAS Parallel Benchmarks* IS and FT proved, the collective algorithms used are not yet optimal and require additional work to equal that of native.

Even though llamaOS is currently available for public use, it is still not not suitable for large commercial or academic endeavours. Even with the addition of llamaMPI to standardize the communication interface, there are still many improvements needed to be to the basic operating system to allow for its more widespread use. For example, there are still several bugs in the system that result in crashes only fixable by restarting the machines. In addition, more debug support tools need to be added to the system to make the sources of crashes, such as these, more detectable. If implemented correctly, these tools should drastically decrease the amount of time stuck on simple bugs that currently require trial and error to fix. Finally, to make the system even more attractive to users of PDES software, driver support should be specifically added to optimized different facets of Time Warp. For example, a thread scheduler could be implemented with a maximum amount of allotted threads, as to not over-schedule the CPUs and increase latency. Once llamaMPI is able to achieve stability, in combination with the presented improvements gained without even changing the hardware, it will become more attractive for use in clusters. Moreover, the fact that such great results were returned, even though this was only a preliminary study, points to a bright future for the continued research of llamaMPI.

Bibliography

[1] T. Sterling, *Beowulf cluster computing with Linux*. MIT Press, 2001.

[2] H. Nishimura, N. Maruyama, and S. Matsuoka, "Virtual clusters on the fly - fast, scalable, and flexible installation," in *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pp. 549 –556, may 2007.

[3] M. F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis, "Virtualization for high-performance computing," *SIGOPS Oper. Syst. Rev.*, vol. 40, pp. 8–11, Apr. 2006.

[4] W. Huang, J. Liu, B. Abali, and D. Panda, "A case for high performance computing with virtual machines," in *Proceedings of the 20th annual international conference on Supercomputing*, pp. 125–134, ACM, 2006.

[5] K. Mansley, G. Law, D. Riddoch, G. Barzini, N. Turton, and S. Pope, "Getting 10 gb/s from xen: Safe and fast device access from unprivileged domains," in *Euro-Par 2007 Workshops: Parallel Processing* (L. Boug, M. Forsell, J. Trff, A. Streit, W. Ziegler, M. Alexander, and S. Childs, eds.), vol. 4854 of *Lecture Notes in Computer Science*, pp. 224–233, Springer Berlin Heidelberg, 2008.

[6] W. A. Magato, J. H. Gideon, and P. A. Wilsey, "llamaos: A solution for high performance virtualized beowulf computing clusters," in *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, IEEE, under review.

[7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.

[8] W. Gropp, E. L. Lusk, and A. Skjellum, *Using MPI-: Portable Parallel Programming with the Message Passing Interface*, vol. 1. MIT press, 1999.

[9] P. S. Pacheco, "A user's guide to mpi," *University of San Francisco*, vol. 56, 1998.

[10] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, *et al.*, "The nas parallel benchmarks summary and preliminary results," in *Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on*, pp. 158–165, IEEE, 1991.

[11] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. Di Loreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel, H. Younger, and S. Bellenot, "Distributed simulation and the Time Warp operating system," in *Proceedings of the 12th SIGOPS — Symposium of Operating Systems Principles*, pp. 77–93, 1987.

[12] K. S. Perumalla and R. M. Fujimoto, "Efficient large-scale process-oriented parallel simulation," in *Proceedings of the 1998 Winter Simulation Conference (WSC'98)*, pp. 459–466, 1998.

[13] D. E. Martin, T. J. McBrayer, and P. A. Wilsey, "Warped: A time warp simulation kernel for analysis and application development," in *System Sciences, 1996., Proceedings of the Twenty-Ninth Hawaii International Conference on,*, vol. 1, pp. 383–386, IEEE, 1996.

[14] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, (New York, NY, USA), pp. 164–177, ACM, 2003.

[15] D. Chisnall, *The definitive guide to the xen hypervisor*. Upper Saddle River, NJ, USA: Prentice Hall Press, first ed., 2007.

[16] Qumranet, "Kvm project homepage," 2009. `http://www.linux-kvm.org/`.

[17] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel, "Diagnosing performance overheads in the xen virtual machine environment," in *Proceedings of the 1st ACM/USENIX inter-*

national conference on Virtual execution environments, VEE '05, (New York, NY, USA), pp. 13–23, ACM, 2005.

[18] AMD, "Amd i/o virtualization technology (iommu) specification," tech. rep., Advanced Micro Devices, Inc., 2009.

[19] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert, "Intel virtualization technology for directed i/o," *Intel Technology Journal*, vol. 10, pp. 179–192, August 2006.

[20] K. Fraser, S. H, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, "Safe hardware access with the xen virtual machine monitor," in *In 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS*, 2004.

[21] B.-A. Yassour, M. Ben-Yehuda, and O. Wasserman, "Direct device assignment for untrusted fully-virtualized virtual machines," tech. rep., IBM Research, 2008.

[22] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High performance network virtualization with sr-iov," *Journal of Parallel and Distributed Computing*, vol. 72, no. 11, pp. 1471 – 1480, 2012. ¡ce:title¿Communication Architectures for Scalable Systems¡/ce:title¿.

[23] G. Kecskemeti, G. Terstyanszky, P. Kacsuk, and Z. Nemth, "An approach for virtual appliance distribution for service deployment," *Future Generation Computer Systems*, vol. 27, no. 3, pp. 280 – 289, 2011.

[24] J. Liedtke, "On micro-kernel construction," *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5, pp. 237–250, 1995.

[25] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: an operating system architecture for application-level resource management," in *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, (New York, NY, USA), pp. 251–266, ACM, 1995.

[26] K. Ye, X. Jiang, S. Chen, D. Huang, and B. Wang, "Analyzing and modeling the performance in xen-based virtual cluster environment," in *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, pp. 273 –280, sept. 2010.

[27] G. Ciaccio, "Gamma project homepage," 2009. `http://www.disi.unige.it/project/gamma/`.

[28] J.-W. Jang, E. Seo, H. Jo, and J.-S. Kim, "A low-overhead networking mechanism for virtualized high-performance computing systems," *The Journal of Supercomputing*, vol. 59, pp. 443–468, 2012. 10.1007/s11227-010-0444-9.

[29] B. Meyer, "An introduction to c++ (from material by nadia polikarpova)," in *Touch of Class*, pp. 805–838, Springer, 2009.

[30] R. Thakur and W. D. Gropp, "Improving the performance of collective operations in mpich," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 257–267, Springer, 2003.

[31] M. Frumkin, "Data flow pattern analysis of scientific applications," in *Workshop on Patterns in High Performance Computing*, 2005.

[32] A. A. Faraj and X. Yuan, *Communication characteristics in the NAS parallel benchmarks*. PhD thesis, Florida State University, YEAR., 2002.

[33] F. C. Wong, R. P. Martin, R. H. Arpaci-Dusseau, and D. E. Culler, "Architectural requirements and scalability of the nas parallel benchmarks," in *Supercomputing, ACM/IEEE 1999 Conference*, pp. 41–41, IEEE, 1999.

[34] J. Misra, "Distributed discrete-event simulation," *ACM Computing Surveys (CSUR)*, vol. 18, no. 1, pp. 39–65, 1986.

[35] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, 1990.

[36] R. M. Fujimoto, J.-J. Tsai, and G. Gopalakrishnan, "Design and performance of special purpose hardware for time warp," *ACM SIGARCH Computer Architecture News*, vol. 16, no. 2, pp. 401–409, 1988.

[37] R. E. Bryant, “Simulation on a distributed system,” in *Proc. of the First Intern. Conf. on Distributed Computing Systems, IEEE, New York, NY*, pp. 544–552, 1979.

[38] K. M. Chandy and J. Misra, “Distributed simulation: A case study in design and verification of distributed programs,” *Software Engineering, IEEE Transactions on*, no. 5, pp. 440–452, 1979.

[39] D. R. Jefferson, “Virtual time,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 3, pp. 404–425, 1985.

[40] R. King, *Warped redesigned: An api and implementation for discrete event simulation analysis and application development*. PhD thesis, University of Cincinnati, 2010.

[41] H. Carter, R. Vemuri, P. Wilsey, J. Aylor, R. Waxman, and T. Hartrum, “High speed acceleration of vhdl simulation, synthesis, and atpg: Overview of the quest project,” in *VHDL Users Group Spring 1991 Conference*, pp. 85–90, 1991.

[42] T. McBrayer, V. Krishnaswamy, S. Mohanty, L. Moore, X. Liu, J. Carter, D. Charley, P. A. Wilsey, D. A. Hensgen, H. W. Carter, *et al.*, “Vast: Time warp simulation of vhdl on smp workstations,” in *VHDL Users Group Fall 1994 Conference*, 1994.

[43] D. E. Martin, T. McBrayer, and P. A. Wilsey, “WARPED: A Time Warp simulation kernel for analysis and application development,” 1995. (available on the www at `http://www.ece.uc.edu/~paw/warped/`).

[44] P. Reiher, S. Bellenot, and D. Jefferson, “Temporal decomposition of simulations under the time warp operating system,” *Parallel and Distributed Simulation (PADS), San Diego*, 1991.

[45] F. Wieland, L. Hawley, A. Feinberg, M. D. Loreto, L. Blume, J. Ruffles, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot, *et al.*, “The performance of a distributed combat simulation with the time warp operating system,” *Concurrency: Practice and Experience*, vol. 1, no. 1, pp. 35–50, 1989.

[46] P. L. Reiher, F. Wieland, and D. Jefferson, “Limitation of optimism in the time warp operating system,” in *Proceedings of the 21st conference on Winter simulation*, pp. 765–770, ACM, 1989.

[47] S. Bellenot, "Performance of a riskfree time warp operating system," in *ACM SIGSIM Simulation Digest*, vol. 23, pp. 155–158, ACM, 1993.

[48] K. S. Perumalla, "Scaling time warp-based discrete event execution to 104 processors on a blue gene supercomputer," in *Proceedings of the 4th international conference on Computing frontiers*, pp. 69–76, ACM, 2007.

[49] W. Gropp and E. Lusk, "Creating a new mpich device using the channel interface," *MPICH working note (Dec. 1995)*, 1995.

[50] W. Gropp and E. Lusk, "Installation guide for mpich, a portable implementation of mpi," tech. rep., Technical Report ANL-96/5, Argonne National Laboratory, 1996.

[51] P. Bridges, N. Doss, W. Gropp, E. Karrels, E. Lusk, and A. Skjellum, "Users guide to mpich, a portable implementation of mpi," *Argonne National Laboratory*, vol. 9700, pp. 60439–4801, 1995.

[52] R. Butler, W. Gropp, and E. Lusk, "A scalable process-management environment for parallel programs," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 168–175, Springer, 2000.

[53] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir, "Mpi-2: Extending the message-passing interface," in *Euro-Par'96 Parallel Processing*, pp. 128–135, Springer, 1996.