DESIGN AND EVALUATION OF A COMMUNICATION LIBRARY FOR VOLUNTEER COMPUTING ENVIRONMENTS

A Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Troy Paul LeBlanc

December, 2009

DESIGN AND EVALUATION OF A COMMUNICATION LIBRARY

FOR VOLUNTEER COMPUTING ENVIRONMENTS

Troy Paul LeBlanc

APPROVED:

Dr. Jaspal Subhlok, Chairman
Department of Computer Science

Dr. Edgar Gabriel
Department of Computer Science

Dr. J.C. Huang
Department of Computer Science

Dr. Rong Zheng
Department of Computer Science

Dr. Norman Kluksdahl
NASA - Johnson Space Center

Dean, College of Natural Sciences and Mathematics

Acknowledgements

There are many people who offered their knowledge and encouragement throughout this endeavor.

I would like to thank my advisor, Dr. Jaspal Subhlok, for his guidance and open-mindedness as I climbed the learning curve of parallel programming and volunteer programming. Next, I want to thank Dr. Edgar Gabriel for the countless hours of expert development and debugging he spent with me in creating VolpexMPI.

Next, I would like to thank some of my fellow students for their involvement and assistance. These individuals include Dr. Qiang Xu, Ravi Prithivathi, and Nagarajan Kanna. Most importantly, I want to thank Rakhi Anand for her work on MCFA, debugging on VolpexMPI and SockLib, and her many hours of testing VolpexMPI.

Keith Martin and Michael Sen-Roy, who both work with me at the Johnson Space Center, deserve special thanks for the time they spent helping me set up and troubleshoot the virtualized cluster. I will venture to say that they also gained valuable experience that is sure to help in the decision making concerning the use of virtualized clusters in the Mission Control Center.

Finally, I want to thank my wife, Michelle, and my three children, Daniel, Eric, and Adeline for their love and patience without which this would not have been possible.

Troy P. LeBlanc
Houston, Texas
December 2009

DESIGN AND EVALUATION OF A COMMUNICATION LIBRARY FOR VOLUNTEER COMPUTING ENVIRONMENTS

An Abstract of a Dissertation

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Troy Paul LeBlanc

December, 2009

Abstract

To date, idle desktop computers, volunteered by the general public to create volunteer computing environments, have been limited to sequential scientific computing. The objective of this research is to convert idle desktop computers into virtual cluster nodes for executing parallel scientific applications. The dissertation introduces VolpexMPI, which is designed to enable seamless forward application progress in the presence of frequent node failures as well as dynamically changing networks speeds and node execution speeds. Process replication is employed to provide robustness in such volatile environments. The central challenge in VolpexMPI design is to efficiently and automatically manage dynamically varying number of process replicas in different states of execution progress. The key fault tolerance technique employed is fully distributed, sender-based logging. The dissertation presents the design and performance of two architectures of VolpexMPI. One architecture implements asynchronous message passing with non-blocking sockets in C, with an emphasis on performance. The other architecture utilizes Python and threaded TCP services with the goal of portability between heterogeneous desktop computers. These implementations are tested by executing parallel benchmarks on dedicated clusters as well as virtualized clusters and pools of clusters managed by Condor. The C architecture results validate that the overhead of providing process replication is modest for parallel applications, having a favorable ratio of communication to computation and a low degree of communication. However, the Python architecture results show significant performance degradation when executing parallel scientific applications, even though the development process was remarkably easier.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Idle desktop computers represent an immense pool of unused computation, communication, and data storage capacity [5, 36, 11]. However, a very small fraction of idle desktop computers are used for volunteer computing, and the usage is largely limited to "bag of tasks" and sequential scientific applications. As defined by Anderson [3], volunteer computing uses computers volunteered by the general public to do distributed scientific computing. Scientific applications in high-energy physics, molecular biology, medicine, astrophysics, climate study, and other areas are being researched using volunteer computing environments. The volunteered computers vary widely in terms of software and hardware type, speed, availability, reliability, and network connectivity. Similarly, the scientific applications vary widely in terms of their resource requirements.

Volunteer computing environments must deal with several problematic aspects of the volunteered computers. Namely, the arrival and departure of nodes, their heterogeneity, their availability which can change frequently and without warning, and the need to not interfere with the volunteer computer's performance during regular use. Additionally, volunteer computers will always remain unaccountable and essentially anonymous. Most of

today's volunteer computing environments have the same basic structure. A client program runs on the volunteer's computer and periodically contacts project-operated servers over the Internet to request jobs. The volunteer attempts to complete the job and report the results back to the server for "credit". This credit is a numerical measure of how much work that user's computers have done for the project. As a result, most existing volunteer computing projects are throughput-oriented, i.e., minimal latency constraints, and have relatively small memory, disk, and network bandwidth requirements.

Traditionally, volunteer computing projects have targeted large scientific search problems that generate a large set of tasks to be distributed across the volunteer computer environment. The Berkeley Open Infrastructure for Network Computing (BOINC) [3] volunteer computing projects include SETI@home, Predictor@home, Folding@home, and Climate@home to name a few. They all use the master-slave paradigm and are essentially sequential scientific applications. BOINC uses replication of work to address volatility as well as malicious attacks, hardware malfunctions, and software modification. In the recent past, some of the largest pools of commercial compute resources, specifically Amazon [2] and Google [24], have opened up part of their computation farms for public computing. Often these commercial compute resources are very busy on a few occasions (e.g. Christmas shopping) and underutilized the rest of the time. This use of commercial compute resources is often referred to as "cloud computing" and is further investigated in GridBatch [38]. The advent of multi-core CPUs and increasing deployment of Gigabit capacity interconnects have made mainstream institutional networks an increasingly attractive volunteer platform for executing scientific codes as "guest" applications.

Idle desktop computers have been formed into volunteer computing environments and have been successfully used to run sequential and master-slave task parallel codes, most notably under Condor [56] and BOINC. However, we are not aware of any Message Passing Interface (MPI) implementation that is used widely to support truly parallel execution in

these environments. Therefore, it is an ambitious goal to develop a framework for execution of parallel scientific applications in volunteer computing environments based on MPI, the de-facto standard for message passing parallel computing.

1.1 Available Environments

In addition to the potentially failure-prone volunteered compute resources introduced above, volunteer computing projects can get unused compute resources from clusters, grids, and clouds. Here we describe the characteristics of these available compute environments.

Clusters are tightly coupled computers that act like a single system and are used for computation and communication intensive tasks. Clusters are based on off-the-shelf hardware components such as Intel [41] or AMD [12] processors and utilize standard operating systems such as Linux and Windows. Most clusters utilize high performance network interconnects such as 10Gbps Ethernet, Myrinet and InfiniBand. Clusters are best suited for the applications that are highly communication intensive, and need faster turnaround times. The overall cost involved in purchasing, maintaining, and running a cluster can be quite significant and includes the cost of hardware components, softwares licenses, cooling, and the manpower required to install and operate a cluster.

Grid computing is a type of distributed computing composed of loosely coupled computers to perform very large tasks. Grid computing allows the sharing and coordinated use of geographically distributed resources, enabling the creation of ad hoc scientific research communities. Despite offering several advantages to the end-users, grids also suffer from some major disadvantages such as only supporting non-interactive job submissions and requiring the installation of additional software which can interfere with the host operating system and batch scheduler.

Cloud computing lets the user access data from anywhere in the world, allowing him to be free from the individual desktop. Through virtualization, the location of data and computation are not necessarily exposed to the end-user, hence the name “cloud”. An advantage of cloud computing is that as the need for compute resources increases, that additional resource can be purchased from the cloud computing provider without having to purchase any local additional resouces. However, current clouds are not well suited to applications that need many tightly coupled threads/processes, as in large MPI-based data analysis applications, because many of the high-speed network interconnects are not currently supported by virtual machine monitors.

1.2 Message Passing Interface

The MPI Standard [43] exists to maintain a widely used, clearly defined base set of functions for writing efficient message-passing programs. Of primary importance is portability and ease of use. Among the goals listed in the standard are an allowance for implementations that can be used in a heterogeneous environment and allowance for convenient C and Fortran 77 bindings for the interface. These goals are very much in line with the goals of VolpexMPI.

Another goal listed in the MPI Standard is the assumption of a reliable communication interface. The intention is that the implemented library, and not the user, copes with communication failures. In today’s MPI libraries, this generally results in an abort of the parallel run. Other goals include defining an interface that is not too different from current practice, and allowance for language independence and thread safety.

The MPI Standard includes function definitions broken into several categories. These include point-to-point communications, collective operations and communicator functions.

Point-to-point communications provide definition for blocking and non-blocking send and receive operations. Collective operations are communications that involve a group of processes. The communicator functions provide descriptive context and partitioning of the communication space. Other topics in the MPI Standard include topologies, bindings for C and FORTRAN, and a profiling interface to instrument a MPI Library for testing.

1.3 Challenges

Harnessing idle desktop computers for parallel scientific applications presents significant challenges. The nodes have varying compute, communication, and storage capacity and their availability can change frequently and without warning as a result of, say, a new host application, a reboot or shutdown, or just a user mouse click. Further, the nodes are connected with a shared network where available latency and available bandwidth can vary [34]. Because of these properties, we refer to such nodes as *volatile*, and parallel computing on volatile nodes is challenging. The current state-of-the-art in fault tolerant execution on parallel systems is based on regular check-pointing and restart on failure. However, a checkpoint-based approach is not effective or scalable for volunteer computing, when the typical interval between failures is very small, because the overhead of frequent checkpointing and routine recovery can dominate.

Another challenge for executing parallel scientific applications on volunteer computing enviroments is heterogeneity. The number of Internet-connected desktop computers is growing rapidly, and is projected to reach 2 billion by 2015, with possibly 90% of these computers running Windows operating systems. However, most clusters and grids run Linux operating systems and, therefore, most scientific parallel applications are compiled for Linux. To minimize impact to the science community, the volunteer computing

community must develop methods to compile and execute the legacy parallel scientific applications without alteration. Native compute size and byte ordering must be accounted for when considering the options which range from interpreted language run time environments to virtualization to platform specific compilation.

The MPI Standard makes no provision for these challenges. The expected behavior of a MPI library is to abort if a node becomes unavailable because the standard can not guarantee the traditional robustness of client server systems [43]. However, these challenges must be met because many parallel scientific applications are already written in C or FORTRAN using MPI, and the intention of this research is to provide a capability to execute those programs on idle desktop computers without alteration.

1.4 Design Features

This dissertation introduces Volpex (Parallel Execution on Volatile Nodes) MPI that represents a comprehensive and scalable solution to execution of parallel scientific applications on virtual clusters composed of idle desktop computers. Our solution is coordinated use of redundant nodes as shown in Figure 1.1 and the key features of our approach are the following:

1. *Controlled redundancy:* A process can be initiated as two (or more) replicas. The execution model is designed such that the application progresses at the speed of the fastest replica of each process and is unaffected by the failure or slowdown of other replicas as in Figure 1.2. (Replicas may also be formed by checkpoint-based restart of potentially failed or slow processes, but this aspect is not implemented yet.)

2. *Receiver-based direct communication:* The communication framework supports direct node to node communication with a *pull* model; the sending processes buffer

data objects locally and receiving processes contact one of the replicas of the sending process to get the data object.

3. *Distributed sender-based logging:* Messages sent are implicitly logged at the sender and are available for delivery to process instances that are lagging due to slow execution or recreation from a checkpoint.

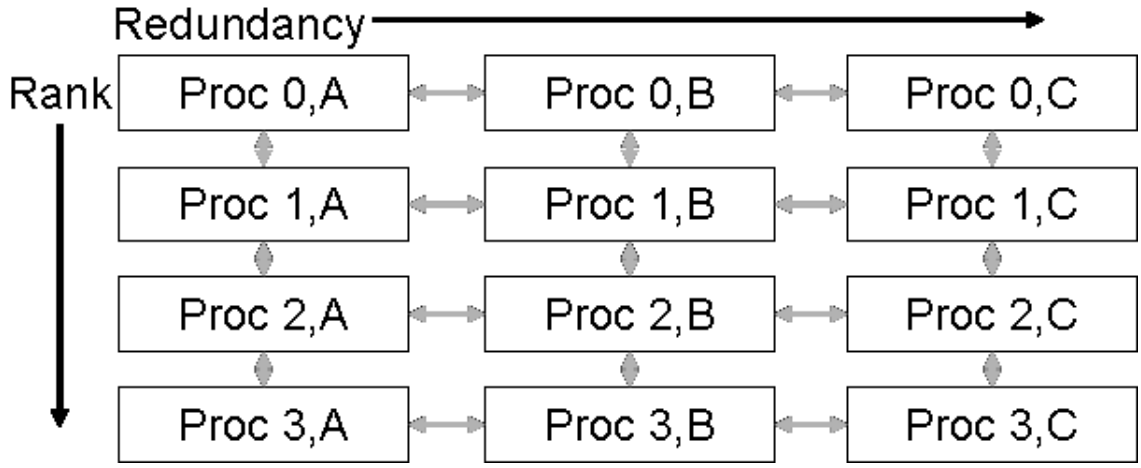

Figure 1.1: VolpexMPI employs redundant nodes to provide fault tolerance in the volunteer environment.

VolpexMPI is designed as a layered library with the intention of being compiled and dynamically linked to the user's MPI application, which is the top layer. The middle layer provides startup and shutdown routines, MPI point-to-point and collective and communicator function calls, as well as an Application Programmers Interface (API) that allows for user programs to be written in either C or FORTRAN. Finally, the lowest layer deals with point-to-point actual movement of bytes between MPI processes, buffer management and redundant target selection. Each component of all three layers will be described in greater detail in Chapter 3. In keeping with the key features listed previously, the VolpexMPI design allows for multiple redundant copies of a process and manages all copies through messages that check for aliveness and active maintenance of a selectable targets list. Also,

the design enforces the actual transfer of messages to be initiated by receiving processes which then requires the sending processes to store messages persistently for potential redundant data calls.

VolpexMPI is designed for applications with moderate communication requirements and is expected to scale to hundreds of nodes on institutional Local Area Networks (LANs). Certainly many parallel applications will not run effectively on desktop computers under VolpexMPI (or any other framework) because of memory and communication requirements that can only be met with dedicated clusters. In particular, for an application to run effectively on volatile nodes, it must have a low communication degree and limited sensitivity to latency. It has been shown that many parallel scientific applications have a low degree stencil as the dominant communication pattern [52, 33]. Hence, we believe that many parallel applications are good candidates, and an important goal of this project is to identify the extent of applicability of this approach.

An example motivating application is Replica Exchange Molecular Dynamics (REMD) formulation [51] where each node runs a piece of molecular simulation at a different temperature using the AMBER application [10]. At certain time steps, communication occurs between neighboring nodes based on the Metropolis criterion of a given parameter being less than or equal to zero. REMD requires low volume, loosely coupled communication, making it a good candidate for VolpexMPI. It is currently implemented in the Volpex environment [30] but not yet ported to MPI.

1.5 Basic Contributions

This dissertation contributes the design and validation of two architectures for VolpexMPI. One is an all C architecture that emphasizes performance and is tested in comparison

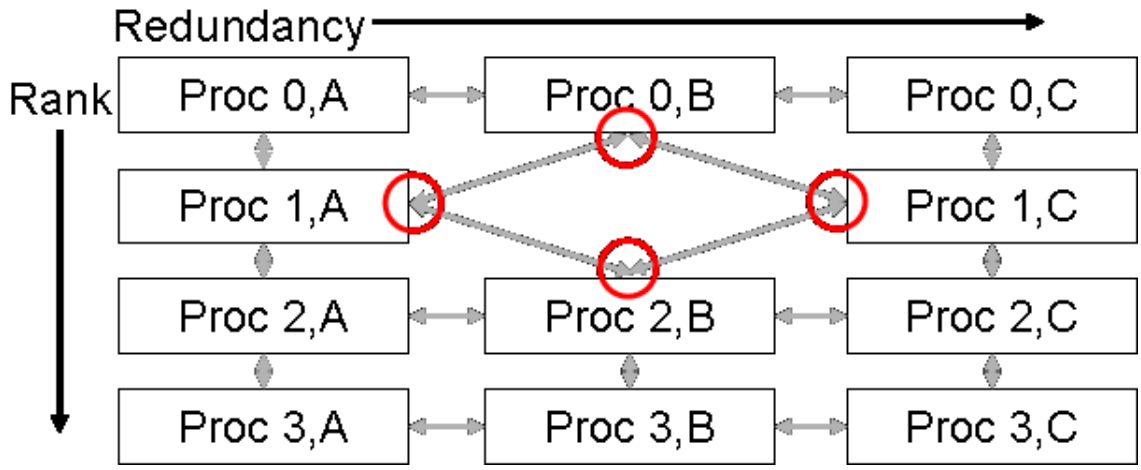

Figure 1.2: VolpexMPI attempts to utilize redundant nodes to provide fault tolerance in the event of a node becoming unavailable during execution.

to Open MPI [22] running on a dedicated cluster. The C-based architecture allows for user applications to be written in either C or FORTRAN and is built with an emphasis on performance. The second architecture is built using Python and many of its features to emphasize portability between Linux and Windows desktop computers. We also describe trying higher-level communications protocols (e.g., XML-RPC) and databases and data structures. Both architectures can successfully execute the NAS Parallel Benchmarks.

The C-based architecture validates that we incur minimal replica communications overhead in maintaining controlled redundancy as compared to Open MPI on Linux systems using no replication. However, the results from the Python-based architecture validate that significant overhead is incurred in using high-level interpreted languages for high performance parallel computing on heterogeneous desktop computers. We present results for a dedicated commodity desktop computer cluster, a virtualized IBM blade cluster and a Condor pool, all comparing VolpexMPI to Open MPI on the dedicated cluster. We analyze the overhead of replication and node failure or slow-down scenarios for the Class B NAS Parallel Benchmarks executing on 8 and 16 nodes.

The rest of the dissertation is organized as follows. In Chapter 2 we describe other work related to VolpexMPI in the areas of fault tolerant MPI libraries, public resource computing, Python-based MPI libraries and peer-to-peer systems. The objectives and design of VolpexMPI are discussed in Chapter 3. Chapter 4 presents the performance oriented architecture of the VolpexMPI library, especially the MPI standard functions developed to-date along with the results from three testbed environments. Chapter 5 explains the architecture for heterogeneous environments and presents the experiments and results for that system. We discuss future possible work on VolpexMPI and provide conclusions to this dissertation in Chapter 6.

Chapter 2

Related Work

The work we present in this dissertation capitalizes on common properties across many developments in both parallel and volunteer computing. Of primary importance to our work is Anderson's work [3] and [4] on the development and resource characterization of the BOINC software system. The goals of the BOINC software system are to reduce the barriers of entry to public computing resources, share those resources among autonomous projects, support diverse applications and reward participation. This recognition of the potential of public resource computing has led the way for allowing the volunteer computing community to participate in scientific research that requires vast computing resources. Anderson and Fedak [5, 18] also conclude that extending the BOINC software system to allow peer-to-peer communication, rather than client-initiated communication only, will increase overall system capacity for new scientific computations requiring intermediate files and replicated inputs. In this section, we divided our survey of related work into public resource computing, peer-to-peer computing, and Python-based MPI libraries, and we reviewed the latest research in fault-tolerant MPI libraries.

2.1 Public Resource Computing

Real-world application of the BOINC software system is described by Anderson based on Gedye's proposal for SETI@home [4] which has received world-wide attention. It uses the master-slave paradigm where the worker is a client application running as a screen-saver that many people installed on personal computers to allow radio-telescope data to be analyzed during idle CPU time. Another application of the BOINC software system is Predictor@home [53] which also employs the master-slave paradigm implemented with updated BOINC daemons for the specific computing task, validation of returned results, and storage of the initial data and results. As a result of using volunteer computing, Predictor@home was able to increase its sampling by one or two orders of magnitude. These two examples only required the client application to request a work unit (i.e., data), analyze it and return the result to the server with no required intermediate communication. There are also discrete event simulators for BOINC using SimBA [20] and Docking@home [54] to better characterize the public resource computing environment.

2.2 Peer-to-Peer Systems

Next our literature survey focused on peer-to-peer (P2P) process-sharing solutions to problems arising in volunteer computing. BambooTrust [37] is described as a scalable distributed trust management system to address threats posed by the open and public nature of volunteer computing. BambooTrust is built as a peer-to-peer system using an open distributed hash table to allow for dynamic scaling using a single-threaded programming style. WaveGrid [60] is also a self-organized peer-to-peer volunteer computing system, but it focuses on providing a timezone-aware overlay network taking advantage of user daily routines targeting idle personal computers during night-time cycles. WaveGrid emphasizes

scheduling and process migration due to the heterogeneous nature of volunteer computing resources, but still assumes one type of operating system for its simulation experiments. Phoenix [55] is a peer-to-peer overlay routing infrastructure that is based upon the fact that parallel computing applications are migrating towards volunteer computing networks, and it is primarily concerned with the dynamic nature of nodes joining and leaving the network. Phoenix supports transparent process migration and a dynamic routing table to maintain a network of nodes for volunteer computing. G2:P2P [40] is also a peer-to-peer distributed hash table overlay network built to allow fault tolerant process execution. G2:P2P emphasizes that it opted to use the .NET Framework to form a distributed object model versus a distributed process model common to MPI. This approach allows developers to introduce middleware to intercept messages being passed between the objects. Virtual Private Grid [29] is another example of a grid manager that constructs a spanning tree among hosts and provides a simple view of machines in multiple administrative domains. These peer-to-peer volunteer computing solutions all recognize the need to provide a framework to oversee a myriad of tasks that should not be the responsibility of parallel scientific application developers. Parallel computations within volunteer computing need a framework that establishes trust among the nodes, provides fault-tolerant execution, allows for dynamic joining and leaving of nodes and even takes advantage of profiling CPU cycle usage.

2.3 Python-based MPI Libraries

Python provides run-time environments for both Linux and Windows, allowing programmers to write one set of functions for heterogeneous platforms. A search on the World Wide Web for implementations of the MPI standard using Python produced several results. The efforts that most closely matched the VolpexMPI criteria for implementing MPI include Pypar, Mpi4py, pyMPI, and Pydusa; however, none of these implementations offer

redundancy for fault tolerance.

The first two implementations found simply add a Python API to an existing MPI library for the purpose of writing user MPI programs in Python. Pypar [44] provides bindings to a subset of the MPI standard functions. To execute MPI programs, Pypar depends on a Python installation, a MPI library, Numeric Python and a C compiler. Pypar works with each Python process, importing pypar which, in turn, imports a shared library that has been statically linked to the C MPI library (e.g., mpich). Supported MPI calls are made available to Python through the Pypar interface. Pypar is not an independent implmentation of the MPI standard. MPI for Python (Mpi4py) [14] provides an object oriented approach to message passing in accordance with the standard MPI-2 C++ bindings. This implementation was designed for translating MPI syntax and semantics of standard MPI-2 bindings for C++ to Python. In Mpi4py, the Send(), Recv() and Sendrecv() methods of communicator objects provide support for blocking point-to-point communications passing general Python objects. Mpi4py depends on a working MPI distribution and Python 2.3 distribution minimally. Mpi4py is also not an independent implmentation of the MPI standard.

The next two examples researched are independent Python MPI implementations. The pyMPI [45] completely rebuilds the python interpreter to be an integral part of each parallel process. The intention of the developers is to provide a MPI library for MPI programs written in Python. This differs fundamentally from Volpex's basic requirement to be able to compile and execute user MPI programs written in both C and FORTRAN. Pydusa [28] is closest in design to Volpex because it uses the existing Python interpreter and has implemented 30 of the MPI functions in its library. Pydusa is, however, also intended for executing MPI programs written in Python.

The researched Python implementations of MPI are mostly scoped to allow the user to write MPI programs in Python instead of C or FORTRAN. While this may be a good

teaching tool, the majority of the scientific MPI programs are written in C or FORTRAN. It is assumed that Pypar and mpi4py could execute the NAS Parallel Benchmarks because they provide an API to existing MPI libraries. PyMPI did report in ping-pong tests being two orders of magnitude slower than a C MPI installation [45]. Also, none of these implementations provide redundancy for fault tolerance or have been tested against the NAS Parallel Benchmarks, so no comparable data exists.

2.4 Fault Tolerant MPI Libraries

The MPI specifications are rather vague about failure scenarios. As many as ten years ago, efforts such as Starfish [1] were researching fault tolerance solutions for MPI. In recent years many MPI implementations have been developed to deal with process and network failures. Fault tolerant methods supported by various implementations of MPI can be divided into three categories: 1) extending the semantics of MPI, 2) check-point restart mechanism, and 3) replication techniques.

FT-MPI [17] is the best known representative of the approach of extension of semantics for failure managment. The specification of FT-MPI defines the status of the MPI process handles and messages in case of a process failure. FT-MPI has the ability to either replace a failed process or continue execution without it. The library deals only with MPI-level recovery but lets the application manage the recovery of user level data items in a performance efficient manner [39]; however, it requires significant modifications to the application, and thus does not provide a transparent fault tolerance mechanism.

LAM/MPI [49], MPICH-V [8] and MPICH-V2 [9] all belong to the category of MPI libraries that employ checkpoint-restart mechanisms for fault tolerance. All are based on

uncoordinated check-pointing and pessimistic message logging. For example, in MPICH-V the library stores all communications of the system on reliable media through the usage of a *channel memory*. In case of a process failure, MPICH-V is capable of restarting the failed application process from the last checkpoint and replay all messages to that process. The MPICH community also has implemented a coordinated check-pointing library called MPICH-Pcl [13] that, upon detecting a fault, forces every process to reload its last local image and reinitiate connections before the computation can continue. Similarly to MPICH-V and MPICH-V2, MPICH-Pcl, RADICMPI [15] also fundamentally relies on checkpointing MPI processes but tries to avoid any central instance or single point of failure within its overall design. Although some of the conceptual aspects of VolpexMPI are similar to MPICH-V and RADICMPI, there are key architectural differences. Neither of these two MPI libraries is designed to run multiple replicas of a MPI process. Also, Open MPI [26] offers a design and implementation of an infrastructure to support checkpoint-restart fault tolerance in the Open MPI project. The approach was to identify the capabilities required for distributed checkpoint-restart and implement those capabilities as extensible frameworks within Open MPI's modular component architecture. Although their implementation includes some initial checkpoint-restart mechanisms, the Open MPI framework is actually meant to encourage experimentation of fault tolerance techniques within a production quality MPI implementation. Thus, while VolpexMPI can continue the execution of an application seamlessly in case of a process failure if replicas are available for that process, MPICH-V, RADICMPI and Open MPI's work in fault tolerance will have to deal with the overhead generated by restarting processes from an earlier checkpoint. Also, message logging in VolpexMPI is fully distributed on host nodes themselves and there is no equivalent of channel memories.

Egida [48] is an object oriented toolkit designed to support transparent rollback recovery. Egida automatically synthesizes an implementation of a rollback protocol, such as

failure-detection events, by gluing together appropriate objects from an available library. The Egida implementation replaces all send and receive calls from the MPICH implementation of the MPI standard to enable failure-detection through a watchdog for each application process. Unlike Egida, VolpexMPI has built failure-detection directly into the MPI library and therefore directly into the executable process. MPI/FT [6] also provides transparent fault tolerance by replicating MPI processes and introducing a central coordinator. The library is able to recognize malicious data by using a global voting algorithm among replicas. However, this feature also leads to an exponential increase in the number of messages with the number of replicas: each replica sends every message to all destination replicas. VolpexMPI avoids the penalty resulting from this communication scheme by ensuring that a message is pulled from exactly one sender with receiver initiated communication [46]. P2P-MPI [23] is also based on replication techniques where each set of process replicas maintain a master replica that distributes messages. Fault detection is done using a gossip-style protocol [57], which has the ability to scale well and provides timely detection of failures. P2P-MPI also takes advantage of locality awareness and co-allocation strategies. A key difference is that, unlike P2P-MPI, VolpexMPI utilizes a *pull* model for data communication that ensures that the application advances at the speed of the fastest replica for each process.

Distributed resource managment and dynamic process management are key ideas for some MPI implementations such as Stampi [27]. Stampi also supports Java communications in an effort allow for heterogeneous environments. As well, the PC3 system described in [19] proposes that MPI applications need to be transparently portable to take advantage of resources at different sites and clusters as they become available. Because PC3 migrates processes, the developers built a thin layer providing a virtual MPI interface so that an application only needs to deal with logical MPI processes and the thin layer handles the

logical to physical mapping. EasyGrid [7] describes itself as dynamic scheduling middleware embedded within MPI applications to provide a framework for dynamic process creation without resorting to task replication. EasyGrid uses a Global Manager to supervise sites on a grid where a MPI application is running, a Site Manger to handle node allocation, and a Host Manager to handle creation, scheduling and execution of MPI application processes. Koenig and Kale [35] present a run-time system of message-driven objects developed with Charm++ and Adaptive-MPI to mask latency encountered in moving application processes between geographically distributed clusters. Koenig and Kale see this work scaling to handle extremely large computations that exceed the capacity of a single cluster and scheduling resources on-demand. Finally, there are grid-enabled MPI libraries such as PACX-MPI [32] as well as MPICH-G2 [31] that enables MPI on a grid that exploits the communicator construct to discover and adapt to changing network topology, but not failure. These developments of MPI, each individually, address issues central to parallel volunteer computing such as random node failure, resource replication, and the need for communications framework middleware.

Across the literature survey, we endeavored to discover common research among the volunteer programming systems, the peer-to-peer systems, Python-based MPI libraries and the latest application of fault tolerance to MPI libraries. Volunteer systems are still not attempting direct node-to-node communcation in the execution of parallel programs and peer-to-peer systems are focused, appropriately, on availability and trust issues and do not discuss performance which is paramount to parallel computing. There are many different approaches to providing fault tolerance in executing MPI programs in volunteer environments and VolpexMPI chooses to employ replication for fault tolerance. Check-pointing offers a good solution when failures are infrequent whereas replication offers better performance when failure rates are high [59]. VolpexMPI is designed to balance replication and checkpoint-restart, although the current implementation is limited to replication.

Chapter 3

Objectives and Design

The design of VolpexMPI calls for seamless forward application progress in the presence of frequent node failures as well as dynamically changing networks speeds and node execution speeds. Process replication is employed to provide robustness in such volatile environments. The central challenge in VolpexMPI design is to efficiently and automatically manage dynamically varying number of process replicas in different states of execution progress. The key fault tolerance technique employed is fully distributed, sender-based logging. VolpexMPI is designed for applications with moderate communication requirements meaning a low communication degree and limited sensitivity to latency.

3.1 Objectives

Before describing the design of VolpexMPI, we first need to identify the key objectives that are to be achieved to consider any implementation of this research to be successful. The key design objectives of VolpexMPI are the following:

1. *Fail-safe execution:* A parallel scientific application should make continuous, robust progress, despite varying capabilities of nodes and frequent, unexpected failures.

2. *Transparency:* A pool of unused desktops should appear as a virtual cluster that can execute unaltered MPI programs written in either C or FORTRAN and running either Linux or Windows operating systems.

3. *Scalability:* The volunteer network of desktop computers should scale to 100s or more volatile nodes by minimizing the overhead of fail-safe execution.

3.2 Design

Fundamentally, VolpexMPI is a software library composed of functional modules implementing the MPI-1 specification [42]. To design VolpexMPI, a more explicit set of design statements need to written and traced up to the design objectives and design features and allocated to appropriate functions in the software library modules. Recall that our solution is coordinated use of redundant nodes and that the key design features of our approach as described in Section 1.4 are the following:

1. *Controlled redundancy:* A process can be initiated as two (or more) replicas. The execution model is designed such that the application progresses at the speed of the fastest replica of each process and is unaffected by the failure or slowdown of other replicas.

2. *Receiver-based direct communication:* The communication framework supports direct node to node communication with a *pull* model; the sending processes buffer data objects locally and receiving processes contact one of the replicas of the sending process to get the data object.

3. *Distributed sender-based logging:* Messages sent are implicitly logged at the sender and are available for delivery to process instances that are lagging due to slow execution or recreation from a checkpoint.

We have discussed the key objectives and several design features in this and the previous section which provide us with a general design strategy. Now we will present a more specific set of design statements that can be prototyped as working architectures. These high level "requirements" should be easily discerned when reading about the two prototyped architectures in the subsequent chapters.

VolpexMPI needs to provide for continued execution in the event of node process failure. In the volunteer computing environment, nodes will fail. Timely detection and response to these failures is paramount to design toward application progress at the speed of the fastest node. This design feature satisfies the fail-safe execution and transparency objectives by using node replication. Primarily, we designed target selection functions that are called within the non-blocking send and receive functions with time-outs to ensure continued execution. More details on the functions implementing this design feature are in the Point-to-Point, SockLib, and Target Selection modules.

VolpexMPI needs to progress at the speed of the fastest replica and identify replica targets and track aliveness of every replica. This design feature satisfies the fail-safe execution, scalability, controlled redundancy objectives and features. This first design uses a simple round-robin approach to target selection, but more sophisticated algorithms are currently being designed. Identification of replicas is described in an upcoming section along with ports and ip addresses to allow for tracking and statusing hundreds of nodes. Function details implementing this design feature are in the Point-to-Point, Startup, and Target Selection modules.

VolpexMPI needs to inform all replicas when a replica has become unavailable. This design feature satisfies the fail-safe execution, scalability, and controlled redundancy objectives and features. This feature alludes to the fact that there are framework communication needs in addition to the message passing communication for the user application. This design employs a set of daemons that establish connectivity from a cluster front end computer to compute nodes. Functionality implementing this design feature is in the Startup module.

VolpexMPI needs to allow the user to set any level of redundancy and request any number of nodes. This statement satisfies the transparency design feature by managing the total set of replicas behind the scenes after the user initiates a parallel scientific application. Usually this information is provided as command line arguments. Again, functionality implementing this design feature is in the Startup module.

VolpexMPI needs to provide an API to the named MPI functions per the MPI Standard in both C and FORTRAN. The initial set of functions implemented is based on usage in NAS Parallel Benchmarks. This statement satisfies the transparency design feature. A fundamental idea from the conception of this research has been to execute unaltered parallel scientific applications written with MPI. Functions implementing this design feature are in the API, Point-to-Point, Collective, and Communicator modules.

VolpexMPI needs to provide a startup mechanism similar to mpirun that allows for command line arguments such as number of processes, redundancy level desired, use in condor pools, hostfile, and executable location. This feature can be implemented as a command line tool or as a user interfaced tool. This statement satisfies the transparency design feature and is implemented in the Startup module.

VolpexMPI needs to be able to run on dedicated clusters, virtualized clusters, and Condor pools of clusters. Volunteer computing environments as described in Chapter 1 could comprise resources from any of these compute capabilities. This design feature satisfies the

scalability objective. Functions implementing this design feature are in all of the modules.

VolpexMPI needs to be able to execute on Linux and Windows operating systems. This design idea supports the scalability design objective and addresses the heterogeneous nature of volunteer computing environments. In a later chapter we describe how Python architecture helps to meet this design. The C architecture can also meet this design, but the actual implementation is complicated by the fact that Unix sockets and Windows winsock are not functionally similar. Functions implementing this design feature are in all of the modules.

VolpexMPI needs to implement asynchronous message passing. This design statement supports the receiver-based direct communication design feature. MPI requires that Isend and Irecv TCP traffic flow between nodes asynchronously. VolpexMPI's *pull* model capitalizes on asynchronous message passing for determination of message availability from targeted replicas. Functions implementing this design feature are in the SockLib module.

VolpexMPI needs to handle messages that are sent and received to self. This design feature is directly in-line with the MPI Standard. In fact, the features of receiver-based direct communication and distributed sender-based logging have to be circumventing to enforce efficient send and receive to self. This functionality is implemented in the Point-to-Point module.

VolpexMPI needs to handle garbage collection for the memory buffers. The storage of all messages in a local send buffer heightens the need to actively manage allocated memory. This design statement supports receiver-based direct communication and distributed sender-based logging. Functions implementing this design feature are in all of the modules.

VolpexMPI needs to utilize MPI message metadata (count, datatype, source or destination, tag, communicator, and reuse number) to search the sent messages buffer. The sent messages buffer is designed as a circular buffer to allow for easy size adjustment.

The circular design also allows for backward searches from most recent entry to minimize search times, aiding in application progress at the rate of the fastest replica. This design statement supports receiver-based direct communication and is implemented in the Buffer Management module.

VolpexMPI needs to handles reuse of message tags. This design feature is directly inline with the MPI Standard in that VolpexMPI must protect for the fact that the user is allowed to reuse a tag number at will and the library must handle messages accordingly. This design statement supports the distributed sender-based logging feature and is implemented in the Point-to-Point and Buffer Management modules.

VolpexMPI is a layered design that is intended to be compiled and dynamically linked to the user's MPI application which is the top layer. The layered architecture of the library, shown in Figure 3.1 is divided into 3 major layers with several distinct modules in the lower layers that are implemented based on the design statements just presented.

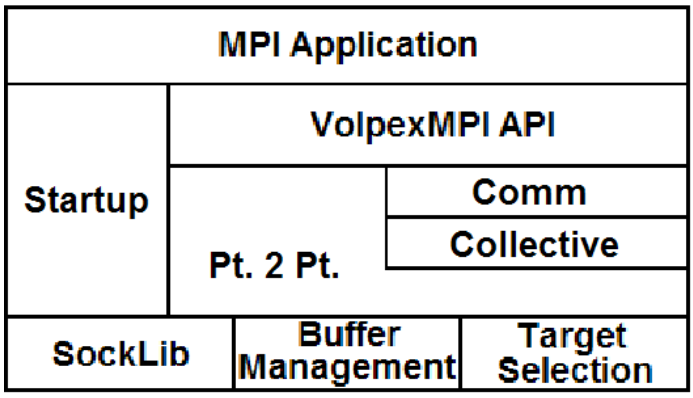

Figure 3.1: VolpexMPI Architecture

3.2.1 Startup Module

The startup of a VolpexMPI application utilizes a customized `mpirun` application which takes the desired replication level as a parameter, in addition to the number of MPI processes, the name of the executable, and a list of hosts where the processes shall be started. As of today, the startup mechanism relies on secure shell operations. We have extended the startup functionality to handle Condor pools and we anticipate to extend this section of the library in the near future also handle BOINC clusters. `mpirun` has furthermore the functionality to inform all MPI processes about their rank, the group they belong to, as well as the information required by a MPI process to contact any other process within this application.

3.2.2 VolpexMPI API Module

The purpose of the VolpexMPI API is to allow users to write MPI programs in either C or FORTRAN. The API utilizes weak pragma declarations to include the main variants for FORTRAN function naming including all upper case naming and the use of trailing underscoring by some compilers.

3.2.3 Point-to-Point Module

The point-to-point communication module of VolpexMPI has to be designed for MPI processes with multiple replicas in the system. This is required in order to handle the main challenge of grids built from idle desktop computers, namely the fact that processes are considered fundamentally unreliable. A process might go away for no obvious reason, such as the owner pressing a button on the keyboard. From the communication perspective, the library has two main goals: (I) avoid increasing the number of messages on the

fly by a factor of $n_{replicas} \times n_{processes}$, i.e., every process sending each message to every replica, and (II) make the progress of the application correspond to the fastest replica for each process.

In order to meet the first goal, the communication model of VolpexMPI employs a receiver initiated message exchange between processes where data is pulled by the receiver from the sender. In this model, the sending node only buffers the content of a message locally, along with the message envelope. Furthermore, it posts for every replica of the corresponding receiver rank, a non-blocking, non-expiring receive operation. When contacted by a receiver process about a message, a sender participates in the transfer of the message if it is buffered otherwise informs the receiver that the message is not available.

The receiving process polls a potential sender and waits then for the data item or a notification that the data is not available. As of today, VolpexMPI does not support wildcard receive operations as an efficient implementation poses a significant challenge. A straight-forward implementation of `MPI_ANY_SOURCE` receive operations is possible, but the performance would be significantly degraded compared to non-wildcard receive operations.

Since different replicas can be in different execution states, a message matching scheme has to be employed to identify which message is being requested by a receiver. For deterministic execution, a simple scheme that timestamps messages by counting the number of messages exchanged between pairs of processes is applied based on the tuple (communicator id, message tag, sender rank, receiver rank). These timestamps are also used to monitor the progress of individual process replicas for resource management. Furthermore, a late replica can retrieve an older message with a matching logical timestamp, which allows restart of a process from a checkpoint.

3.2.4 Collective Module

An implementation of each of the MPI collective functions built into VolpexMPI will be described in more detail in the next chapter on the performance-based architecture. The functions build upon the point-to-point module functions and capitalize on the replicated nodes and target selection to successfully complete collective operations.

3.2.5 Communicator Module

Similarly, implementations of each of the MPI communicator functions built into Volpex-MPI will be described in more detail in the next chapter. These functions build upon both the collective module functions and the point-to-point module functions to capitalize on the replicated nodes and target selection to successfully complete communicator operations.

3.2.6 Buffer Management Module

The buffer management module provides the functionality to store and retrieve a MPI message based on the tuple described above. An important question is whether the message buffers on the sender processes must be maintained for the duration of execution or whether they can be cleared at some point. From the logical perspective, a message buffer can never be cleared due to the fact that, even if all replicas of a particular rank have received a given message, all of them might fail to finish the execution. Thus, a new replica of that process might have to be started, which would have to retrieve all messages. Our current approach employs a circular buffer where the oldest log entry is removed when the buffer is full. The long-term goal is to coordinate the size of the circular buffer with checkpoints of individual processes, which will allow guaranteed restarts with a bounded buffer size.

3.2.7 Target Selection Module

In order to meet the goal that the progress of an application correspond to the fastest replica for each process, the library has to provide an algorithm which allows a process to generate an order in which to contact the sender replicas. This is the main functionality provided by the target selection module. The algorithm utilized by the target selection module has to handle two seemingly contradicting goals: on one hand, it would be beneficial to contact the "fastest" replica from the performance perspective. On the other hand, the library does not want to slow-down the fastest replica by making it handle significantly larger number of messages, especially when a message is available from another replica. The specific goal, therefore, is to determine a replica which is "close" to the execution state of the receiver process. Currently the library utilizes a simple approach which identifies replicas into redundancy groups described in Section 3.3. A receiver tries to contact the first replica within its group and only contacts a replica of another group if its own replica does not respond within a given time slot. This is, however, a topic of active research with more sophisticated algorithms in the process of being implemented and tested.

3.2.8 SockLib Module

The data transfer module of VolpexMPI relies on a socket library utilizing non-blocking sockets. In the context of VolpexMPI, the relevant characteristics of this socket library are the ability to handle failed processes, on-demand connection setup in order to minimize the number of network connections, an event delivery system integrated into the regular progress engine of the socket library and the notion of timeouts for both communication operations and connection establishment. The latter feature will be used in future versions of VolpexMPI to identify replicas which are lagging significantly.

3.3 Nomenclature

The MPI specifications use rank and communicator groups to identify processes that communicate with each other. The rank is a non-negative integer with one of the processes identified as the root process. Usually, but not always, the root process is rank 0 (zero). Because we employ process replication, a new nomenclature is defined to identify replicated processes. Rank is identified as a alpha-numeric pair of a non-negative integer and a letter. For example, in the Figure 1.1 ”1,B” represents the rank 1 node of the second redundancy group and ”3,C” represents the rank 3 node of the third redundancy group. In this example, triply redundant fault tolerance is available.

Chapter 4

Performance-oriented Architecture

Based on the design described in the previous chapter, we have architected an all C version of the VolpexMPI library using asynchronous sockets. The intention is to understand and characterize the performance overhead associated with controlling multiple replicas of each MPI process. This all C architecture utilizes non-blocking sockets to allow for asynchronous data transfer operations. The buffers are also classical data structures such as circular linked lists and queues because these provide the highest performance in data storage and transfer. For this dissertation, the target selection uses a simple round-robin approach, but more efficient routines are already being tested by other researchers. Table 4.1 lists all of the MPI functions chosen for this implementation which were based on their usage in the NAS Parallel Benchmarks. The rest of this chapter will describe each MPI function and show a pseudo code snippet as well as describe the major data types and data structures. Finally, we give the results from testing this architecture in three distinct environments.

Table 4.1: MPI Functions Implemented in VolpexMPI to execute the NAS Parallel Benchmarks. The Frequency % refers to how often the MPI function is utilized in the NAS Parallel Benchmarks.

MPI Function	Frequency (%)	MPI Function	Frequency (%)
MPI_Irecv	14.4%	MPI_Comm_rank	2.5%
MPI_Send	10.6%	MPI_Init	2.5%
MPI_Isend	10.2%	MPI_Reduce	2.5%
MPI_Bcast	9.7%	MPI_Alltoall	1.7%
MPI_Wait	9.7%	MPI_Comm_dup	1.7%
MPI_Allreduce	7.2%	MPI_Comm_split	1.7%
MPI_Barrier	7.2%	MPI_Gather	1.7%
MPI_Abort	4.7%	MPI_Allgather	1.7%
MPI_Comm_size	4.2%	MPI_Recv	1.7%
MPI_Waitall	3.4%	MPI_Wtime	0.8%
MPI_Finalize	3.0%	MPI_Alltoallv	0.4%

4.1 MPI Functions

Volpex_Init - Volpex_Init initializes the Volpex MPI execution environment. Among the items initialized from within the startup module (MCFA_Init) are a request list for non-blocking message traffic, structures for tracking tag reuse, reading the configuration file in a local data structure called the Volpex_Proc_Array, initializing the sent messages buffer (Send_Buffer), and resolving the private IP address and hostname of the local processor for `MPI_COMM_WORLD`. The Volpex_Barrier function is called after the Volpex_Init function to ensure that all nodes within a group are prepared to leave the initialization function at the same time. As per the MPI Standard, the C implementation of Volpex_Init allows for command line arguments.

Volpex_Finalize - Volpex_Finalize terminates the Volpex MPI execution environment.

Volpex_Init

```
int Volpex_Init( argc, argv ){
   MCFA_Init()  /* calls SL_Init() */
   Volpex_Barrier(…)
}
```

Figure 4.1: Pseudo code for Volpex_Init Function

Along with freeing some data structures, the Volpex_Finalize function makes a Volpex_Barrier function call to promote the availability of a node's send buffer in the case of node failures or drop-offs in other redundant groups of nodes. Essentially, when the barrier is reached, the progress of outstanding message requests is updated and messages that are flagged as ready to send or receive are executed. After a call to MCFA_Finalize to initiate graceful node shutdown, the send buffer is deleted and all global data structures are freed.

Volpex_Finalize

```
int Volpex_Finalize(){
   Volpex_Barrier(…)
   MCFA_Finalize()  /* calls SL_Finalize() */
   Volpex_send_buffer_delete()
   Free_global_data()
}
```

Figure 4.2: Pseudo code for Volpex_Finalize Function

Volpex_Comm_size - Volpex_Comm_size determines the size of the group associated with a communicator. This function loops through the Volpex_Proc_Array created from the configuration file and determines the size of the group in the current redundancy level of the communicator. The value of size returned is stored in a global data structure for the particular MPI communicator.

Volpex_Comm_rank - Volpex_Comm_rank determines the rank of the calling process in the communicator. This function loops through the Volpex_Proc_Array created from the configuration file and determines the rank of the node in the current redundancy level of

Volpex_Comm_size

```
int Volpex_Comm_size( comm, size ){
   for loop over comm in VolpexProcArray
      *size = comm->size
}
```

Figure 4.3: Pseudo code for Volpex_Comm_size Function

the communicator. The value of rank returned is stored in a global data structure for the particular MPI communicator.

Volpex_Comm_rank

```
int Volpex_Comm_rank( comm, rank ){
   for loop over comm in VolpexProcArray
      *rank = comm->myrank
}
```

Figure 4.4: Pseudo code for Volpex_Comm_rank Function

Volpex_Isend - In the *pull* model, Volpex_Isend copies the message to the local Send_Buffer data structure. However, the first action taken in the function is to determine if the send is to self. If this condition is true, the message is copied to the target memory location and is not sent to the local Send_Buffer. Otherwise, the length of the message is determined and the tag is checked to keep track of tag reuse by the user's MPI application. Notice in Figure 4.5 the use of the SL_Isend. SL_Isend is a non-blocking function in the SockLib module that conducts the actual movement of bytes either to new memory locations or between nodes using the TCP write function available in C. Next, depending on the redundancy requested, the potential data receivers (i.e., targets) to send to are identified. For all potential data receivers, the function creates MPI_Requests, adds each to a request list, and then calls SL_Irecv because all potential data receivers could request the message if failures occur in their redundancy groups. SL_Irecv is also a non-blocking function of the SockLib module for TCP memory read and transfer functions. Essentially, to send a message, the

sender must first receive a message from the potential reciever containing the attributes of the original message (i.e, the count, datatype, target, communicator and tag).

Volpex_Isend

```
int Volpex_Isend( buf, count, datatype, destination, tag, comm, request ){
   if dest == myrank
      SL_Isend(…)
   for loop over redundant nodes in VolpexProcArray
      SL_Irecv(…)
   send_buffer_insert(…)
   Volpex_progress()
}
```

Figure 4.5: Pseudo code for Volpex_Isend Function

Volpex_Send - Volpex_Send is implemented by calling Volpex_Isend with an immediate call to Volpex_Wait to force the MPI_Request to return before proceeding.

Volpex_Send

```
int Volpex_Send( buf, count, datatype, dest, tag, comm ){
   Volpex_Isend(…)
   Volpex_Wait(…)
}
```

Figure 4.6: Pseudo code for Volpex_Send Function

Volpex_Irecv - As described in Volpex_Isend, in order to use Volpex_Irecv to receive a message in the *pull* model, a node must first use SL_Isend to determine if the source node is available and has the message desired. Again, the first action taken by the function is to determine if the receive is from self. If this condition is true, the message is copied from the target memory location by a call to SL_Irecv. Otherwise, the length of the message is determined and the tag is checked to keep track of tag reuse by the user's MPI application. Next, depending on the redundancy requested, the potential data sender (i.e., sources) to receive from are identified. For each potential data sender, the function calls SL_Isend

because each potential data sender should have a SL_Irecv waiting to be requested for the message if failures occur in the redundancy groups. Essentially, to receive a message, the receiver must first send a message to each potential sender containing the attributes of the original message (i.e, the count, datatype, target, communicator, and tag). The receiver does not try to contact all potential sender simultaneously but instead loops over the potential redundant senders until one is found to be available and has the message desired. Future versions of VolpexMPI will have the receiver contact the sending node who most closely matches its own progress.

Volpex_Irecv

```
int Volpex_Irecv( buf, count, datatype, source, tag, comm, request ){
   if source == myrank
      SL_Irecv(…)
   for loop over redundant nodes in VolpexProcArray
      SL_Isend(…)
   Volpex_progress()
}
```

Figure 4.7: Pseudo code for Volpex_Irecv Function

Volpex_Recv - Volpex_Recv is implemented by calling Volpex_Irecv with an immediate call to Volpex_Wait to force the MPI_Request to return before proceeding.

Volpex_Recv

```
int Volpex_Recv( buf, count, datatype, source, tag, comm, status ){
   Volpex_Irecv(…)
   Volpex_Wait(…)
}
```

Figure 4.8: Pseudo code for Volpex_Recv Function

Volpex_Bcast - Volpex_Bcast broadcasts a message from the process with rank ”root” to all other processes of the communicator group. This function utilizes a loop to have the ”root” process call Volpex_Send and each other node in the communicator group call

Volpex_Recv to accomplish the broadcast. These calls will then call Volpex_Isend and Volpex_Irecv to employ the built-in redundancy described previously to accomplish the broadcast across the replicas.

Volpex_Bcast

```
int Volpex_Bcast( buf, count, datatype, root, comm ){
   for loop over comm in VolpexProcArray
      if root
         Volpex_Send(…)
      if not root
         Volpex_Recv(…)
}
```

Figure 4.9: Pseudo code for Volpex_Bcast Function

Volpex_Reduce - Volpex_Reduce reduces values on all processes to a single value. This function first has all non-root processes execute a Volpex_Send to pass each buffer to the root node for computation. The root node creates temporary storage space to hold buffer inputs from each Volpex_Recv called for each node in the communicator. Each buffer is processed based on the computation indicated in the user MPI application call to the function. The VolpexMPI Library currently handles MPI_MAX, MPI_SUM, MPI_MIN, and MPI_PROD for several data types described in Section 4.2.

Volpex_Reduce

```
int Volpex_Reduce( sendbuf, recvbuf, count, datatype, op, root, comm ){
   Volpex_Send(…)
   if root
      for loop over comm in VolpexProcArray
         Volpex_Recv(…)
         Reduction operation
}
```

Figure 4.10: Pseudo code for Volpex_Reduce Function

Volpex_Allreduce - Volpex_Allreduce is accomplished with a call to Volpex_Reduce and then a call to Volpex_Bcast to return the result of the reduction to all nodes.

Volpex_Allreduce

```
int Volpex_Allreduce( sendbuf, recvbuf, count, datatype, op, comm ){
   Volpex_Reduce(…)
   Volpex_Bcast(…)
}
```

Figure 4.11: Pseudo code for Volpex_Allreduce Function

Volpex_Wait - Volpex_Wait first checks to see if the request to be fulfilled is set to MPI_REQUEST_NULL. This is normally the case for Volpex_Isend and the Volpex_Wait function returns MPI_SUCCESS immediately. If the Volpex_Wait is being invoked to check on a Volpex_Irecv, the function enters into a while loop within the Volpex_progress engine until the requested data is received. Once the request is fulfilled, the request data structure is updated as well as the status data structure. This function is callable by both the end user MPI application as well as internally by other VolpexMPI library functions.

Volpex_Wait

```
int Volpex_Wait( request, status ){
   while(1)
      Volpex_progress()
}
```

Figure 4.12: Pseudo code for Volpex_Wait Function

Volpex_Waitall - Volpex_Waitall is accomplished by a simple loop calling the Volpex_Wait function. Volpex_Waitall does not return until all requests have been fulfilled. The order of return of the requests is arbitrary.

Volpex_Barrier - Volpex_Barrier is accomplished by tracking, per node and per communicator, the number of barriers passed. This information is passed by calls to Volpex_Send by non-root nodes and calls to Volpex_Recv by the root node of the communicator. If the barrier number passed is mismatched an error message is generated. Once the root node has collected the same barrier number for all nodes, it sends the value back and execution

Volpex_Waitall

```
int Volpex_Waitall( count, requests[…], statuses[…] ){
   for loop over requests[…]
      Volpex_Wait(…)
}
```

Figure 4.13: Pseudo code for Volpex_Waitall Function

continues.

Volpex_Barrier

```
int Volpex_Barrier( comm ){
   if not root
      Volpex_Send(…)
   if root
      Volpex_Recv(…)
   Volpex_progress
   if root
      Volpex_Send(…)
   if not root
      Volpex_Recv(…)
}
```

Figure 4.14: Pseudo code for Volpex_Barrier Function

Volpex_Abort - If Volpex_Abort is called, the node process terminates. Generally this would cause all nodes to attempt to terminate, but with VolpexMPI, this would only cause this node to terminate and not the other nodes within the same replica group. These nodes would continue executing with a replica of the terminated process.

Volpex_Abort

```
int Volpex_Abort( comm, errorcode ){
   exit(…)
}
```

Figure 4.15: Pseudo code for Volpex_Abort Function

Volpex_Alltoall - Volpex_Alltoall sends data from all to all processes. Volpex_Alltoall

sends distinct data to each of the receivers. Volpex_Alltoall calls Volpex_Isend and Volpex_-Irecv which allows for redundancy in collecting each block of data. The jth block received from process i or any of its redundant nodes is received by process j and is placed in the ith block of recvbuf. The function completes when Volpex_Waitall is called and all requests complete.

Volpex_Alltoall

```
int Volpex_Alltoall( sendbuf, sendcount, sendtype, recvbuf, recvcnt, recvtype,
                     comm ){
   for loop over comm in VolpexProcArray
      Volpex_Isend(…)
      Volpex_Irecv(…)
      Volpex_Waitall(…)
}
```

Figure 4.16: Pseudo code for Volpex_Alltoall Function

Volpex_Alltoallv - Volpex_Alltoallv sends data from all to all processes where each process may send a different amount of data and provide displacements for the input and output data. The location of data for the send is specified by sdispls and the location of the placement of the data on the receive side is specified by rdispls. Volpex_Alltoallv calls Volpex_Isend and Volpex_Irecv which allows for redundancy in collecting each variable block of data. The function completes when Volpex_Waitall is called and all requests complete.

Volpex_Alltoallv

```
int Volpex_Alltoall( sendbuf, sendcnts, sdispls, sendtype, recvbuf, recvcnts,
                     rdispls, recvtype ,comm ){
   for loop over comm in VolpexProcArray
      Volpex_Isend(…)
      Volpex_Irecv(…)
      Volpex_Waitall(…)
}
```

Figure 4.17: Pseudo code for Volpex_Alltoallv Function

Volpex_Gather - Volpex_Gather collects together values from a group of processes. Here the VolpexMPI Library uses Volpex_Send and Volpex_Recv to gather the data in the same manner as Volpex_Bcast. Unlike Volpex_Bcast the collected data are not passed back to the non-root processes.

Volpex_Gather

```
int Volpex_Gather( sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                   root, comm ){
   if not root
      Volpex_Send(…)
   if root
      Volpex_Recv(…)
}
```

Figure 4.18: Pseudo code for Volpex_Gather Function

Volpex_Allgather - Volpex_Allgather is accomplished by executing a Volpex_Gather to collect the data and then executing a Volpex_Bcast to send the collected data to the non-root processes.

Volpex_Allgather

```
int Volpex_Allgather( sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
                      root, comm ){
   Volpex_Gather(…)
   Volpex_Bcast(…)
}
```

Figure 4.19: Pseudo code for Volpex_Allgather Function

Volpex_Comm_dup - Volpex_Comm_dup utilizes Volpex_Allreduce to determine the next available communicator. Once the value is determined and returned to all processes, the content of the current communicator is copied to the new communicator within the Volpex_Proc_Array. As with the other VolpexMPI functions, Volpex_Comm_dup ultimately uses Volpex_Isend and Volpex_Irecv which check for loss of communication with other nodes and uses redundant nodes if necessary to collect or distribute data.

Volpex_Comm_dup

```
int Volpex_Comm_dup( comm, newcomm ){
   Volpex_Allreduce(…)
   Create newcomm
}
```

Figure 4.20: Pseudo code for Volpex_Comm_dup Function

Volpex_Comm_split - Volpex_Comm_split creates new communicators based on colors and keys. In much the same manner as other MPI libraries, VolpexMPI uses a Volpex_Allreduce, a Volpex_Allgather, and a qsort to gather and order the keys by color. The function then creates the appropriate Volpex_Proc_Array data and checks the communicator size and node rank with Volpex_Comm_size and Volpex_Comm_rank.

Volpex_Comm_split

```
int Volpex_Comm_split( comm, color, key, newcomm ){
   Volpex_Allreduce(…)
   Volpex_Allgather(…)
   qsort over colors and keys
   for loop over comm in VolpexProcArray
      Create newcomm
}
```

Figure 4.21: Pseudo code for Volpex_Comm_split Function

Volpex_progress - Volpex_progress is an all new internal function in VolpexMPI that is invoked whenever there is a need to check for updates to both send and receive MPI_requests. The function loops over the Request_List, described in an upcoming section, which is a list send and receive requests. The function then determines if a request is for a send or receive operation, and a SL_Test is then run against the request. SL_Test is another function of the SockLib module that tests for completion of TCP read and write operations. If the request is to send data, the send buffer is searched for the message, and if found, a SL_Isend is executed. If the request is to receive data, the system attempts to invoke a SL_Irecv to connect to the sending process. If successful, the message is transferred. If not successful,

the Volpex_Irecv is invoked to attempt to connect to any available redundant nodes. Calls to Volpex_progress are strategically placed in many functions of the library to facilitate message traffic transfer between nodes.

Volpex_progress

```
int Volpex_progress(){
   for loop{
      if send operation{
         SL_Test(…) for completion
         if complete{
            send_buffer_search(…)
            SL_Isend(…)
            free request
         }
      }
      if receive operation{
         SL_Test(…)
         if complete
            SL_Irecv(…)
         else
            Volpex_Irecv(…)
      }
   }
}
```

Figure 4.22: Pseudo code for Volpex_progress Function

Volpex_Redundancy_Barrier - This is an all new function in the VolpexMPI Library but has been removed in the production library. It was thought this function might be necessary to enforce a barrier for nodes of the same rank across redundant groups. Volpex_Redundancy_Barrier was utilized internally by the VolpexMPI library and invoked only in the Volpex_Init and Volpex_Finalize functions. In Volpex_Init the redundancy barrier was the last function called and ensures that, at least at the start of MPI application execution, all replica groups are up and running. In Volpex_Finalize, the redundancy barrier was called to ensure all surviving replicas can access other replica's Send_Buffer for data who may have already reached Volpex_Finalize and would otherwise have exited. This function, essentially a two-stage commit protocol, will be unnecessary in future versions of VolpexMPI because target selection and availability of data is being reworked utilizing checkpoint-restart

mechanisms.

Volpex_Redundancy_Barrier

```
int Volpex_Redundancy_Barrier( comm, rank ){
   for loop over redundant nodes in VolpexProcArray
      Volpex_Isend(…)
      Volpex_Irecv(…)
   while loop over requests created
      SL_Test(…)
      Volpex_progress()
}
```

Figure 4.23: Pseudo code for Volpex_Redundancy_Barrier Function

4.2 Data Types and Data Structures

Due to the initial focus on testing the NAS Parallel Benchmarks, the VolpexMPI Library can handle only the most common MPI data types. Currently, the library handle arrays of bytes, integers, reals, floats, doubles and double_complex data types as shown in Table 4.2.

There are multiple different data structures that VolpexMPI requires to execute. These data structures range from simple arrays to queues to double-linked circular lists. The SockLib layer primarily uses queues for message tranfers over TCP. These are defined as a send queue, a receive queue, a unexpected msgs queue, a send complete queue, and a receive complete queue. Several of the major VolpexMPI data structures are described in this section. All data structures are defined in C header files and initialization occurs in the Volpex_Init function.

Table 4.2: MPI Data Types (shaded) implemented in VolpexMPI to execute the NAS Parallel Benchmarks.

C Data Types		Fortran Data Types	
MPI_INT	signed int	MPI_INTEGER	integer
MPI_FLOAT	float	MPI_REAL	real
MPI_DOUBLE	double	MPI_DOUBLE_PRECISION	double precision
MPI_CHAR	signed char	MPI_CHARACTER	character(1)
MPI_BYTE	8 binary digits	MPI_BYTE	8 binary digits
MPI_SHORT	signed short int		
MPI_LONG	signed long int		
MPI_UNSIGNED_CHAR	unsigned char		
MPI_UNSIGNED_SHORT	unsigned short int		
MPI_UNSIGNED	unsigned int		
MPI_UNSIGNED_LONG	unsigned long int		
MPI_LONG_DOUBLE	long double		
		MPI_COMPLEX	complex
MPI_DOUBLE_COMPLEX	double _Complex	MPI_DOUBLE_COMPLEX	double complex
		MPI_LOGICAL	logical
MPI_PACKED		MPI_PACKED	

4.2.1 Volpex_Proc_Array

The first data structure required is the Volpex_Proc_Array which captures all of the basic information required to establish TCP connections between nodes within a communicator. The current implementation is a linked list array of Volpex_Proc data structs. Each Volpex_Proc data struct in the Volpex_Proc_Array contains information on all replicated sets of nodes. The Volpex_Proc data structs identifies an integer id, the host ip address, the port, the VolpexMPI rank as described in Chapter 3 and the TCP connection state. Each node also keeps a single data structure associated with each communicator that holds the communicator's size, the node's rank in this communicator, and the last barrier by integer number that the node has passed through. When a Volpex_Comm_dup is executed, it is data from this data structure that are copied to duplicate the communicator.

4.2.2 Request_List

VolpexMPI is designed for asynchronous message passing with non-blocking sockets, so a list of the requested, but uncompleted, messages must be tracked. The Volpex_progress function makes use of the Request_List data structure to track requested message progress. The Request_List data structure is currently implemented as an array of structs. Some of the items contained in a request are the reqnumber, the intended message header array, the returned message header array, the request type, the flag set at completion, and the actual buffer of data received from a Volpex_Irecv. The message header arrays contain the length calculated as the count multiplied by MPI data type, destination or source rank, tag, communicator, and an integer that tracks tag reuse within the user MPI application.

4.2.3 Send_Buffer

In order to allow redundant nodes to have access to a message for some period of time, the message is stored in the Send_Buffer. The Send_Buffer in VolpexMPI is defined as a circular doubly-linked list of structs. The struct contains items such as back and forward message pointers, a message counter, the message header, the actual message buffer, and an array for tracking the requests made for the stored message. The functions that manage the send buffer perform tasks such as initialization, deletion, insertion, and searches. The send buffer is initialized to a pre-defined size (e.g., 500 messages), and any message will remain in the buffer until it is overwritten as the circular linked list is traversed. This allows some persistence while not forcing storage of all messages generated by a user MPI application. The Send_Buffer is only deleted during Volpex_Finalize. New message attributes and buffers are inserted into the existing circular linked list with the send_buffer_insert function. Most important is the send_buffer_search function that is called by Volpex_progress. This function executes a do-while loop to ensure one complete reverse traversal of the send

buffer while attempting to locate a message. If the loop completes, the progress engine moves on and the message is searched for again by Volpex_progress checks throughout the library.

4.3 Experiments and Results

This section describes the experiments with the C-based VolpexMPI library and the results obtained. VolpexMPI has now been deployed on a dedicated cluster, a vitualized cluster[47], as well as Condor pools as shown in Figure 4.24. Although VolpexMPI is designed for desktop computer grids and volunteer environments, the experiments are shown in reference to runs on the dedicated cluster in order to determine the fundamental performance characteristics of VolpexMPI. This ensures the results are stable and reproducible.

4.3.1 Dedicated Cluster Test Results

The dedicated cluster utilizes 29 compute nodes with 24 of them having a 2.2 GHz dual core AMD Opteron processor and 5 nodes having two 2.2GHz quad-core AMD Opteron processors. Each node has 1 GB main memory per core and is network connected by 4xInfiniBand as well as a 48 port Linksys GE switch. For evaluation, we utilize the Gigabit Ethernet network interconnect of the cluster to compare VolpexMPI run times to Open MPI v1.2.6. and examine the impact of replication and failure on performance.

First, we document the impact of the VolpexMPI design on the latency and the bandwidth of communication operations. For this, we ran a simple ping-pong benchmark using both Open MPI and VolpexMPI on the dedicated cluster. The results shown in Figure 4.25 indicate that the receiver-based communication scheme used by VolpexMPI can achieve close to 80% of the bandwidth achieved by Open MPI. The latency for a 4 byte message

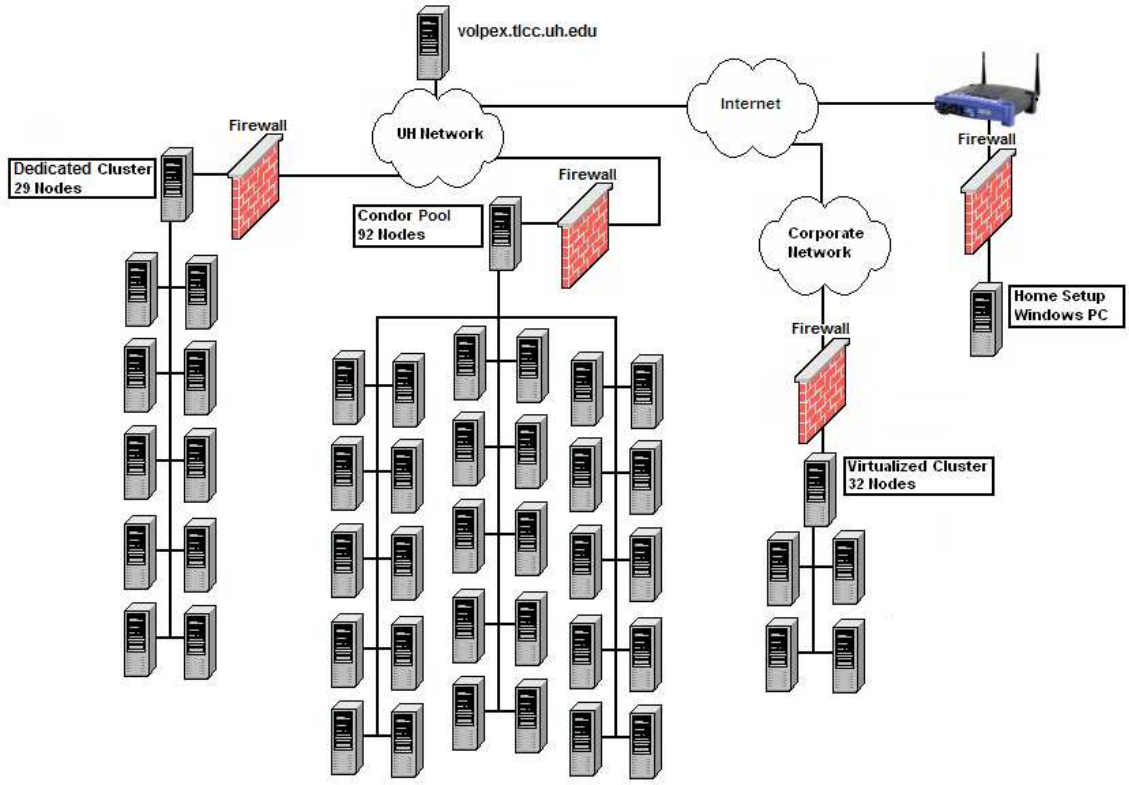

Figure 4.24: VolpexMPI was tested on a dedicated cluster, a virtualized cluster, and a condor pool. For the test cases, the jobs were initiated from each cluster front end machine. Future testing could be initiated from a web server like the one shown at the top of the figure to include various volunteered nodes from each cluster and even home desktop computers behind firewalls and routers.

increases from roughly 0.5ms with Open MPI to 1.8ms with VolpexMPI. This is not surprising as receiver-based communication requires a ping-pong exchange before the actual message exchange.

Next, the NAS Parallel Benchmarks are executed for various process counts and data class set sizes. For each experiment, the run times were captured as established and reported in each NAS Parallel Benchmark with the normal `MPI_Wtime` function calls for start and stop times. Since VolpexMPI targets the execution of applications with moderate number of processes, we present results obtained for 8 process and 16 process scenarios.

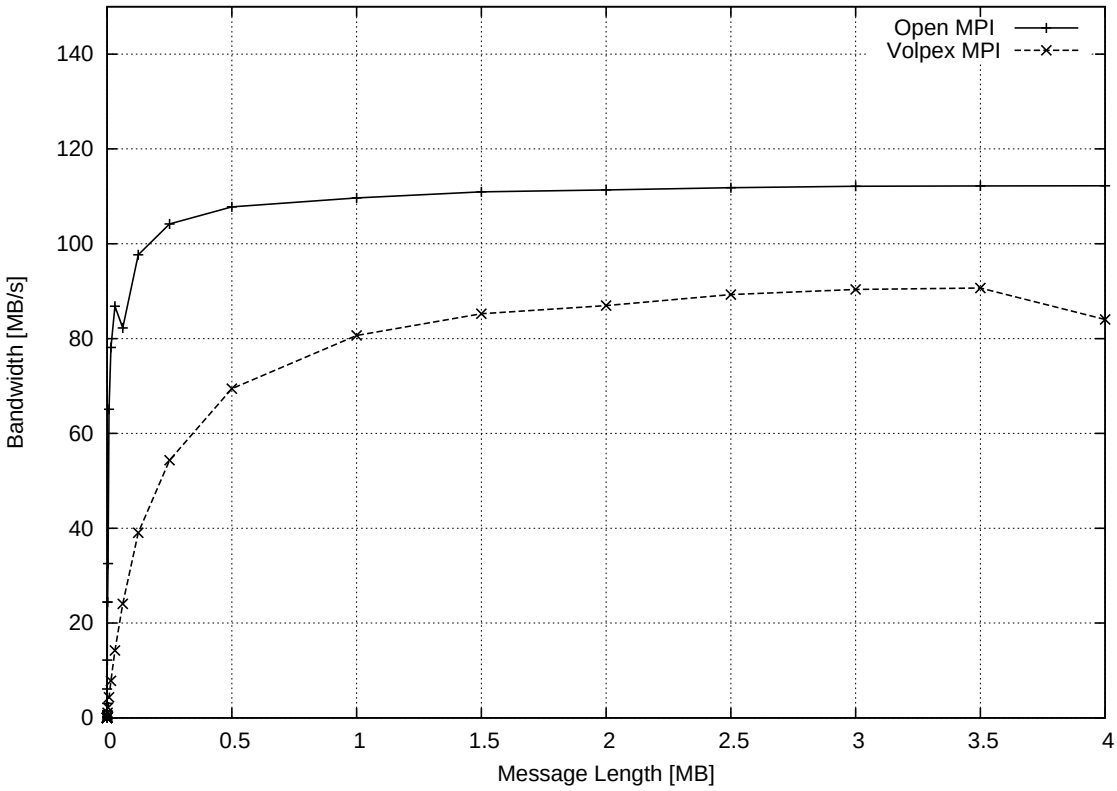

Figure 4.25: Bandwidth comparison of OpenMPI and VolpexMPI using a Ping-Pong Benchmark.

Figure 4.26 and Figure 4.27 show results for runs of 8 processes and 16 processes, respectively, utilizing the Class B data sets for six of the NAS Parallel Benchmarks. We have excluded LU and MG from our experiments due to their use of `MPI_ANY_SOURCE` which is not currently supported in VolpexMPI. These reference executions did not employ redundancy (x1). The run times for Open MPI are shown for comparison in the bar graph. All times are noted as normalized execution times with a reference time of 100 for Open MPI.

The overhead incurred in the VolpexMPI implementation is virtually non-existent for

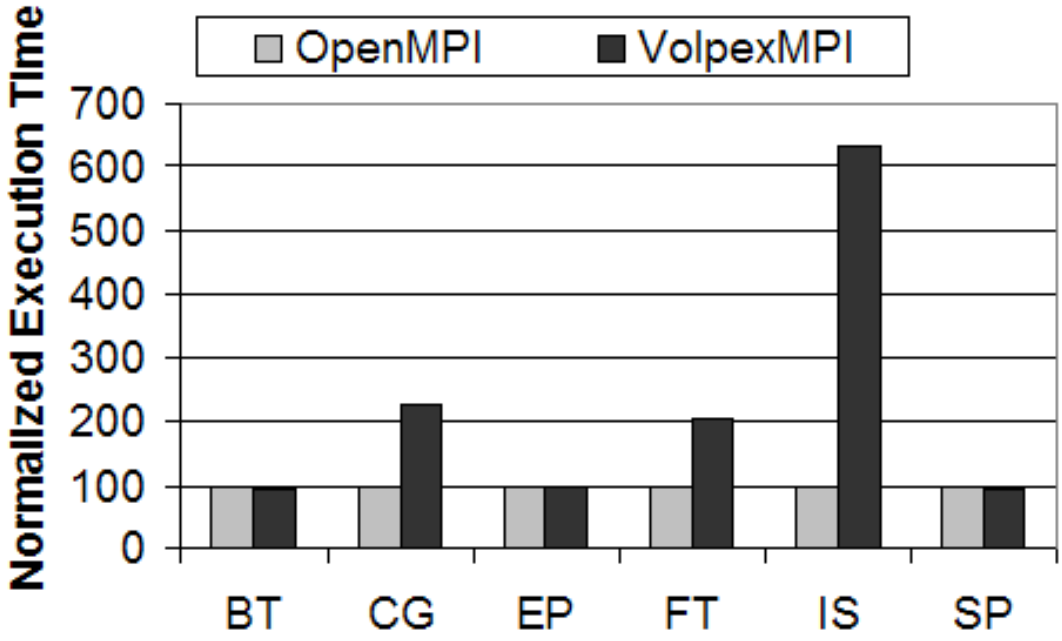

Figure 4.26: Comparison of OpenMPI to VolpexMPI for Class B NAS Parallel Benchmarks using 8 Processes on a dedicated cluster.

BT, SP, and EP for the 8 and 16 processes, except that SP shows a noticeable overhead of 45% for 16 processes. The overhead for CG, FT, and IS is significantly higher due to a variety of reasons, such as a greater use of collective calls such as `MPI_Alltoall(v)`, and in the case of IS, a ratio of computation to communication which is unfavorable to higher-latency environments. This also documents the fact that the class of applications considered suitable for execution with VolpexMPI have to follow a sparse communication scheme, i.e., a process should optimally only communicate with a small number of other processes, and should have a favorable communication to computation ratio. These requirements broadly hold for BT, SP, and EP, but not necessarily for CG, FT, and IS.

Next, we document the effect of executing an application with multiple copies of each MPI process. Figure 4.28 shows the normalized execution times of VolpexMPI for the 8 process NAS Parallel Benchmarks running with no (same as single) redundancy (x1),

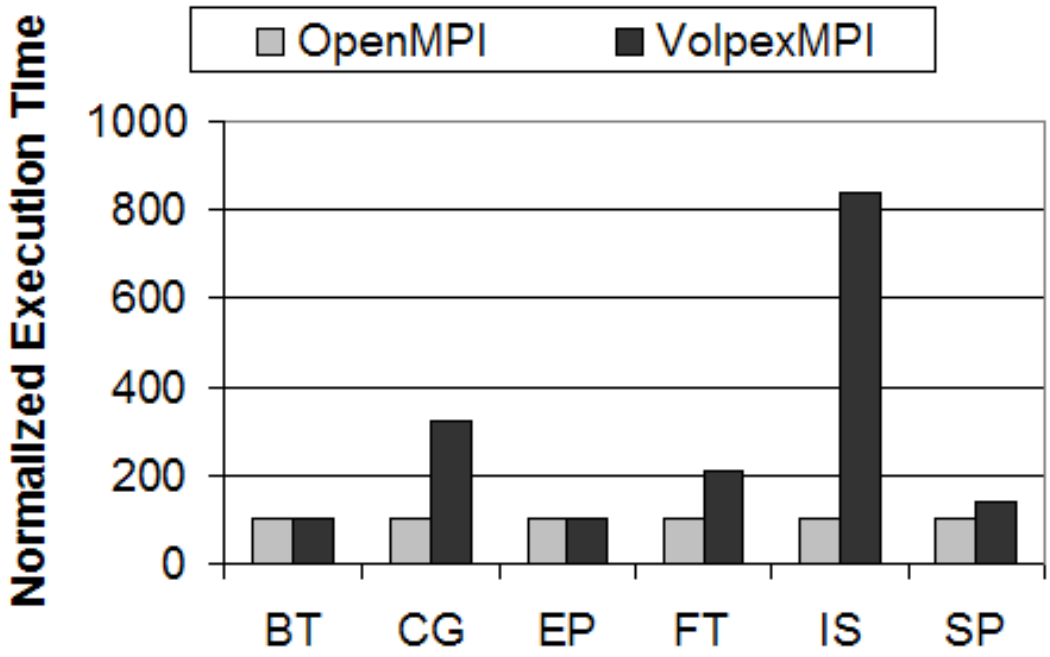

Figure 4.27: Comparison of OpenMPI to VolpexMPI for Class B NAS Parallel Benchmarks using 16 Processes on a dedicated cluster.

double redundancy (x2), and triple redundancy (x3). The results indicate that, for most benchmarks, the overhead due to redundant execution is minimal if no failure occurs, i.e. executing multiple copies of each MPI processes does not impose a significant performance penalty in the VolpexMPI *pull* model. Note that this is a significant improvement over the replication-based related work in the field. The benchmarks that show some sensitivity to replication are CG and SP, and the reasons are currently under investigation. Since Open MPI is not designed for utilizing redundant nodes, there are no directly comparable results for the double redundancy (x2) and triple redundancy (x3) runs.

Finally, we document the performance impact of a process failure for the NAS Parallel Benchmarks when using VolpexMPI. For this, we inserted into the source code of each benchmark some statements which terminate the execution of the second replica of rank 1 in `MPI_COMM_WORLD`, emulating a process failure. All processes communicating with the

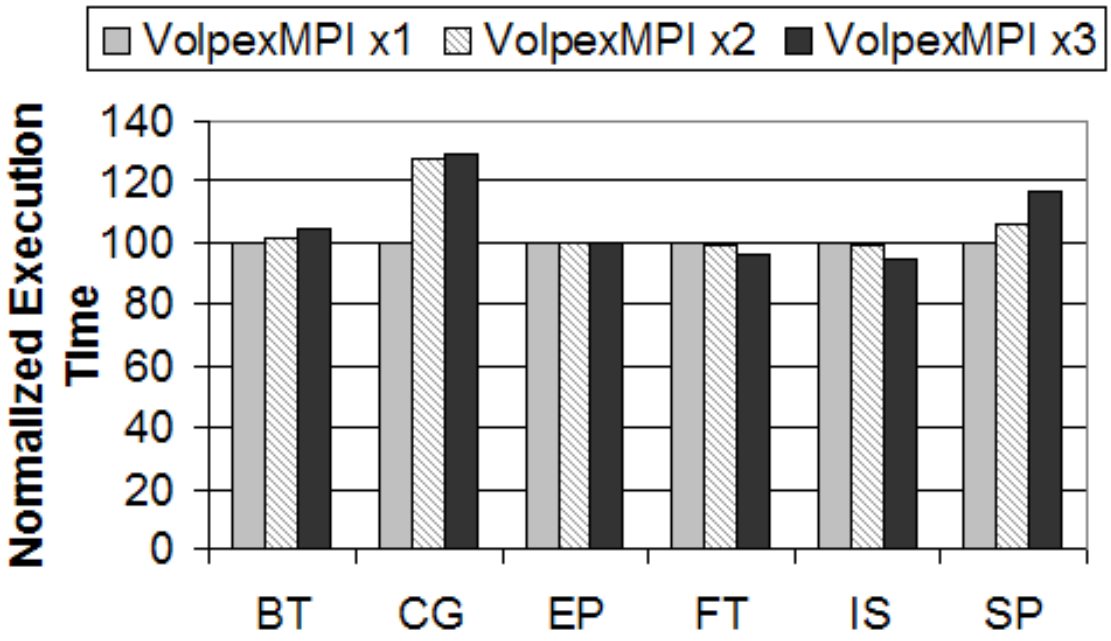

Figure 4.28: Comparison of VolpexMPI execution times for 8 MPI processes with varying degree of replication.

terminated process will thus have to repost all pending communication operations to the only remaining replica of process 1. This test case represents one of the worst case scenarios for VolpexMPI, because the number of processes communicating with a single process doubles at runtime. Killing more than one process would actually relieve the remaining processes with rank 1, since the number of communication partners is reduced.

The results shown in Figure 4.29 show virtually no overhead in the scenario outlined above compared to the fault-free execution of the same benchmark using double redundancy. There are two potential sources for overhead. The first comes from the fact that the surviving process with rank 1 is being queried for data items by the second group of processes. Second, detecting the process failure is, as of today, based on a timeout mechanism or the break-down of a TCP connection. However, since the correct result of the simulation is available as soon as any replica of each process finishes the execution, these overheads

might not necessarily show up in the final result.

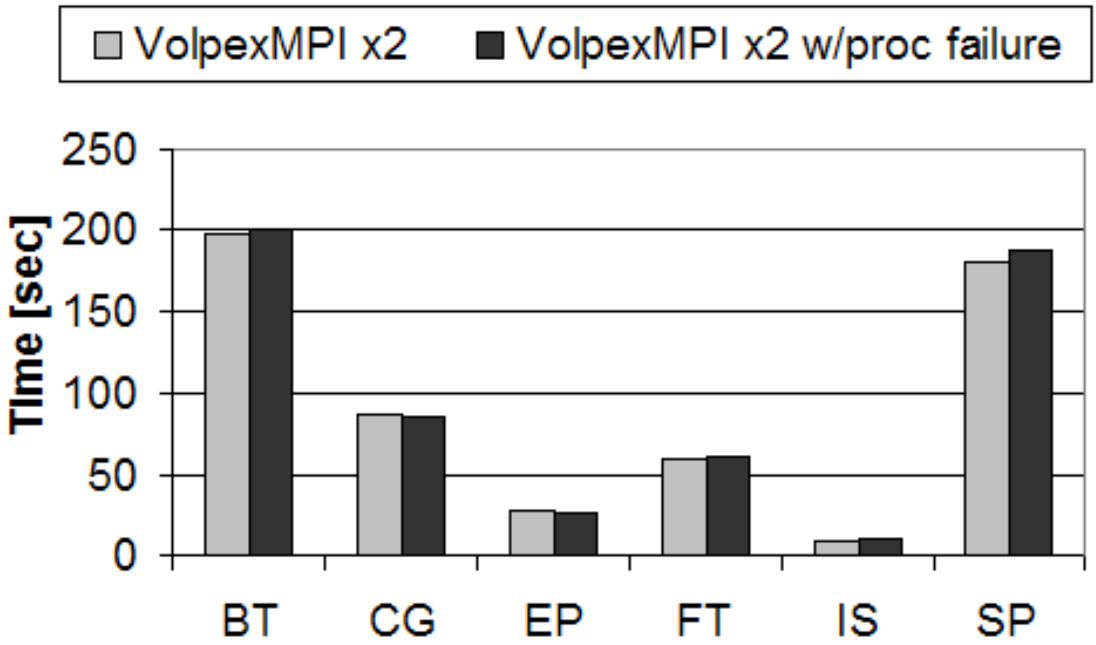

Figure 4.29: Comparison of VolpexMPI execution times for 8 MPI processes in case of a process failure.

4.3.2 Performance Impact of Virtualized Compute Environments

Virtualization allows execution of multiple instances of the same or different operating systems simultaneously on a single machine. The main advantage of virtualization from the end-users perspective comes from the fact that an application is typically deployed with its own operating system. Thus, different applications running on the same host are “shielded” from each other and have no obvious way how to interfere with each other. Virtualization is therefore one of the main reasons for the success of cloud computing in commercial settings. The same concept also increases the security of both volunteer applications and clients from malicious hosts/applications. However, virtualization imposes a performance

penalty for both compute operations as well communication due to the fact that access to shared resources, such as networking card or hard drives, has to be processed through the virtual machine monitor. Due to the increasing popularity of virtualization in volunteer computing, the goal of this section is to evaluate the performance implication that an application using VolpexMPI would face compared to non-virtualized setting. Furthermore, virtual machine monitors allow control of certain parameters of the execution environment such as processor frequency or network bandwidth. A second goal of this section therefore is to explore the sensitivity of the VolpexMPI architecture to those parameters, emulating more closely the realities of volunteer computing.

The virtualized cluster utilizes 8 physical nodes hosting 33 virtual compute nodes. The physical nodes are IBM Blade HS21 XM servers with 2 Intel Xeon E5450 quad-core processors, 2 Broadcom 1GbE NICs and 32GB DDR2 RAM. Each virtual machine is specified at 1 GB RAM, 3.0 GHz CPU, 10Gb/sec internal networking and 1Gb/sec external networking. The systems run Linux Redhat 5.3 and Vmware 4.0. For this evaluation we utilize the virtual cluster to compare VolpexMPI run times to the Open MPI run times on the dedicated cluster from the previous section to determine the impact of replication and volunteer-level performance.

First, the NAS Parallel Benchmarks are again executed for various process counts and data class set sizes on the virtualized cluster. For each experiment, the run times were captured as established and reported in the NAS Parallel Benchmark with the normal `MPI_Wtime` function calls for start and stop times.

The first question that we try to explore is dealing with the performance impact of virtual machines on the application. For this, we ran the NAS Parallel Benchmarks for a fixed number of MPI processes but varied the number of virtual machines executed on a single node. The results indicate that running up to four virtual machines per physical node (which has 8 cores all in all) best matched the performance of the dedicated cluster. Testing

showed that having five virtual machines per node already added a performance overhead of 50% to the execution time. Oversubscribing the resources [25], a common technique in cloud computing, e.g. by running 16 virtual machines per physical node, showed a significant performance slowing by 1000%. Therefore, for the remainder of the section, all results presented are using four virtual machines per physical node.

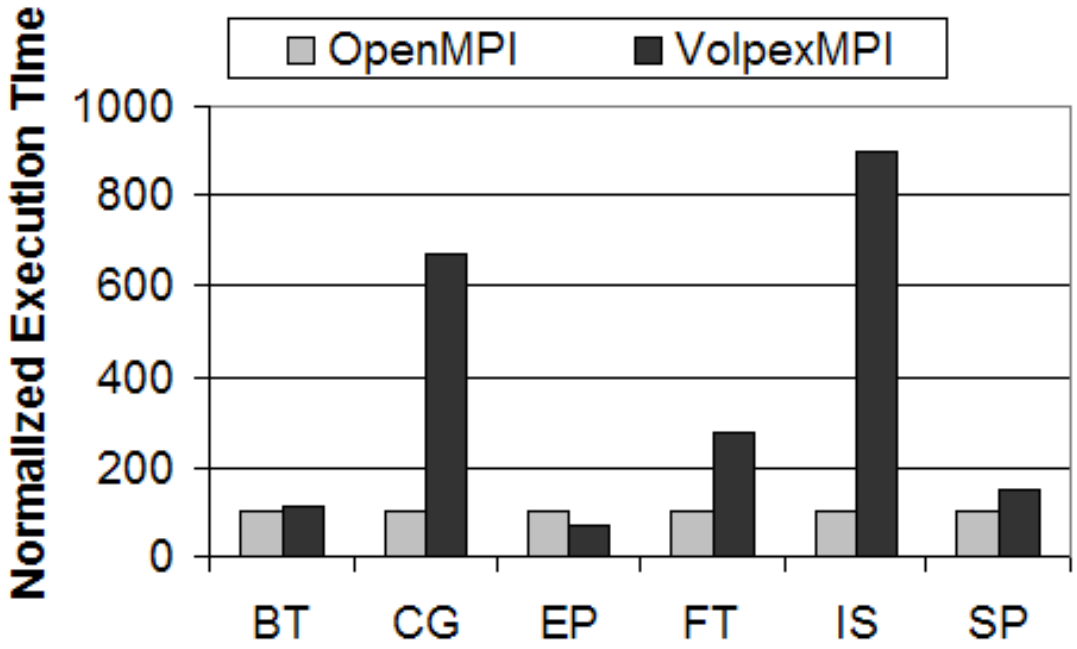

Figure 4.30: Comparison of OpenMPI to VolpexMPI for Class B NAS Parallel Benchmarks using 8 Processes on a virtualized cluster.

Figure 4.30 and Figure 4.31 show results for runs of 8 processes and 16 processes, respectively, utilizing the Class B data sets for six of the NAS Parallel Benchmarks as compared to the results obtained on the dedicated cluster. All times are noted as normalized execution times with a reference time of 100 for Open MPI on the dedicated cluster. The results overall confirm the observation from the previous section, namely that a minimal overhead for BT, SP, and EP and a more accented overhead for CG, FT and IS.

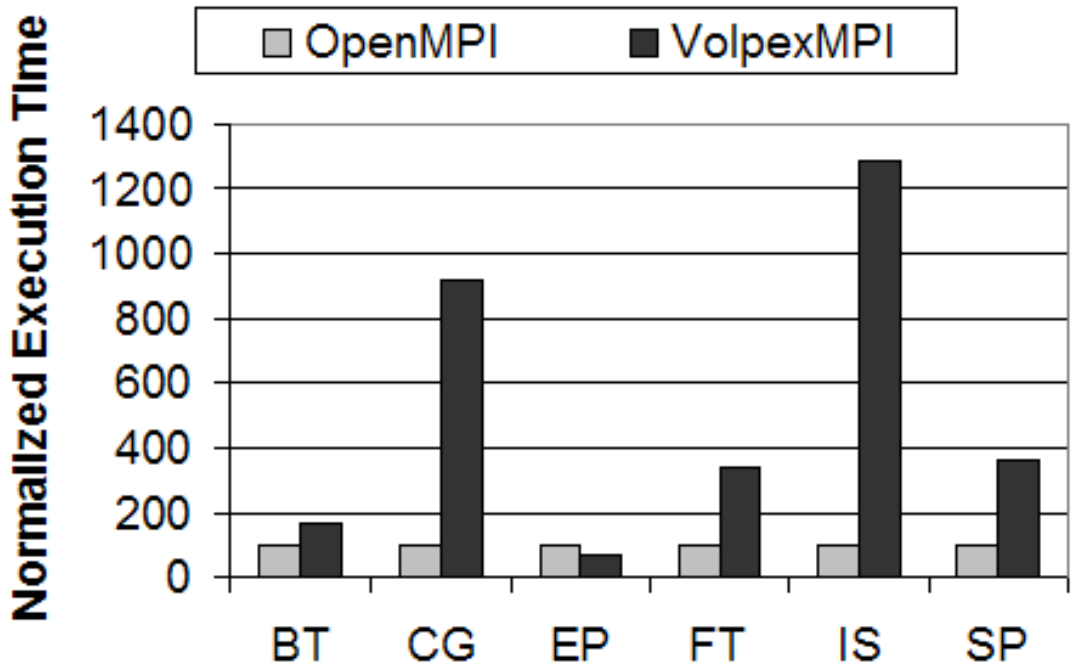

Figure 4.31: Comparison of OpenMPI to VolpexMPI for Class B NAS Parallel Benchmarks using 16 Processes on a virtualized cluster.

Simimlarly, executing an application with multiple copies of each MPI process on the virtualized cluster is shown in Figure 4.32 with no (same as single) redundancy (x1), double redundancy (x2) and triple redundancy (x3). The results indicate that, for the virtualized cluster, the overhead due to redundant execution is again minimal if no failure occurs, i.e. executing multiple copies of each MPI processes does not impose a significant performance penalty in the VolpexMPI *pull* model. The benchmarks that show again some sensitivity to replication are CG and SP.

Finally, we document the performance impact of a slow process for the NAS Parallel Benchmarks when using VolpexMPI. For this, we created two virtual nodes with one having a slow CPU and the other having a slow network connection emulating a volunteer process environment. The capabilities of Vmware allowed for one virtual machine to

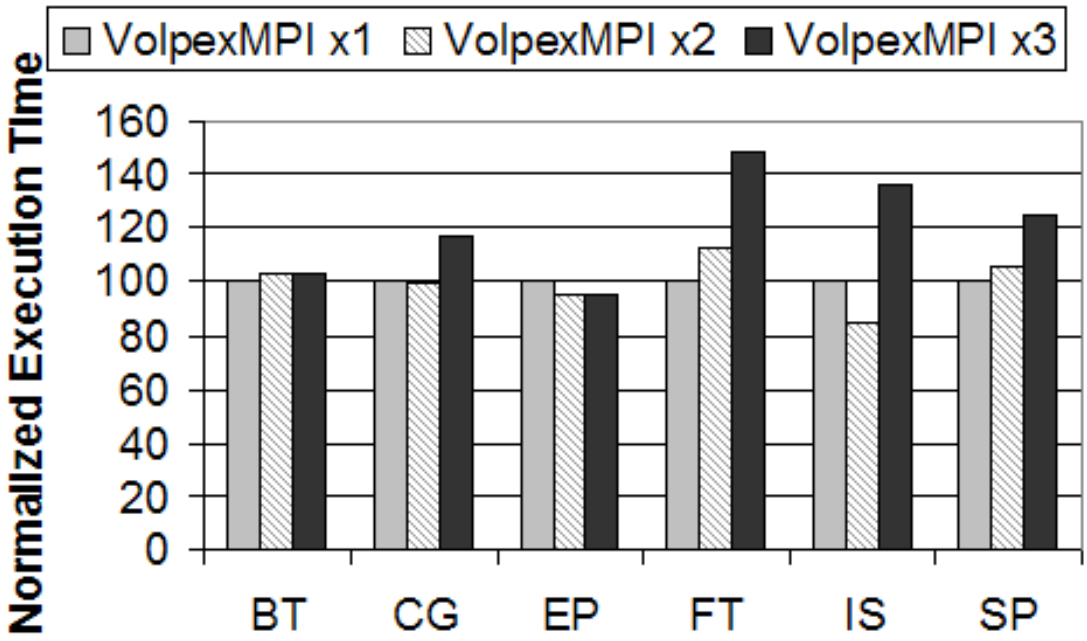

Figure 4.32: Comparison of VolpexMPI execution times for 8 MPI processes with varying degree of replication on a virtualized cluster.

be specified with a slow network connection at 3 Mb/Sec to mimic a volunteer desktop computer with DSL quality connections and another virtual machine to be specified with a slower CPU at 1Ghz to be equivalent to a low-end store purchased computer. All processes communicating with the slow process will thus have to either wait or repost all pending communication operations to the only remaining replica of process 1.

The results shown in Figure 4.33 show virtually no overhead in the scenario outlined above compared to the fault-free execution of the same benchmark using double redundancy. Detecting the process failure is, as of today, based on a timeout mechanism or the break-down of a TCP connection. However, since the correct result of the simulation is available as soon as any replica of each process finishes the execution, these overheads might not necessarily show up in the final result. Table 4.3 shows the actual run times for

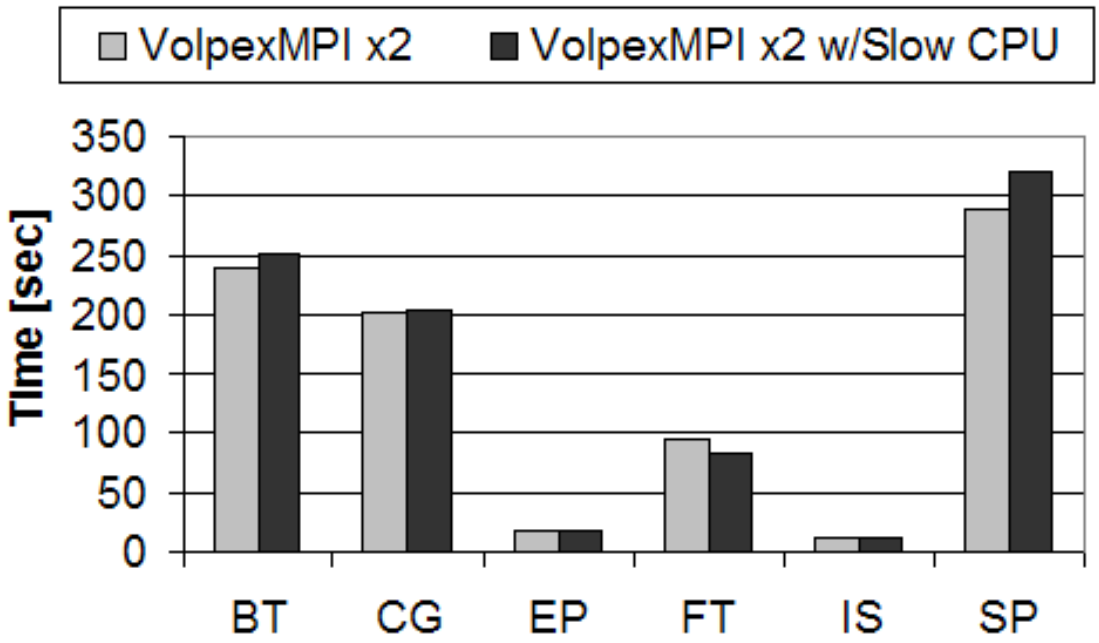

Figure 4.33: Comparison of VolpexMPI execution times for 8 MPI processes in case of a slow CPU.

the double redundancy runs with a slow CPU. Notice the last two columns where the run times for the ”B” redundancy level are generally 3 times greater than the ”A” redundancy level. Slowing down the CPU for node ”1,B” causes all ”B” group nodes to slow rather than request messages from the faster ”A” group because there is no capability to detect slowness in VolpexMPI as of yet.

The results shown in Figure 4.34 also show virtually no overhead in the scenario outlined above compared to the fault-free execution of the same benchmark using double redundancy. Here again, the correct result of the simulation is available as soon as any replica of each process finishes the execution; therfore, these overheads might not necessarily show up in the final result. Table 4.4 shows the actual run times for the double redundancy runs with a slow network connection. The last two columns show the run times for

Table 4.3: Virtualized Cluster Execution Times for 8 Nodes Class B with a Slow CPU

Class B 8 Nodes	Open MPI	VolpexMPI 8x2	VolpexMPI Slow CPU, A	VolpexMPI Slow CPU, B
BT	201.59	240.38	251.37	475.94
CG	30.30	202.23	204.15	401.65
EP	26.60	16.92	17.22	49.76
FT	29.63	93.59	82.01	150.25
IS	1.46	11.16	11.89	19.70
SP	179.06	289.14	321.02	517.12

the ”B” redundancy level are generally 5 to 10 times greater than the ”A” redundancy level. These results show that network latency has a larger effect on application progress than a slow CPU on performance for the Class B NAS Parallel Benchmarks. Again, VolpexMPI currently has no capability to detect process slowness.

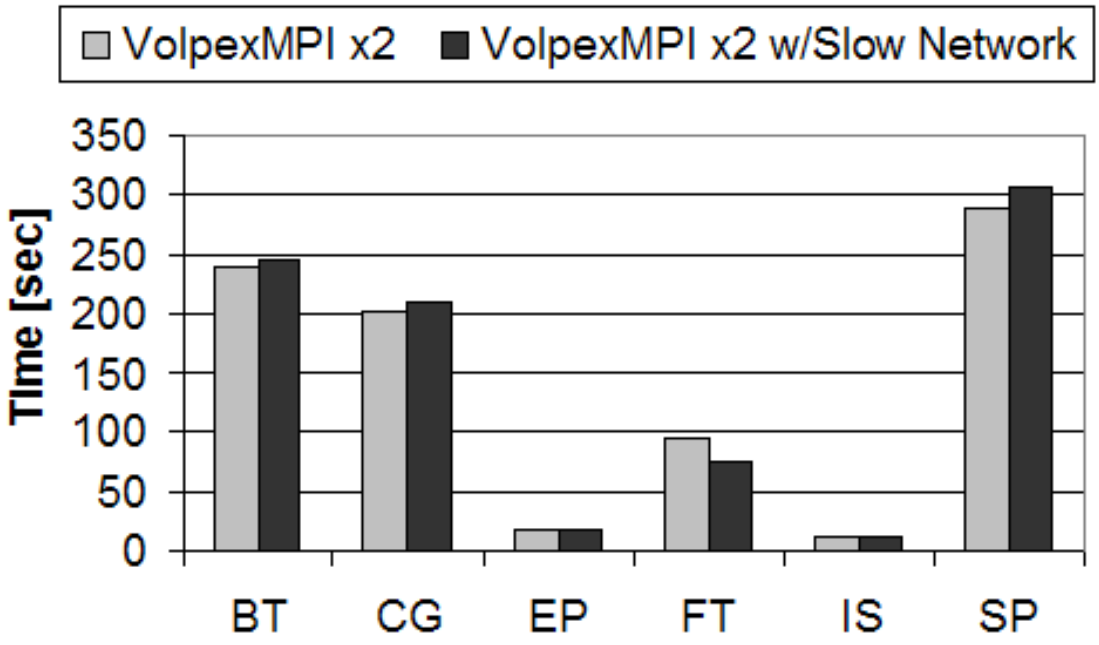

Figure 4.34: Comparison of VolpexMPI execution times for 8 MPI processes in case of a slow network connection.

Table 4.4: Virtualized Cluster Execution Times for 8 Nodes Class B with a Slow Network

Class B 8 Nodes	Open MPI	VolpexMPI 8x2	VolpexMPI Slow Network, A	VolpexMPI Slow Network, B
BT	201.59	240.38	245.84	1778.90
CG	30.30	202.23	211.04	1648.05
EP	26.60	16.92	17.05	17.23
FT	29.63	93.59	75.07	3349.51
IS	1.46	11.16	10.99	419.38
SP	179.06	289.14	307.21	3087.69

4.3.3 Performance Results in Volunteer Computing Environments

In this subsection we present results obtained on a pool of Condor machines at the University of Houston. The pool has access to dedicated and scavenged cycles from servers as well as from selected desktop computers. At this time, 6 pools form a grid through a feature known as ”flocking”. A user with local access to any of the individual pools can submit jobs to the entire grid. The servers in the Condor pool have AMD Quad Core Opteron 2.2GHz, 8GB RAM and Gigabit Ethernet interconnects running Red Hat Enterprise Linux 5.2.

Similarly to the previous section, we executed the NAS Parallel Benchmarks for 8 processes and compare the data to the dedicated cluster results described in section 4.3.1. As the results in Figure 4.35 indicate, VolpexMPI performed overall 26% to 58% slower on the Condor pool than the reference number obtained on the dedicated cluster. The most probable explanation is that some of these test runs utilized desktop class computers. This is a good representation of the volunteer environment and therefore a good representation of the varied performance projected for VolpexMPI. Furthermore, the results also indicate the fact that the connectivity between the nodes is not necessarily restricted to a single subnet/switch, but does very likely include nodes from different subnets, resulting in slightly lower communication perfomance.

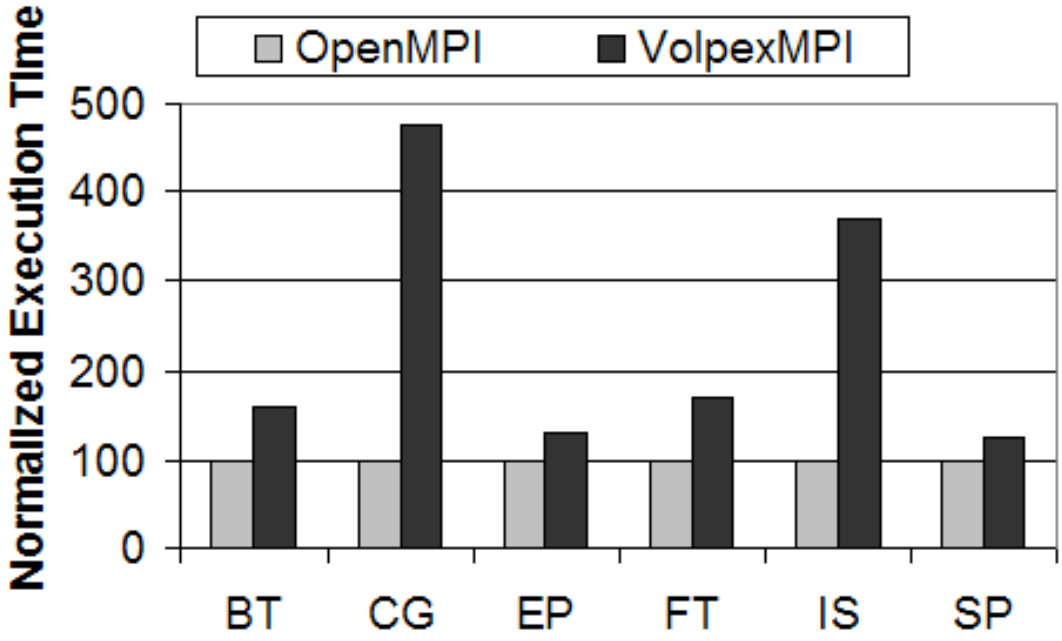

Figure 4.35: Comparison of OpenMPI to VolpexMPI for Class B NAS Parallel Benchmarks using 8 Processes on a Condor pool.

Executing the 8 process NAS Parallel Benchmarks running with no (same as single) redundancy (x1), double redundancy (x2), and triple redundancy (x3) indicate that the overhead due to redundant execution is again minimal if no failure occurs for single and double redundancy. The benchmarks do show however a significant sensitivity to triple redundancy runs. The reason probably is that with increasing number of resources requested within the Condor pool, the probability that the resources allocated are “spread out” across multiple networks and locations increases. Since the VolpexMPI startup mechanism on Condor cannot enforce a particular order of processes, the probability that processes within the same group are also distributed is fairly large, contributing towards the results observed in Figure 4.36. This result emphasizes the importance of the resource selection for the overall success of a VolpexMPI run on real volunteer computing environments.

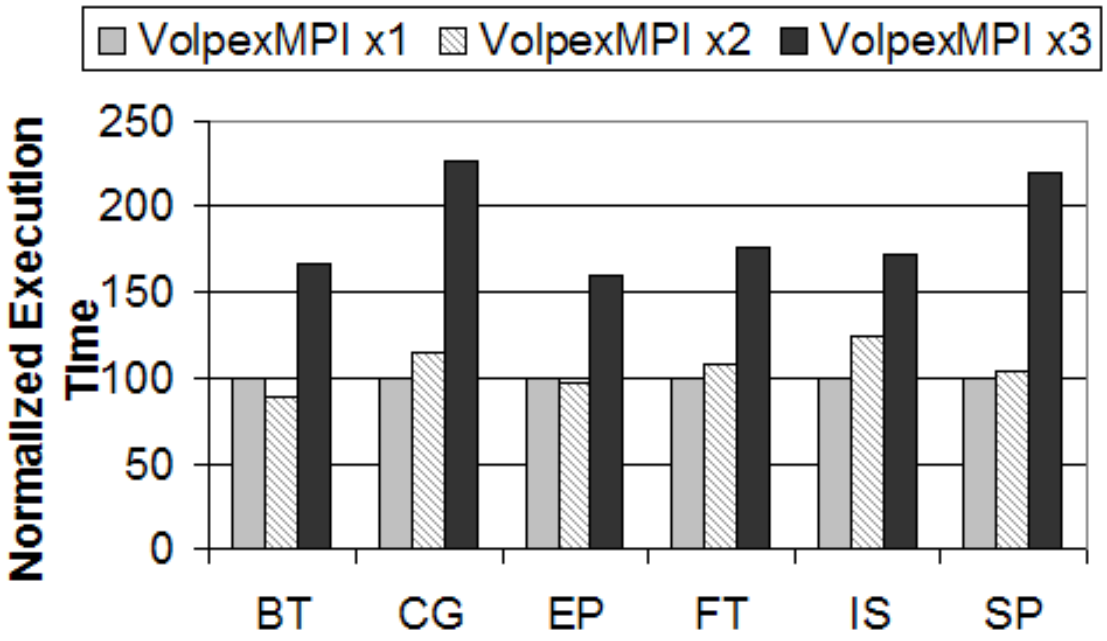

Figure 4.36: Comparison of VolpexMPI execution times for 8 MPI processes with varying degree of replication on a Condor pool.

4.3.4 Summary of Results

Finally, we document a combined view of the performance of VolpexMPI for all 3 cluster environments as compared to Open MPI runs on the dedicated cluster. Figure 4.37 shows BT, EP and SP because these benchmarks more closely represent the pattern of communication for a small number of processes having a favorable communication to computation ratio. EP performance on the virtualized cluster is best explained by the fact that the few communications occurring are between virtual nodes over a virtual network connection within a single physical machine. Figure 4.38 and Table 4.5 depict all six of the NAS benchmarks tested and are shown for completeness.

Overall, the results show consistently that the VolpexMPI library performs well under a variety of volunteer conditions as compared to dedicated cluster runs. We first tested

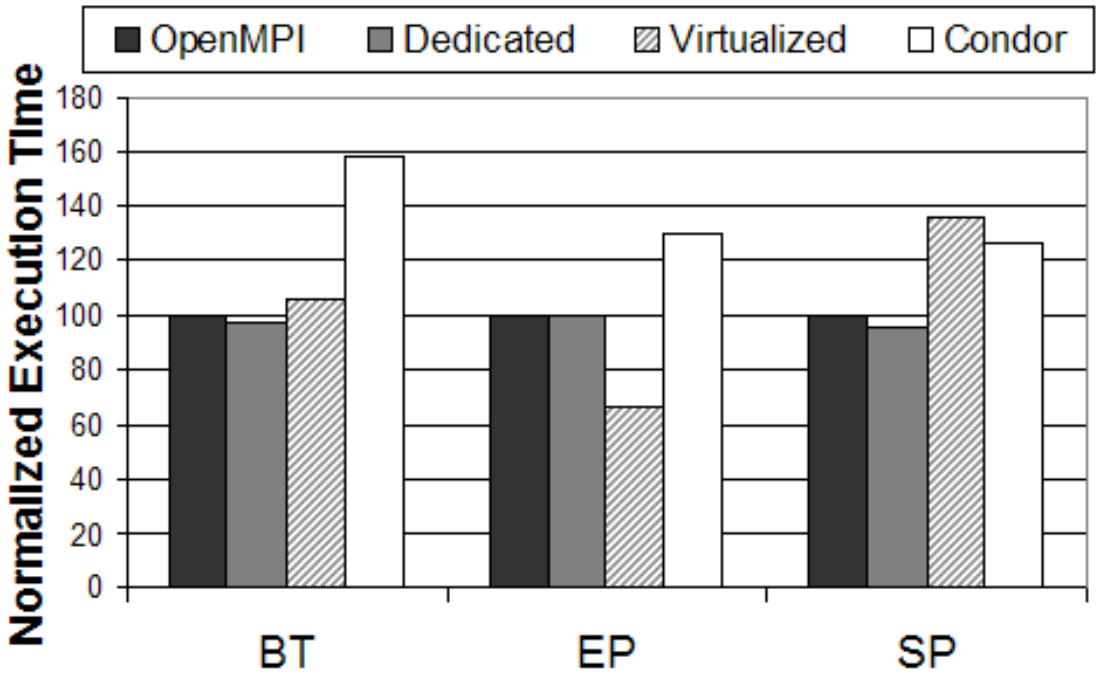

Figure 4.37: Comparison of OpenMPI to VolpexMPI for Class B NAS Parallel Benchmarks using 8 Processes across all test environments for BT, EP and SP.

Table 4.5: Execution Times for 8 Nodes Class B across all Test Environments

Class B 8 Nodes	Open MPI	Dedicated	Virtualized	Condor
BT	201.59	195.72	232.73	319.41
CG	30.30	68.53	203.16	143.99
EP	26.60	26.66	17.81	34.53
FT	29.63	60.73	82.99	50.29
IS	1.46	9.27	13.16	5.38
SP	179.06	170.77	274.47	226.18

VolpexMPI on a dedicated cluster with performance equivalent to Open MPI runs on the same cluster. We next tested VolpexMPI on a virtualized cluster to show that the library can perform well on the latest technology, and the system allowed us to setup and test both slow CPU and slow network connection scenarios. Finally, we tested on a Condor pool

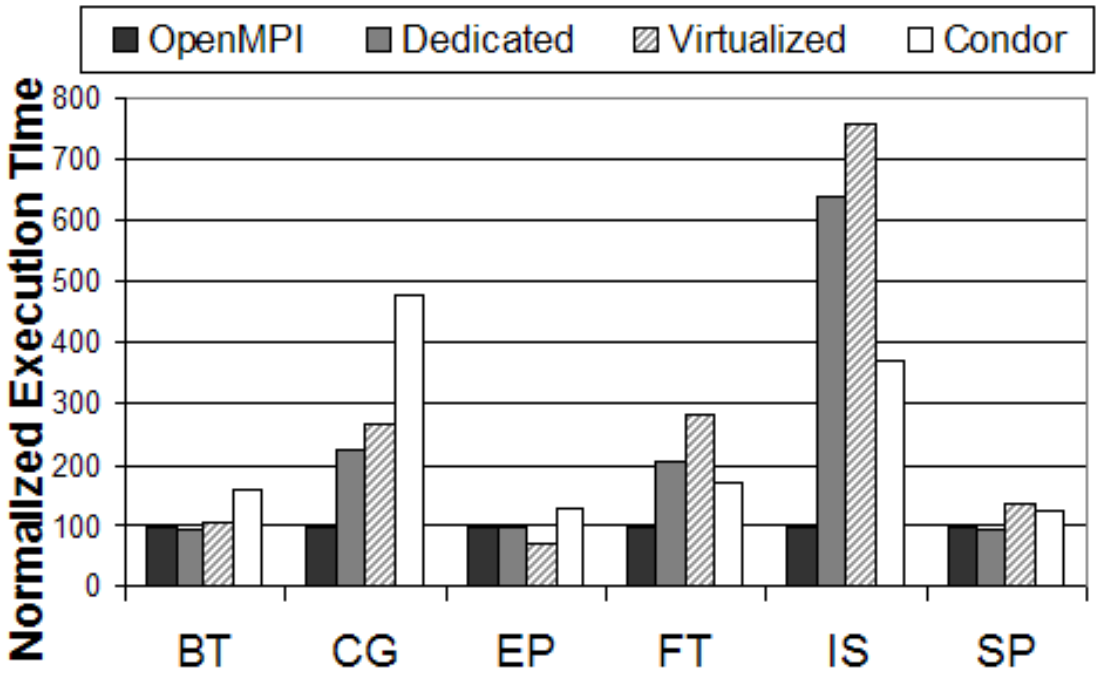

Figure 4.38: Comparison of OpenMPI to VolpexMPI for Class B NAS Parallel Benchmarks using 8 Processes across all test environments for all benchmarks.

because it represents volunteer computing in that the nodes selected for any particular run could have very different CPU and network specifications.

The most important conclusion that can, however, be drawn based on the results of this section is that the overhead observed within all three classes of experiments (dedicated, virtualized, volunteer) is manageable. While there is clearly an overhead due to the fact that the environment is less homogeneous and offers typically a lower communication performance compared to dedicated clusters, for selected applications with favorable characteristics volunteer computing, using VolpexMPI offers an easy and cost-effective alternative to dedicated clusters.

Chapter 5

Architecture for Heterogeneous Environments

In the previous chapter, we introduced VolpexMPI, a software framework that allows failsafe execution of communicating parallel MPI programs on volatile volunteer nodes through the coordinated use of redundant nodes. However, the approach presented has multiple shortcomings. First, the implementation of that library is based on C and relies on non-blocking sockets. While this is might be the most performant approach, porting this implementation to operating systems other than Linux – most notably Windows – is a non-trivial task [16, 21] and requires significant efforts. Second, monitoring the application state turned out to be burdensome and did not meet the user-friendliness of other volunteer computing projects such as BOINC.

In this chapter, we explore an alternative approach, called VolpexPyMPI and shown in Figure 5.1, to provide communication in volunteer computing environments. Volpex-PyMPI, if following many of the design principles of VolpexMPI, is based on Python, a

high-level programming language. There are a number of high-level interpreted programming languages that could have been used including Python, Java [50], and Perl. VolpexPyMPI utilizes many of the features of Python such as a large set of library routines, built-in support for heterogeneous resources, including seamless handling of both Linux and Windows operating systems, and the ease of quickly producing and testing various communication services such as TCP/IP, HTTP over TCP/IP and XML-RPC over HTTP. The compilers used with this architecture are the GNU C and FORTRAN compilers for Linux and the Minimalist GNU Compiler for Windows (MinGW)

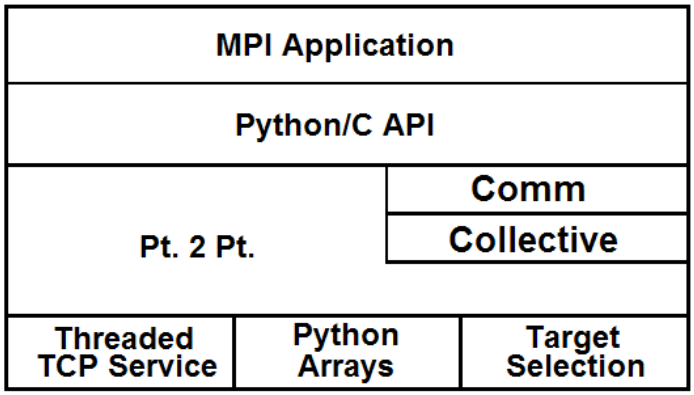

Figure 5.1: VolpexPyMPI Architecture

VolpexPyMPI consists of two major building blocks. The first is a node selection framework to monitor a set of volunteer computers that could be tasked with executing a standard MPI program. The second building block is the actual MPI library. The MPI library utilizes the Python/C Application Programming Interface (API) to interface the VolpexPyMPI library to execute unaltered MPI programs written in C or FORTRAN. The resulting library system allowed for testing on a selection of Linux and Windows nodes in the framework which could execute unaltered user MPI programs. The rest of this chapter will describe a node selection framework, and the slight differences for some of the MPI functions in Python. Finally we test this architecture on a dedicated cluster for stable and reproducible results.

5.1 Node Selection Framework and Data Structures

The main goal of the node selection framework is to monitor a set of volunteer computers that could be tasked with executing a parallel application. The VolpexPyMPI node selection framework was built as an asynchronous XML-RPC webserver (henchforth called Volpex server) and XML-RPC clients (henthforth called Volpex clients). The Volpex server and client software were written in Python and can run on Linux and Windows systems.

Among the main functions of the Volpex server is to maintain a GlobalMap, which is a configuration file in tabular representation showing the status of all Volpex clients. The GlobalMap is implemented using Python's SQLite3 module, a lightweight disk-based database that does not require a separate server process. The GlobalMap status includes the last check-in time (reported as DDD/HH:MM:SS), the last status message from the server to the client, the last status message from the client to the server, and the type of operating system running on the client node. The Volpex server status messages include abort, start, inactive, and suspect, and the Volpex client status messages include done, active, ready, and avail. Upon start of a MPI application, the Volpex server forwards the GlobalMap to the Volpex clients, which store it as an in-memory data structure.

The Volpex server utilizes port 8080 and is run on a standard Linux server also running the Apache webserver which utilizes port 80 for user interaction in MPI run setup. Performance monitoring of the Volpex server and clients shows a relatively low CPU utilization of about 0.1% during quiescent timeframes. CPU and memory utilization during MPI runs will, of course, increase dramatically. The Volpex server interacts with the Volpex clients over the specified port and through a set of public functions described later. The user interface controls available to the end user are via CGI scripts used to interact with the Volpex server as illustrated in Figure 5.2. The webpage allows the user to set the number of MPI processes and the level of redundancy required (currently limited to single, double, or triple

redundancy) and to upload the user's MPI program to the server for compilation. For the purposes of initial testing an additional drop-down list interface allows easy selection, upload and compilation of the NAS Parallel Benchmarks. The user interface also has other buttons such as Execute, Print Event Log, and Reset Mapping for controlling the test case execution.

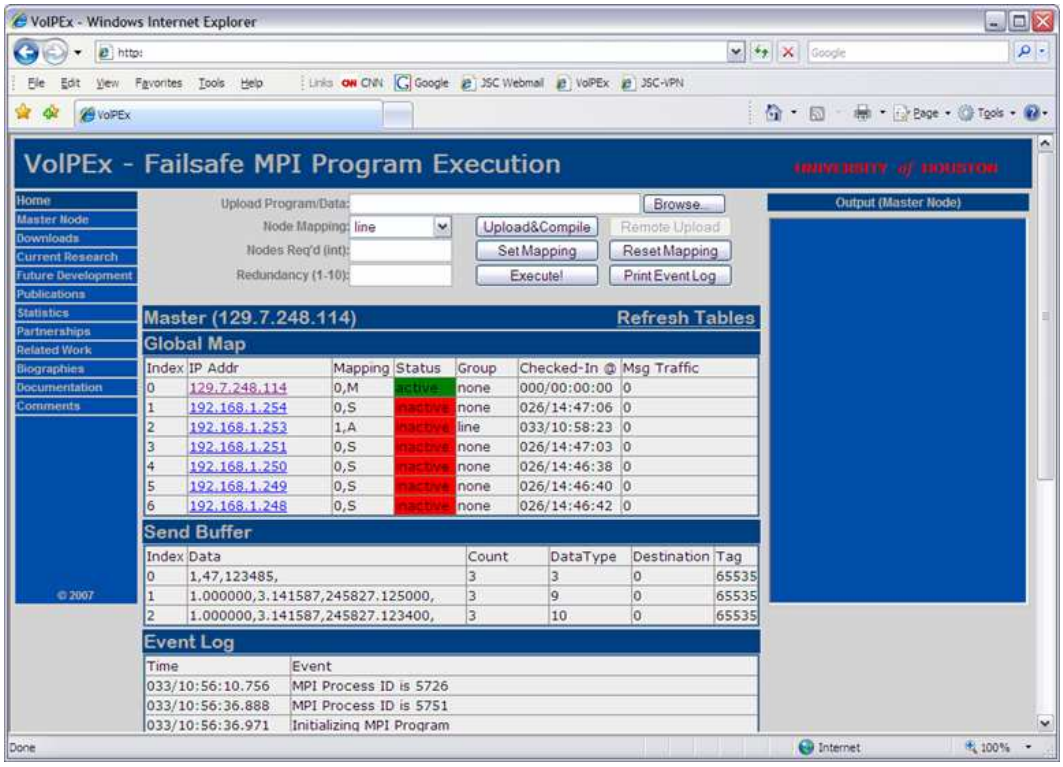

Figure 5.2: Node Selection Framework Web User Interface

The user interface also shows the *SendBuffer* and the *Event Log* on a per process basis. The Event Log offers the ability to track the progress of each process individually. Note that the event log is typically turned off for actual performance measurements. Each node in the GlobalMap that is involved in the execution of a particular MPI run can post the contents of its SendBuffer and Event Log to the Volpex server after the run for analysis.

Like VolpexMPI, VolpexPyMPI must provide distributed sender-based logging. The

SendBuffer provides the functionality to store and retrieve a MPI messages. An important question is whether the message buffers on the sender processes must be maintained for the duration of execution or whether they can be cleared at some point. From the logical perspective, a message buffer can never be cleared due to the fact that, even if all replicas of a particular rank have received a given message, all of them might fail to finish the execution. Thus, a new replica of that process might have to be started, which would have to retrieve all messages. In the C version of the library, we deploy a circular buffer where the oldest log entry is removed when the buffer is full. There are two versions of the Sendbuffer management available in VolpexPyMPI. The first version utilizes, once again, the SQLite3 module and does not limit, as of today, the size of the SendBuffer. The second version uses a Python array treated as a circular buffer. Note that the long-term goal is to coordinate the size of the SendBuffer with checkpoints of individual processes, which will allow guaranteed restarts with a bounded buffer size.

5.2 MPI Functions

For the actual data transfer between the MPI processes, VolpexPyMPI has once again two options. The first utilizes the XML-RPC services for communication between the processes. The second version uses a standard Python Threaded TCP SocketServer class for the direct node-to-node communication and uses the XML-RPC service only for the communication between to the Volpex server and clients. In actuality, the SocketServer Python functions are embedded in the VolpexMPI source files and are compiled into a single library that supports linking to both C and FORTRAN user MPI programs in Linux and Windows environments.

The Python-based VolpexMPI Library comprises 20 of the standard MPI functions. The

C source file in the VolpexMPI Library contains only the Python/C APIs for each MPI function to identify with either C or FORTRAN user programs. These sections use #pragma weak declarations for the various allowed conventions of function naming in FORTRAN. This implementation allows the user to program using the standard MPI library function names which call the Python versions of the MPI functions. Table 4.1 in the previous chapter lists all of the MPI functions chosen for this implementation which were based on their usage in the NAS Parallel Benchmarks.

The previous chapter described all the MPI functions available in the VolpexMPI Library, so we will only describe the four most basic functions that have unique Python functionality. These functions are MPI_Init, MPI_Finalize, MPI_Send, and MPI_Recv. Similar to the C implementation, all of the Python MPI functions are built upon the send and receive functions.

MPI_Init - MPI_Init initializes the VolpexPyMPI execution environment. Among the items initialized are a request list for non-blocking messages, structures for tracking the logical time stamps, reading the configuration file into a local data structure, initializing the SendBuffer, and resolving the private IP address and hostname of the local processor. Figure 5.3 shows the pseudo code for both the C and Python functions. `Py_Initialize` is a Python Library call to initialize the Python run-time interpreter. `Py_Init` is a VolpexPyMPI internal function, which among others, starts the Threaded TCP service. The call to CreateGM reads the GlobalMap by appending socket addresses to the active in-memory data structure. Note, that calls for exception handling are not shown in this pseudo code.

MPI_Finalize - MPI_Finalize terminates the VolpexPyMPI execution environment and frees data structures. MPI_Finalize first waits for a predefined amount of time (e.g. 5 seconds). This additional waiting time is a simple method that allows other processes to still retrieve messages and finish gracefully for benchmarking purposes, but it is not required for production runs. Using `Py_Finalize`, a Python library call, VolpexPyMPI

MPI_Init

```
int MPI_Init( int *argc, char ***argv )
{
    PyObject *pName, *pModule, *pDict, *pFunc;
    PyObject *pArgs, *pValue;
    …
    Py_Initialize();
    pName = PyString_FromString("Volpex");
    PyObject_GetAttrString(pModule, "PyInit");
    …
    return 0;
}
```

PyInit

```
def PyInit():
    HOST, PORT = '', 39000
    server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)
    CreateGM()
    for i in range(0,len(GM)):
        socketaddr.append(GM[i][12])
    …
    return 0
```

Figure 5.3: Pseudo code for MPI_Init Function with the call to PyInit

shuts down the Python run-time interpreter which will finally de-allocate the data structures such as the SendBuffer.

MPI_Finalize

```
int MPI_Finalize()
{
    PyObject *pModule,*pFunc,*pArgs,*pReturn;

    pModule = PyImport_ImportModule("Volpex");
    pFunc = PyObject_GetAttrString(pModule, "PyFinalize");
    PyObject_CallObject(pFunc, NULL);
    Py_DECREF(pFunc);
    Py_DECREF(pModule);
    Py_Finalize();
}
```

PyFinalize

```
def PyFinalize():
    time.sleep(5)
    print 'Finalizing MPI Program'
    return 0
```

Figure 5.4: Pseudo code for MPI_Finalize Function with the call to PyFinalize

MPI_Send - The implementation of MPI_Send (and MPI_Isend) in VolpexPyMPI is fairly simple, since it only copies data to the local SendBuffer after the size of the MPI datatype used is determined. The library also determines the logical time stamp of each message and stores it, along with the other communication parameters, in order to uniquely identify a message upon request of a receiver process.

MPI_Send

```
int  MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
           MPI_Comm comm)
{
     pModule = PyImport_ImportModule("Volpex");
     pFunc = PyObject_GetAttrString(pModule, "PySend");
     if (datatype == MPI_INT || datatype == MPI_INTEGER){
        sprintf (httpsubString, "%d,", iPtr[i]);
        strcat (httpString, httpsubString);
     }
     if (datatype == MPI_FLOAT || datatype == MPI_REAL){
     …
     Py_DECREF(pModule);
}
```

PySend

```
def PySend (bufferdata, length, mytarget, datatag, commgroup):
    reuse = 0
    sendrequest = send:"+str(length)+":"+str(mytarget)…
    reuse = ProcessSBRequest(sendrequest, bufferdata)
    …
    return reuse
```

Figure 5.5: Pseudo code for MPI_Send Function with the call to PySend

MPI_Recv - Figure 5.6 shows the pseudo-code for the MPI_Recv function. Because it highlights how the receiver-based direct communication works, we would like to discuss it in some more details. The `PyRecv` function determines the logical timestamp of the message based on the tuple of (sender rank, receiver rank, communicator, and tag). Next the `TargetSelect` function is called to determine which target nodes to call and in which order to call them in case of failures. For example, if a process with the rank 1 wants to receive a message from the process with rank 2 and the run was utilizing triple redundancy, the `TargetSelect` function would read the GlobalMap for the particular communicator to determine the IP address and port number of nodes ”2,A”, ”2,B” and

”2,C”. The MPI function would then proceed in a round-robin fashion attempting to receive the message. This usually starts with an attempt to contact the rank 2 within the same redundancy level as the requesting rank 1 node. If the node is unreachable, the MPI receive function proceeds to the next higher redundancy level or loops around to the lowest rank, if necessary. If the first receive call is successful, the MPI library proceeds normally without ever attempting to contact the redundant nodes. If attempts are unsuccessful, the target is marked as unreachable within the GlobalMap and there are no further attempts to contact the node for data. Once a target is chosen, a TCP call is made to the target MPI process’ TCP service via the VolpexSBRequest. Note that the non-blocking function `MPI_Irecv` capitalizes on the use of Threads and allows a request for data to proceed in the background until the associated MPI_Wait request is encountered.

MPI_Recv

```
int  MPI_Recv (void *buf, int count, MPI_Datatype datatype, int source,
          int tag, MPI_Comm comm, MPI_Status *status)
{
     pModule = PyImport_ImportModule("Volpex");
     pFunc = PyObject_GetAttrString(pModule, "PyRecv");
     data = PyString_AsString (pReturn);
     if (datatype == MPI_INT || datatype == MPI_INTEGER){
        …
     Py_DECREF(pModule);
…
}
```

PyRecv

```
def PyRecv (length, target, datatag, commgroup):
    reuse = CheckTagReuse (target, datatag, commgroup)
    myrank = PyComm_rank (commgroup)
    recvrequest = 'recv:'+str(length)+":"+str(myrank)+":"+str(datatag)…
    …
    targets = TargetSelect (target,commgroup)
    …
    webpage = VolpexSBRequest (recvrequest, 'none')
    return webpage
```

Figure 5.6: Pseudo code for MPI_Recv Function with the call to PyRecv

Table 5.1: VolpexPyMPI Execution Times

Class S 4 Nodes	VopexMPI 4x1	VolpexPyMPI v1	VolpexPyMPI v2
BT	3.04	134.14	12.09
CG	4.31	242.71	27.53
EP	0.87	5.25	1.24
IS	0.29	7.52	1.55
SP	4.89	221.75	19.88

5.3 Experiments and Results

In this section, the performance evaluation focuses on the MPI Library. For this, we are running some benchmarks on a dedicated cluster in order to have a controlled environment and understand the performance implication of the Python implementation. The dedicated cluster utilizes 29 compute nodes, 24 of them having a 2.2 GHz dual core AMD Opteron processor, and 5 nodes having two 2.2GHz quad-core AMD Opteron processors. Each node has 1 GB main memory per core, and a network connected by 4xInfiniBand, and a 48 port Linksys GE switch. We utilized Python 2.4 across the cluster. For evaluation, we utilize the Gigabit Ethernet network interconnect of the cluster to compare the VolpexPyMPI run times to VolpexMPI.

The BT, CG, EP, IS and SP benchmarks from the NAS Parallel Benchmarks suite are executed for 4 process counts and a data class set size of S. For each experiment, the run times were captured as established and reported in the NPB with the normal `MPI_Wtime` function calls for start and stop times. Table 5.1 gives the actual run times in seconds for the five NAS Parallel Benchmarks. We have excluded from our experiments three of the NAS Benchmarks. We exclude FT due to a software bug and LU and MG due to their use of `MPI_ANY_SOURCE` which is not currently supported in Volpex.

Figure 5.7 shows the first test results for runs of 4 processes utilizing the Class S data

sets for the five NPBs of interest. These reference executions did not employ redundancy. The run times for the C version of VolpexMPI are shown for comparison in the bar graph. All times are noted as normalized execution times with a reference time of 100 for VolpexMPI. Version 1 contains the SQLite3 version of the SendBuffer management and uses the XML-RPC service for node-to-node communication.

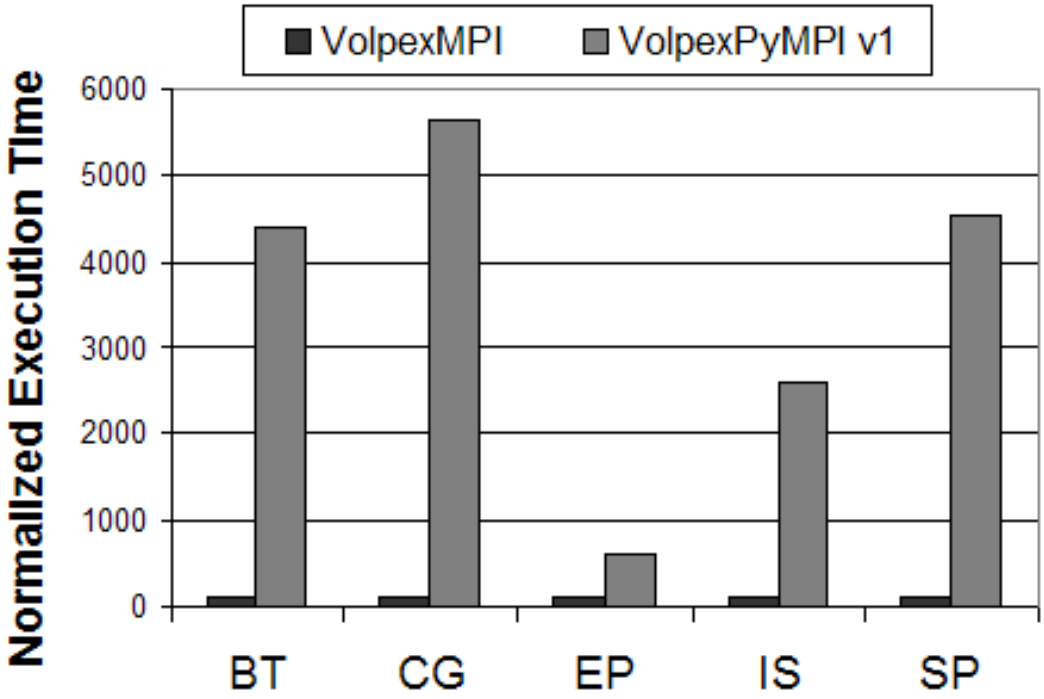

Figure 5.7: Performance results of Version 1 of VolpexPyMPI compared to the results of the C version VolpexMPI.

Version 2, shown in Figure 5.8, uses a Python array for SendBuffer management and the threaded TCP service for data transfer. The results indicate that version 1 of VolpexPyMPI run times are between 5 and 55 times greater than the according C version of the library. VolpexPyMPI version 2 greatly improves the execution time of these benchmarks, reducing the overhead to a more manageable factor of 1.5 in the best case and 6.5 in the worst case compared to the C-only counterpart. Therefore, for the rest of the analysis we are focusing

entirely on the 2nd version of VolpexPyMPI.

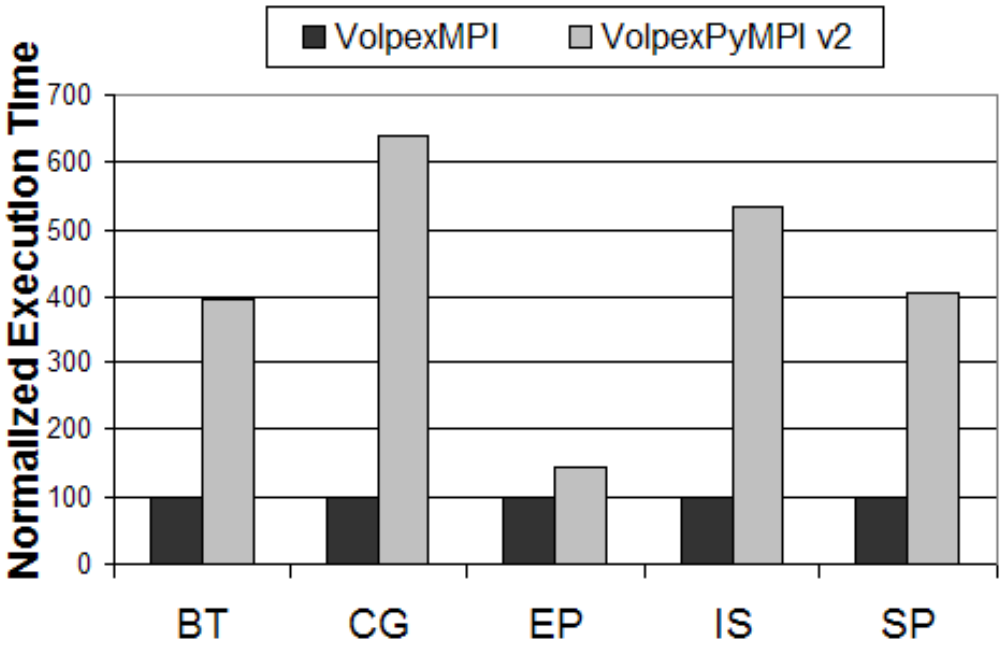

Figure 5.8: Performance results of Version 2 of VolpexPyMPI compared to the results of the C version VolpexMPI.

Next, we document the effect of executing an application with multiple copies of each MPI process. The left part of Figure 5.9 shows the normalized execution times of VolpexPyMPI for the very same test cases running with no (same as single) redundancy (x1), double redundancy (x2), and triple redundancy (x3). The results indicate that, for most benchmarks, the overhead due to redundant execution is minimal if no failure occurs, i.e. executing multiple copies of each MPI processes does not impose a significant performance penalty in the VolpexPyMPI scheme/model.

Finally, we document the performance impact of a process failure for the NAS Parallel Benchmarks when using VolpexPyMPI. For this, we inserted into the source code of each benchmark some statements which terminate the execution of the second replica of rank 1 in `MPI_COMM_WORLD`, emulating a process failure. All processes communicating with

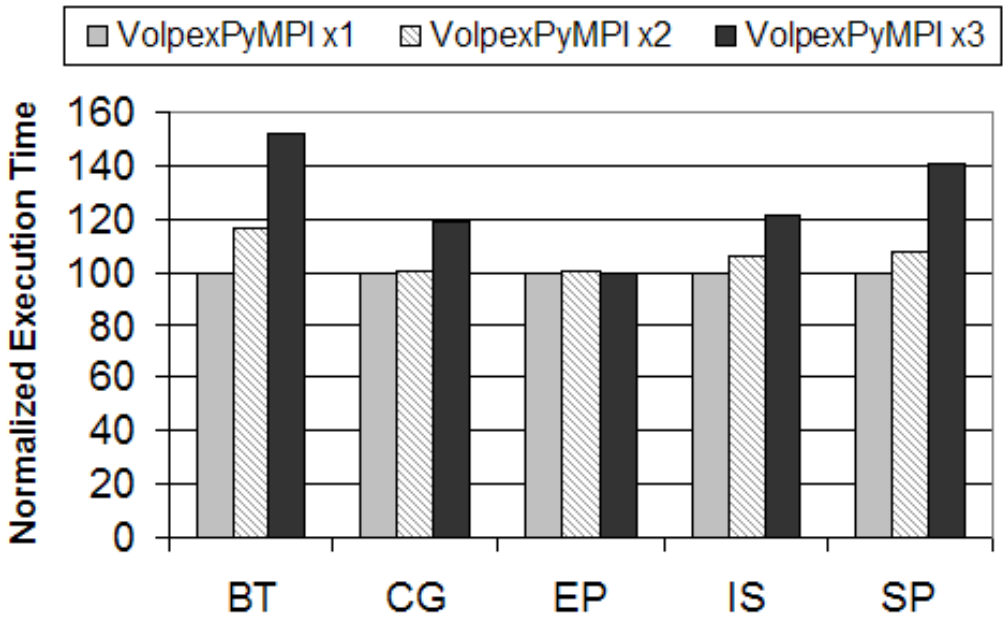

Figure 5.9: Performance results of VolpexPyMPI for single, double, and triple redundancy for 4 nodes.

the terminated process will thus have to repost all pending communication operations to the only remaining replica of process 1. This test case represents one of the worst case scenarios for VolpexPyMPI because the number of processes communicating with a single process doubles at runtime. Killing more than one process would actually relieve the remaining processes with rank 1 because the number of communication partners is reduced. The results shown in Figure 5.10 show virtually no overhead in the scenario outlined above compared to the fault-free execution of the same benchmark using double redundancy.

5.4 Python Discussion

This chapter explores a Python-based implementation of MPI for volunteer computing environments. We demonstrated that the fundamental approach is appropriate to provide

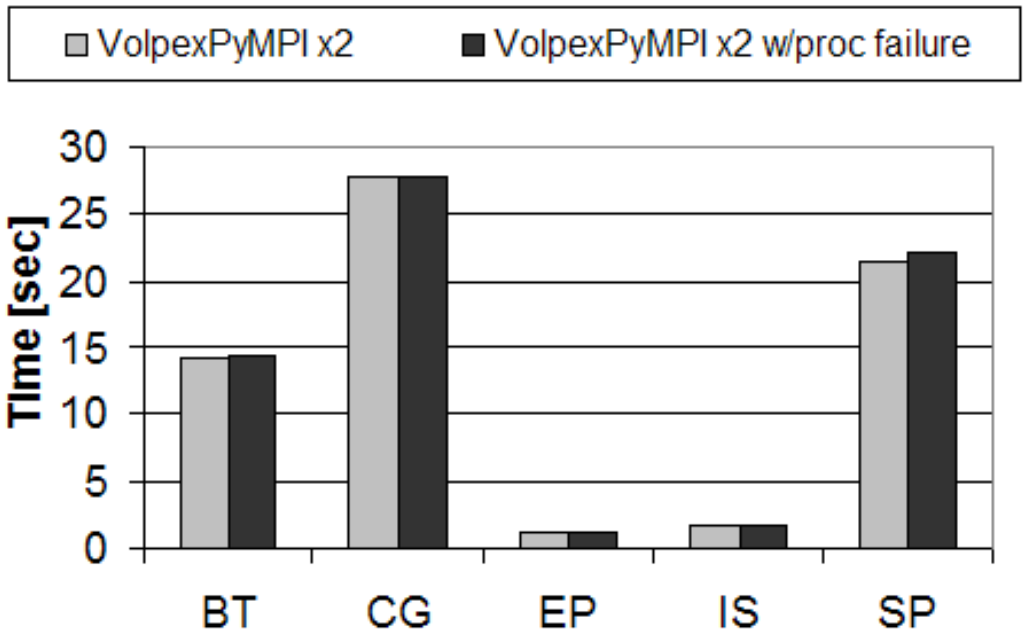

Figure 5.10: Performance results of VolpexPyMPI in case of a node failure.

fault-tolerance based on process replication using a *pull* model communication scheme combined with sender-based message logging. Our experiments demonstrate that unaltered MPI applications, such as given with the NAS Parallel Benchmarks, can be executed using VolpexPyMPI. In our experience, the fundamental benefits of using a "Python only" approach including components such as SQLite3 and the XML-RPC scheme are the increased portability of the software, ease of development of the MPI library, and ease of deployment of applications. The consequences of this approach are, however, a significant performance degradation compared to using a "legacy" programming language such as C. By combining both approaches, namely using Python for the overall management of processes and non-performance critical section, and using more traditional TCP services and in-memory data structures for performance critical aspects of the library, we managed to create, however, a hybrid implementation that combines the best of both worlds: portability

of python and a manageable performance overhead.

Chapter 6

Conclusion

This dissertation introduces a MPI library, VolpexMPI, that is designed for robust execution of parallel scientific applications on idle desktop computers. The key design goal is efficient execution with replicated processes. The VolpexMPI Library employs a receiver-based *pull* model for communication between the processes and a distributed, sender-based message logging scheme. Additionally, the design can handle nodes with varying compute, communication, and storage capacity as well as heterogeneity. Finally, to minimize impacts to the science community, VolpexMPI incorporates methods to ensure no alterations are required to compile and execute legacy parallel scientific applications.

6.1 Contributions

To summarize the contributions of this dissertation, we have designed, implemented, and tested a new MPI Library that provides distributed fault tolerance for volunteer computing environments. The highlights of the technical approach are as follows:

1. *Controlled redundancy:* A process can be initiated as two (or more) replicas. The execution model is designed such that the application progresses at the speed of the fastest replica of each process and is unaffected by the failure or slowdown of other replicas, as in Figure 1.2. (Replicas may also be formed by checkpoint-based restart of potentially failed or slow processes, but this aspect is not implemented yet).

2. *Receiver-based direct communication:* The communication framework supports direct node to node communication with a *pull* model; the sending processes buffer data objects locally and receiving processes contact one of the replicas of the sending process to get the data object.

3. *Distributed sender-based logging:* Messages sent are implicitly logged at the sender and are available for delivery to process instances that are lagging due to slow execution or recreation from a checkpoint.

We have implemented controlled redundancy, including designating redundancy level, node selection, and node termination into the VolpexMPI Library. Additionally, the VolpexMPI Library can execute 6 of the 8 NAS Parallel Bencharks on dedicated clusters, virtualized clusters, and Condor pools. We have addressed receiver-based direct communication by checking for aliveness of nodes prior to sending actual data, and we have tested to enable performance within acceptable percentages of Open MPI executing on a dedicated cluster. Distributed sender-based logging is implemented with a robust Send_Buffer management design that minimizes message searches and memory usage. We also addressed heterogeneity by implementing VolpexPyMPI, a Python-based approach. The Python implementations characterize the cost-to-benefit ratio of higher level communications protocols and databases as compared to lower level protocols and in-memory data structures.

We demonstrated, with both the C and Python implementations of the design, the necessity to focus on the right class of applications for VolpexMPI. Namely, those with a

favorable communication to computation ratio and a modest degree of communication. Benchmarks having those characteristics show only a minor overhead compared to a standard MPI library, such as Open MPI. More importantly, utilizing multiple replicas for each process does not impose a notable overhead for the majority of the benchmarks. Also, the NAS Parallel Benchmarks analyzed could successfully survive a process failure without suffering a major performance degradation. Hence, the objectives and design goals for VolpexMPI are satisfied.

6.2 Future Work

The ongoing work on VolpexMPI includes developments of target selection algorithms and execution in more test environments. The target selection algorithms will consider initial node selection, fastest replicas, and most appropriate replicas. Researchers are working on integrating checkpoint-restart with VolpexMPI to dynamically manage replication by recreating slow and failed replicas from healthy replicas. The plans include deployment and evaluation of VolpexMPI on a larger campus desktop computer grid using BOINC.

The research group is investigating several applications as candidates for execution on desktop computer clusters, and the group is building simulation tools to rapidly assess the suitability of an application for VolpexMPI. In particular, we are in active discussions with a research group at the University of Houston which develops the Replica Exchange Molecular Dynamics (REMD) application [51]. The memory and compute requirements of this application combined with its low, but important, communication requirements make the application ideally suited for VolpexMPI.

There are many capabilities that need to be added to VolpexMPI in order to provide a

MPI Library that is very stable and very transparent to the volunteer community. Some examples of upcoming capabilities are checkpoint-restart [58], tracking and prioritizing most viable volunteers, tracking and maximizing use of idle volunteer time, and adhering to user preferences for availability. Additionally, the scalability of VolpexMPI needs to be stress tested to address any design issues associated with very large volunteer computing environments. As shown in Figure 4.24, a centralized web server could be used in conjunction with the VolpexMPI startup module to attach to various cluster front end nodes as well as home desktop computers to initiate and execute MPI runs. Of course, the main problem to attack is communication between nodes on differing environments. Here, SSH tunneling may be the appropriate design solution.

Other future work includes adding more MPI functions and datatypes, characterizing a *push* model, and researching deployment methodologies. Finally, the testing must extend beyond the NAS Parallel Benchmarks to other test cases or actual applications that stress fault tolerance to slowness, as well as failure, in order to allow execution to proceed at the speed of the fastest replicas. These aspects will be addressed in upcoming research and experiments.

Bibliography

[1] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations. In *8th IEEE International Symposium on High Performance Distributed Computing*, 1999.

[2] Amazon webservices. Amazon Elastic Compute Cloud (Amazon EC2). http://www.amazon.com/gp/browse.html?node=201590011, 2008.

[3] D. Anderson. Boinc: A system for public-resource computing and storage. In *Fifth IEEE/ACM International Workshop on Grid Computing*, November 2004.

[4] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: An experiment in public-resource computing. *Communications of the ACM*, November 2002.

[5] D. Anderson and G. Fedak. The computation and storage potential of volunteer computing. In *Sixth IEEE International Symposium on Cluster Computing and the Grid*, May 2006.

[6] Rajanikanth Batchu, Jothi P. Neelamegam, Zhenqian Cui, Murali Beddhu, Anthony Skjellum, and Yoginder D. Mpi/ft tm : Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing. In *In Proceedings of the 1st IEEE International Symposium of Cluster Computing and the Grid*, 2001.

[7] C. Boeres, A. Nascimento, V. Rebello, and A. Sena. Efficient hierarchical self-scheduling for mpi applications executing in computational grids. In *Proceedings of the 3rd international workshop on Middleware for grid computing MGC '05*, November 2005.

[8] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fédak, C. Germain, T. Hérault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Néri, and A. Selikhov. Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes. In *Proceedings of the* SuperComputing 2002 *Conference*, November 2002.

[9] Aurélien Bouteiller, Franck Cappello, Thomas Herault, Géraud Krawezik, Pierre Lemarinier, and Frédéric Magniette. Mpich-v2: a fault tolerant mpi for volatile nodes based on pessimistic sender based message logging. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2003.

[10] D. Case, D.A. Pearlman, J. W. Caldwell, T.E. Cheatham, W.S. Ross, C.L. Simmerling, T.A. Darden, K.M. Merz, R.V. Stanton, and A.L. Cheng. *Amber 6 Manual*. 1999.

[11] Andrew Chien, Brad Calder, Stephen Elbert, and Karan Bhatia. Entropia: Architecture and performance of an enterprise desktop grid system. *Journal of Parallel and Distributed Computing*, 2003.

[12] Pat Conway and Bill Hughes. The AMD Opteron Northbridge. *IEEE Micro*, 2007.

[13] Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguezb, and Franck Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi. In *Proceedings of the ACM/IEEE SC 2006 Conference*, November 2006.

[14] Lisandro Dalcin. Mpi for python. http://mpi4py.scipy.org/.

[15] Angelo Duarte, Dolores Rexachs, and Emilio Luque. An Intelligent Management of Fault Tolerance in Cluster Using RADICMPI. In *B. Mohr, J.L. Träff, J. Worringen, J. Dongarra (Eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, LNCS 4192, 2006.

[16] M.M. ElSaifi and E.T. Midorikawa. Pmpi: A multi-platform, multi-programming language mpi using .net. http://dotnet.zcu.cz/NET_2006/Papers_2006/short/A59-full.pdf, May 2006.

[17] Graham E. Fagg, Edgar Gabriel, Zizhong Chen, Thara Angskun, George Bosilca, Jelena Pjesivac-Grbovic, and Jack J. Dongarra. Process fault-tolerance: Semantics, design and applications f or high performance computing. *International Journal of High Performance Computing Applica tions*, 2005.

[18] G. Fedak, C. Germain, V. Neri, and F. Cappello. Xtremweb: A generic global computing platform. In *IEEE/ACM CCGRID2001 Special Session Global Computing on Personal Devices*, May 2001.

[19] R. Fernandes, K. Pingali, and P. Stodghill. Mobile mpi programs in computational grids. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPoPP '06*, March 2006.

[20] D. Flores, T. Estrada, M. Taufer, P. Teller, and A. Kerstens. Simba: a discrete event simulator for performance prediction of volunteer computing projects. In *Poster in Proceedings of SC2006,ACM/IEEE Supercomputing 06*, November 2006.

[21] Ian Foster, Jonathan Geisler, William Gropp, Nicholas karonis, Ewing Lusk, George Thiruvathukal, and Steven Tuecke. Wide-area implementation of the message passing standard. *Parallel Computing*, 1998.

[22] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004.

[23] Stephane Genaud and Choopan Rattanapoka. Large-scale experiment of co-allocation strategies for peer-to-peer supercomputing in p2p-mpi. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium*, 2008.

[24] Google Press Center. Google and IBM Announce University Initiative to Address Internet-Scale Computing Challenges. http://www.google.com/intl/en/press/pressrel/20071008_ibm_univ.html, October 2007.

[25] Chuck Hollis. The three faces of cloud. http://chucksblog.emc.com/chucks-blog/2009/06/the-three-faces-of-cloud.html, June 2009.

[26] Joshua Hursey, Jeffrey M. Squyres, Timothy I. Mattox, and Andrew Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, March 2007.

[27] T. Imamura, Y. Tsujita, H. Koide, and H. Takemiya. An architecture of Stampi: MPI library on a cluster of parallel computers. In *Recent Advances in Parallel Virutal Machine and Message Passing Interface*. September 2000.

[28] Timothy H. Kaiser. Pydusa- parallel programming in python. http://sourceforge.net/projects/pydusa/, 2008.

[29] K. Kaneda, K. Taura, and A. Yonezawa. Virtual private grid : A command shell for utilizing hundreds of machines efficiently. In *CCGrid 2002*, 2002.

[30] N. Kanna, J. Subhlok, E. Gabriel, M. Cheung, and D. Anderson. Redundancy tolerant communication on volatile nodes. Technical Report UH-CS-08-17, University of Houston, December 2008.

[31] N. Karonis, B. Toonen, and I. Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 2003.

[32] Rainer Keller, Edgar Gabriel, Bettina Krammer, Matthias S. Mller, and Michael M. Resch. Efficient execution of MPI applications on the grid: porting and optimization issues. *Journal of Grid Computing*, 2003.

[33] D. Kerbyson and K. Barker. Automatic identification of application communication patterns via templates. In *Proc. 18th International Conference on Parallel and Distributed Computing Systems (PDCS-2005)*, Las Vegas, NV, September 2005.

[34] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (Ppopp'99)*. ACM, May 1999.

[35] G. Koenig and L. Kale. Using message-driven objects to mask latency in grid computing applications. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, April 2005.

[36] D. Kondo, M. Taufer, C. Brooks, H. Casanova, and A. Chien. Characterizing and evaluating desktop grids: An empirical study. In *International Parallel and Distributed Processing Symposium (IPDPS'04)*, April 2004.

[37] E. Kotsovinos and A. Williams. Bamboo trust:practical scalable trust management for global public csomputing. In *Proceedings of the 2006 ACM Symposium on Applied Computing SAC '06*, April 2006.

[38] H. Liu and D. Orban. Gridbatch: Cloud computing for large-scale data-intensive batch applications. In *8th IEEE International Symposium on Cluster Computing and the Grid, 2008. CCGRID '08*, May 2008.

[39] Hatem Ltaief, Edgar Gabriel, and Marc Garbey. Fault Tolerant Algorithms for Heat Transfer Problems. *Journal of Parallel and Distributed Computing*, 2008.

[40] R. Mason and W. Kelly. G2-p2p: A fully decentralised fault-tolerant cycle stealing framework. In *Proceedings of the 2005 Australasian workshop on Grid computing and e-research - Volume 44 ACSW Frontiers '05*, January 2005.

[41] Cameron McNairy and Rohit Bhatia. Montecito: A dual-core, dual-thread itanium processor. *IEEE Micro*, 2005.

[42] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, June 1995. `http://www.mpi-forum.org`.

[43] Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface*, July 1997. `http://www.mpi-forum.org`.

[44] Ole Moller Nielsen. Pypar. http://datamining.anu.edu.au/ ole/pypar/.

[45] P. Miller. pyMPI An introduction to parallel Python using MPI. https://computing.llnl.gov/code/pdf/pyMPI.pdf, September 2002.

[46] S. Pakin. Receiver-initiated message passing over rdma networks. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008)*, April 2008.

[47] Z. Pan, X. Ren, R. Eigenmann, and D. Xu. Executing mpi programs on virtual machines in an internet sharing system. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006, 20th International*, April 2006.

[48] S. Rao, L. Alvisi, and H.M. Vin. Egida: An extensible toolkit for low-overhead fault-tolerance. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, FTCS 99*. IEEE Computer Society, March 1999.

[49] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, Winter 2005.

[50] H. De Sterck, R. S. Markel, T. Phol, and U. Rde. Parallel and distributed systems and networking: A lightweight java taskspaces framework for scientific computing on computational grids. In *Proceedings of the 2003 ACM symposium on Applied computing*, March 2003.

[51] Yuji Sugita and Yuko Okamoto. Replica-exchange molecular dynamics method for protein folding. *Chemical Physics Letters*, November 1999.

[52] T. Tabe and Q. Stout. The use of the MPI communication library in the NAS Parallel Benchmark. Technical Report CSE-TR-386-99, Department of Computer Science, University of Michigan, November 1999.

[53] M. Taufer, A. Chahm, A. Kerstens, and C. Brooks III. Predictor@home: A protein structure prediction supercomputer based on global computing. In *IEEE Transactions on Parallel and Distributed Systems*, August 2006.

[54] M. Taufer, A. Kerstens, T. P. Estrada, D. A. Flores, R. Zamudio, P. J. Teller, R. Armen, and C. L. Brooks III. Moving volunteer computing towards knowledge-constructed, dynamically-adaptive modeling and scheduling. In *Submitted to the First Workshop on Large-Scale and Volatile Desktop Grids. In conjunction with IPDPS 2007*, March 2007.

[55] K. Taura, K. Kaneda, T. Endo, and A. Yonezawa. Phoenix: A parallel programming model for accommodating dynamically joining/leaving resources. In *Proceedings of the 9th ACM SIGPLAN symposium - Principles and practice of parallel programming PPoPP'03*, June 2003.

[56] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 2005.

[57] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. Technical report, Ithaca, NY, USA, 1998.

[58] J.P Walters and V. Chaudhary. Replication-based fault tolerance for mpi applications. In *IEEE Transactions on Parallel and Distributed Systems*, July 2009.

[59] R. Zheng and J. Subhlok. A quantitative comparison of checkpoint with restart and replication in volatile environments. Technical Report UH-CS-08-06, University of Houston, June 2008.

[60] D. Zhou and V. Lo. Wavegrid: a scalable fast-turnaround heterogeneous peer-based desktop grid system. In *20th International Parallel and Distributed Processing Symposium*, April 2006.