2008

Vcluster: A Portable Virtual Computing Library For Cluster Computing

Hua Zhang
University of Central Florida

VCLUSTER: A PORTABLE VIRTUAL COMPUTING LIBRARY FOR CLUSTER COMPUTING

by

HUA ZHANG
M.S. University Of Central Florida, 2006

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2008

Major Professors: Ratan K. Guha
Joohan Lee

ABSTRACT

Message passing has been the dominant parallel programming model in cluster computing, and libraries like Message Passing Interface (MPI) and Portable Virtual Machine (PVM) have proven their novelty and efficiency through numerous applications in diverse areas. However, as clusters of Symmetric Multi-Processor (SMP) and heterogeneous machines become popular, conventional message passing models must be adapted accordingly to support this new kind of clusters efficiently. In addition, Java programming language, with its features like object oriented architecture, platform independent bytecode, and native support for multithreading, makes it an alternative language for cluster computing.

This research presents a new parallel programming model and a library called VCluster that implements this model on top of a Java Virtual Machine (JVM). The programming model is based on virtual migrating threads to support clusters of heterogeneous SMP machines efficiently. VCluster is implemented in 100% Java, utilizing the portability of Java to address the problems of heterogeneous machines. VCluster virtualizes computational and communication resources such as threads, computation states, and communication channels across multiple separate JVMs, which makes a mobile thread possible. Equipped with virtual migrating thread, it is feasible to balance the load of computing resources dynamically.

Several large scale parallel applications have been developed using VCluster to compare the performance and usage of VCluster with other libraries. The results of the experiments show that VCluster makes it easier to develop multithreading parallel applications compared to conventional libraries like MPI. At the same time, the performance of VCluster is comparable to

MPICH, a widely used MPI library, combined with popular threading libraries like POSIX Thread and OpenMP.

In the next phase of our work, we implemented thread group and thread migration to demonstrate the feasibility of dynamic load balancing in VCluster. We carried out experiments to show that the load can be dynamically balanced in VCluster, resulting in a better performance. Thread group also makes it possible to implement collective communication functions between threads, which have been proved to be useful in process based libraries.

Dedicated to my dear family.

ACKNOWLEDGMENTS

I would like to thank my advisors, Dr. Ratan Guha and Dr. Joohan Lee, for their invaluable help, support and guidance throughout my degree pursuit. I would also like to thank members of my research committee – Dr. Mostafa Bassiouni, Dr. Jayanta Kapat and Dr. Cliff Zou – for their assistance and guidance in completing this work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES AND LISTINGS

LIST OF ACRONYMS/ABBREVIATIONS

BPNN	Back Propagation Neural Network
CORBA	Common Object Request Broker Architecture
COTS	Commodity Off The Shelf
JDK	Java Development Kit
JNI	Java Native Interface
JVM	Java Virtual Machine
MPI	Message Passing Interface
PVM	Parallel Virtual Machine
RMI	Remote Method Invocation
S-DSM	Software Distributed Shared Memory
SMP	Symmetric Multi-Processor

CHAPTER ONE: INTRODUCTION

1.1 Motivation

Cluster computing [1,2] provides a low cost parallel computing platform consisting of multiple interconnected PCs or workstations through commodity network technology as a replacement for an expensive parallel computer. The most widely used parallel programming model in cluster computing is the message passing model where each processor executes a different stream of instructions and exchanges messages when they need to share data or coordinate with other processors.

Significant improvement in the performance of microprocessors during the last decade promoted the popularity of cluster computing. Recently, processor vendors started producing relatively low cost Symmetric Multiprocessor (SMP) machines. In these machines, multiple processors share the same memory, I/O devices, and other resources and run a single copy of the operating system. Emergence of low cost SMPs and low cost operating systems running on these platforms such as Linux/Unix made it possible to build a cluster of multiprocessors economically. Also, for clusters of SMP machines, less number of machines need to be connected through a slower network and communications among the processors within the same machine can take place using a much faster bus or an interconnected network. This improved the performance as well as the scalability of a cluster. Moreover, fast communication between processors on a same machine make it possible to achieve better granularity as work can be further divided within a physical machine.

Current message passing programming models do not support this new type of cluster computing architecture effectively. To attain better granularity, multithreading techniques, POSIX Thread [16] and OpenMP [17], have been widely used in shared memory programming for SMP machines. For clusters consisting of SMP machines, it is desired that the shared memory programming model be incorporated into the programming model to support intra-node communication while the message passing model is still used to support inter-node communication. A parallel programming model that supports this architecture must address two different parallel computing paradigms in a single framework. In shared memory multiprocessors, multithreading and synchronization based on shared variables are common techniques. In message passing, communication is defined between two processes rather than threads and explicit message exchanges are used for work distribution, data sharing, coordination, and synchronization.

While many of the existing parallel programming libraries have been targeted for C/C++ and Fortran programming languages, another programming language, Java, and its associated technologies opened a door to portable and integrated development of distributed computing software, due to its built-in thread support, platform neutral byte codes, concurrent programming model based on monitor concept, object oriented, and inter-process communication mechanisms such as TCP/IP sockets, Remote Method Invocation (RMI) and Common Object Request Broker Architecture (CORBA). Recently, Java has also enforced its viability as a distributed computing tool by incorporating Java cryptography and security packages as a part of recent JDKs.

1.2 Features of VCluster

Motivated by the above observations, we have developed a new parallel programming model and a parallel runtime system, VCluster, which implements this model. VCluster is a prototype realization of the proposed framework. VCluster addresses both tightly coupled shared memory multiprocessors and loosely coupled distributed memory multiprocessors, a cluster, in a single framework. The main features of VClsuter are described below.

Support of Multithreading

VCluster is a thread-based model instead of a process-based model. The basic processing unit in VCluster is a virtual thread, which is built on top of Java thread. Through this feature, any parallel application developed on top of VCluster will support multithreading, to better utilize clusters of SMP machines.

Support of Message-Passing and Shared-Memory in One Framework

Message-Passing between threads is not supported by conventional message passing libraries, which only support communication between processes. In VCluster, Virtual channels are established between two virtual threads that want to communicate through message-passing. If two threads are on different nodes, the messages are sent through the network; if two threads are on a same node, the message is sent through memory copy. This is carried out by the channels and is totally transparent to the user.

Also, one or more virtual states can be associated with a virtual thread. If two threads on a same node want to communicate through shared-memory, a virtual state can be associated with two threads at the same time.

Support of Thread Migration

Information stored for a Java thread that is bounded to a local machine is too difficult and large to migrate to a different machine due to the large overhead of the transfer. In VCluster, a parallel application stores the computation information in virtual states instead of virtual threads. When a virtual threads need to migrate, the virtual state is transmitted to another machine through object serialization [20] and a new virtual thread is created and it can continue the computation using this virtual state. Channels are automatically updated by VCluster so that threads can continue communicate after migration.

Support of Thread Groups

In MPI, group members are processes and collective communications are defined between processes. For clusters of SMP machines which favor thread-based applications instead of process-based applications, this becomes very inconvenient. VCluster support groups of threads and collective communications between threads.

Designed for Java

Our programming model is designed based on Java threads and other features of Java. The architecture is totally object-oriented, making it easy to develop and maintain large scale parallel applications. VCluster is implemented using 100% Java, and can be used on clusters of heterogeneous machines due to the platform neutral Java bytecode.

Support a Wide Range of Computer Platforms

Although VCluster is primarily designed for cluster computing, it can also run on SMP machines, computers in a local area network, or even computers on the internet. In VCluster, a daemon is running on every participating node and acts as an agent. These agents communicate through TCP/IP sockets. Theoretically speaking, as long as a collection of computer nodes are connected through a TCP/IP network, they are eligible for VCluster.

1.3 Contributions of the Thesis

We proposed a new parallel programming model that can achieve better granularity on cluster of SMP machines and implemented this model as VCluster, a portable virtual computing library for cluster computing. The contributions of our work are listed as follows:

1. VCluster can achieve better granularity on clusters of SMP machines through multithreading. With better granularity, work can be further divided for possible parallel execution to improve performance.
2. VCluster support the shared memory paradigm and message passing paradigm in a same framework, making it much easier to develop parallel applications for clusters of SMP machines.
3. VCluster supports thread migration through virtual states, which can be used for advance features like dynamic load balancing and fault tolerance.
4. VCluster support point-to-point and collective communications between threads, which is lack in conventional message passing models.
5. VCluster is designed with an object oriented framework, which is necessary to develop large scale parallel applications.

1.4 Organization of This Thesis

The rest of this thesis goes as follows. In Chapter 2, we introduce the background and research related to this dissertation and compare approaches used in VCluster to other related middleware libraries. Chapter 3 describes the architecture and implementation of VCluster. In Chapter 4, we discuss the method to develop parallel applications using VCluster. Chapter 5 presents the experiments and results comparing VCluster with other approaches, including single-threading and multithreading. After that, we discuss a performance prediction model in chapter 6. Load balancing is discussed Chapter 7. Finally future works and conclusion are given in chapter 8.

CHAPTER TWO: LITERATURE REVIEW

In this chapter, we first discuss the background of cluster computing and Java language. Then works and techniques that are related to the VCluster research are discussed. They are divided into the following sections: Java Libraries for Cluster Computing, Cluster-wide Multithreading, and Thread/Process migration. Each section is organized as a small survey of the related works and techniques. Also in each section, we compare approaches proposed or adopted by VCluster with related approaches and explain why they are chosen for VCluster.

2.1 Cluster Computing

"A cluster is a type of parallel or distributed processing system, which consists of a collection of interconnected stand-alone computers working together as a single, integrated computing resource." [1]

Rapid advances in high performance commodity computers and networks made cluster computing [1, 2] an attractive solution to large scale parallel applications. Cluster computing provides a cost-effective parallel computing platform as a network of PCs or workstations, providing a comparable performance to those expensive, specially designed parallel supercomputers at a fraction of the cost. The 31st TOP500 list [3], which was released in June 2008, shows that 400 out of the 500 fastest computer systems in the world are clusters. Moreover, as clusters are low-cost and easy to build, they drastically improve the availability of supercomputing to small companies and organizations. This is why clusters are also called *commodity supercomputing* [2].

Clusters are normally built up with commodity-off-the-shelf (COTS) hardware components, e.g. PCs or workstations and network products. Each PC or workstation in a cluster is a stand-alone computer with its own processors and memory, and is equipped with a network interface card (NIC). We will call it a node, or a computer node throughout this thesis. A copy of operating system is running on each of the node. The most accepted operating systems for clusters are Linux/Unix systems. They are free, stable, and have excellent support for networking. Different versions of Windows operating systems are also widely used in clusters, mostly because of their popularity in PCs or workstations, and the availability of abundant applications and tools. These computer nodes are then connected through their NICs to form a cluster. A parallel application running on a cluster has one or more processes running on each node, and uses the interconnecting network for communications between these processes.

Here is a list of the advantages of clusters [1, 4, 5]:

- SMP machines have been available to the market at a relative low cost, so it is relatively cheap and easy to build a cluster of SMP machines.
- New network technologies and protocols are implemented to increase the bandwidth and reduce the latency.
- Clusters have good scalability. The number of nodes in a cluster can be easily increased. Each node can also be easily upgraded by upgrading the components, which are standardized commodity products that can be easily replaced or upgraded.
- As clusters use free or commodity operating systems like Linux/Unix and Windows, the applications and tools are more mature and abundant compared to conventional specialized proprietary parallel supercomputers.

- Clusters are easier to be integrated into existing networks. Networking enables remote operation of clusters and makes it possible to share resources of clusters by a wide range of people. There is an increasing trend to connect several clusters through internet to provide a larger computing platform.

A parallel application on a cluster runs processes on every node in the cluster, and these processes communicate through the network that connects these nodes. This work is difficult, hard, tedious and error-prone. To facilitate the programming on clusters, an extra layer is introduced between the application and the operating system running on the nodes. This layer is called middleware. Middleware often has a programming model and an associated library that implements this model. Users develop their parallel applications on top of these libraries as if they were programming on a single computing resource. This greatly simplified the development and a standardized library can improve the portability of the parallel application developed. The most widely used programming models used in clusters are message passing and software distributed shared memory (S-DSM).

In message passing, each node executed a different stream of instructions and exchange message when they need to share data or coordinate with other nodes. Message Passing Interface (MPI) [6] has been used as a de facto standard for message passing based parallel computing. MPI specifies the necessary point-to-point and advanced collective communication primitives for message passing. MPI and other message passing libraries such as Parallel Virtual Machine (PVM) [67] have been widely used in developing parallel applications, proving its effectiveness due to simplicity and portability over various parallel computing platforms.

In software distributed shared memory, the middleware library presents the cluster as a shared-memory, multi-processor supercomputer. Processes in different nodes communicate as if they were sharing a big memory space, and the correctness of this sharing is guaranteed by the underlying middleware. Contrary to message-passing, there is no standard for S-DSM yet. The idea of S-DSM was firstly proposed by Li and Hudak. They also implemented the first S-DSM system, named IVY [21, 22]. TreadMarks [7] is another widely used page-based distributed shared memory system that provides a global shared address space across the different nodes on a cluster.

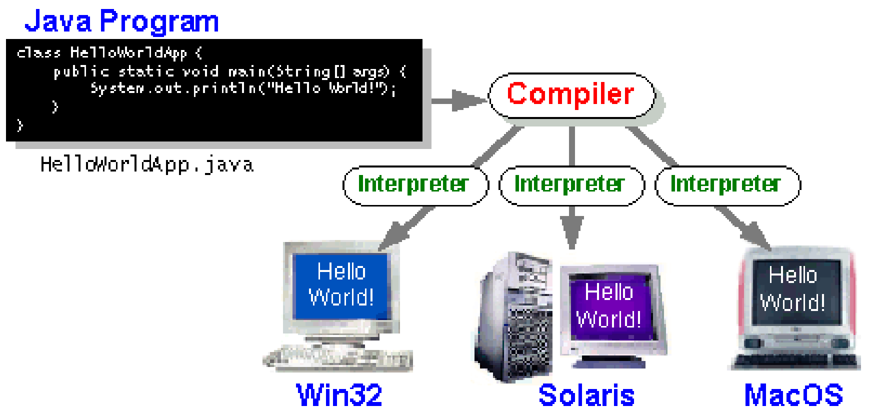
Comparing these two programming models, message passing is generally thought to yield better performance, although harder to program with. Message passing is still the dominant programming model in cluster computing. VCluster adopts the message passing model.

2.2 The Java Language

Java [8] was first introduced by Sun Microsystems to be used as an internet language, because its binary code (bytecode) can run on machines with any architecture without re-compilation, as long as Java Virtual Machine (JVM) [9] is installed. As clusters composed of heterogeneous machines become popular, this feature becomes desirable for cluster computing. Also, other features like native support for multithreading, concurrent programming model based on *monitor* concept, object oriented framework, inter-process communication mechanisms such as TCP/IP sockets, Remote Method Invocation (RMI) [18] and Common Object Request Broker Architecture (CORBA) [19], all make Java an attractive language for parallel computing. Java has been known for inferior performance compared to C/C++ and FORTRAN, but new technologies like Just-In-Time (JIT) compiler [10] and HotSpot [11] have significantly improved the performance of Java. Some results show that Java can compete with the performance of C++ [12]. Several interesting features of Java are briefly introduced below.

Bytecode

When a Java source file is compiled, it is compiled into a class file that is composed of bytecode. Bytecode can be called the machine code of Java Virtual Machine (JVM). To run a class file, JVM, acting like an interpreter, translates bytecode to native machine code. In this way, the same class file can ran on machines with different architectures without re-compilation as long as a JVM is installed. Figure 1 is an illustration of this approach.



```
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Figure 1: Portability of Java

Multithreading

Java provides Java threads which is build upon native threads of the operating system. But as far as the user is concerned, the user just needs to deal with Java threads. And the user can be confident that the code can run on different architectures without problem. Every Java object is a monitor. The *synchronized* keyword enables exclusive access to shared variables. And the *wait* and *notify* keywords provide mechanisms for threads to coordinate with each other.

Just-In-Time compiler

The JIT compiler compiles Java bytecode to native machine code on the fly at runtime. This compilation causes some initial slowdown, but can significantly improve the performance afterwards.

HotSpot technology

The HotSpot technology can further improve the performance by performing runtime analysis of the frequency of the code segments execution and optimizing the performance of the code that are frequently executed.

2.3 Java Libraries for Cluster Computing

In cluster computing environment where computers have their own private memories and are loosely coupled through a slower network, message passing has been the dominating parallel programming paradigm. Data sharing, coordination, and synchronization are done through explicit message exchanges. In this parlance, MPI and PVM are the most successful implementations and are used in developing many different types of parallel applications. Message passing architectures favor coarse-grained parallelism because send and receive operations can be designed to amortize latency over the long messages that are sent infrequently in coarse-grained programs [23].

Another widely used programming paradigm for cluster computing is Software Distributed Shared Memory (S-DSM). S-DSM is a software mechanism that provides a virtual shared memory address space to distributed applications, although the memory is physically distributed, as the memory on different nodes in a cluster.

With its advanced features (Section 1.2), Java has been widely used to develop both message passing and S-DSM systems. A recent work [51] compared the performance of message passing Java libraries and S-DSM Java libraries. Message passing libraries generally gives better performance than distributed shared memory. VCluster falls in the Message passing category. For this reason, only Java libraries adopting the message passing model are discussed in this section. Java S-DSM systems [24-30] are listed in the reference for interested readers. Some of them will also be discussed in the following two sections.

Java libraries adopting the message passing model can be divided into two groups: Java wrappers and pure Java. Java wrappers actually call native MPI method in C, C++ or FORTRAN through Java Native Interface (JNI) [66], and generally provide better communication performance. However, Java wrappers compromise the portability of Java, which is desirable as clusters of heterogeneous machines become popular. Pure Java implementations, on the other hand, maintain the portability of Java but provide a less efficient communication. Some of these systems are described below. More comprehensive discussions of these libraries can be found in [37, 38].

mpiJava [31] is the most active Java wrapper project, providing a C++ like, Object-Oriented interface by calling native MPI methods through JNI. It was developed as part of the HPJava project [32], but it is compatible with all implementations of Java language.

JMPI [33] is a pure Java implementation developed for academic purposes at the University of Massachusetts, following the MPJ [34] standard, which is proposed by the Message Passing Working Group within the Java Grand Forum (www.javagrande.org).

JPVM [35] is a pure Java implementation of PVM. However, it does not follow all the specifications of PVM and thus is not compatible with standard PVM.

In a MPI/PVM system, as communications are only defined between processes, users must take care to make sure a message is received by the correct destination thread, e.g. using a specified tag value. This becomes very difficult, if not infeasible, when thread migration are incorporated to support load balancing. Another problem is collective communication between threads. Collective communications between processes in MPI have been proven very useful to scientific applications. When multithreading is imported to better support clusters of SMP machines, collective communications between threads are desirable. Without direct communication between threads, collective communications between threads are impossible.

CCJ [36] is another pure Java library. It does not follow the MPI or the MPJ specification. Instead, it is designed to integrate the communication into Java’s Object-Oriented framework. It supports inter-thread communication and thread group instead of inter-process communication and process group as in the MPI specification.

VCluster defines communications between threads. A virtual channel is created for two threads to communicate. When two threads are on different machines, the message is transmitted through the underlying network; when they are on a same node, the message is transmitted through memory copy. VCluster also supports shared-memory between threads, by associating a virtual state with several threads. Thread groups and collective communication between threads are supported in VCluster as well. Moreover, virtual channels and thread groups will be automatically updated by the VCluster system after thread migration, which is very useful when thread migration is used to implement load balancing.

2.4 Cluster-wide Multithreading

While clusters of SMP machines become popular, it is desired to incorporate multithreading into cluster computing. For the message passing model, this forms a two-level architecture: shared-memory is used for communications between threads on the same node (intra-node communication) and message-passing is used for communications between threads on different nodes (inter-node) communication. For the S-DSM model, local shared memory can be merged into the distributed shared memory model, forming a virtual one-level architecture. This saves the user from dealing with two types of communication all the time, and makes it much easier to do thread migration as a thread will communicate in the same way with all other threads before and after migration. Otherwise, when a thread migrates to another node, the original intra-node communication will become inter-node communication, and inter-node communication with threads on the destination node will become inter-node communication. VCluster also provides this virtual one-level architecture, yet in message-passing instead of distributed shared-memory. The virtual channel between two threads hides the differences of inter-node communication and intra-node communication to the user.

Systems or approaches that provide multithreading are discussed below. We divide them into two categories, those do not use Java threads and those use Java threads.

2.4.1 Non-Java Approaches

For non-Java languages like C/C++, as thread is not supported by the language, multithreading must be supported through additional thread libraries, or extending the language.

Multithreading libraries, POSIX Thread [39] and OpenMP [40], have been widely used in shared-memory programming for SMP machines. POSIX Thread (also called PThread) provides thread APIs for C/C++. Users can spawn new threads, and use *mutex, join, condition* mechanism for thread synchronization. OpenMP provides both APIs and directives. Threads are created implicitly through defining parallel structures. OpenMP favors large array-based applications. It provides special directives to parallelize loops, like the *for* loop. Synchronization mechanisms are also provided, e.g. the *barrier* directive.

For clusters of SMP machines, it is very natural that people want to combine message passing or S-DSM libraries with multithreading libraries. [41], [42] combine OpenMP with DSM systems, and [44] combines PThread with MPI systems. [43] [45] conducted performance evaluation of these approaches and concluded that MPI/S-DSM systems + Multithreading give better performance than sole MPI/S-DSM systems in most cases on clusters of SMP machines, as "they exploit better the configuration characteristics of an hierarchical parallel platform"[43]. A problem with this kind of approaches is that most MPI/S-DSM systems are not thread-safe. That is, if two or more threads are calling library APIs at the same time, for example, two threads want to send messages at the same time, exceptions may occur. A popular solution is to let only one thread call library APIs, on behalf of the other threads, but this makes development of parallel applications awkward, tedious, and error prone.

Also built on top of a threading library, CLAM [82, 83] uses its own parallel computing architecture. CLAM means Connection-less, Lightweight, and Multi-way. It is build on top of the Ariadne threads library [84]. CLAM provides efficient and scalable user-space support for distributed applications requiring multiple protocols. Multithreading is used both in the communication protocol and the message passing software. For example, one of its communication module has three threads, a *receive*, a *send*, and a *timer* threads.

Another approach is to extend parallel languages, like JR [46] and Cilk [47]. Cilk is an algorithmic multithreaded language based on ANSI C. The two basic parallel keywords are *spawn* and *sync*. *spawn* is used to spawn a new thread, and *sync* is used for synchronization. Cilk is especially effective for exploiting dynamic highly asynchronous parallelism, for example, the divide-and-conquer paradigm. Distributed Cilk [48] implements the basic features of Cilk, and implements its own distributed shared memory, supporting a memory consistency model called *dag-consistent shared memory* [52]. SilkRoad [49] and SilkRoad II [50] further extends Distributed Silk by implementing the *Lazy Release Consistency* memory model [53] to support user level shared memory which is necessary in many multithreaded applications. The thread safe problem is eliminated in this approach. However, it requires the user to learn a new parallel programming paradigm other than the popular message passing and S-DSM parallel paradigms.

2.4.2 Multithreading Through Java

Since multithreading is already supported by the Java language, Java libraries can simply use Java thread to implement multithreading. However, the multithreading model in Java is only for SMP machines. In a computing environment as cluster, the multithreading model in Java must be extended to a cluster level, which includes at least two requirements: creating thread remotely and provide communication mechanism for threads on different nodes. Different approaches have been proposed and implemented to provide cluster wide multithreading. But most systems choose S-DSM as the communication mechanism, because message passing model is de facto for inter-process communication. An exception is CCJ [36], which uses its own thread-based message passing model.

Techniques to implement cluster wide multithreading in Java can be divided into four categories: Distributed JVM (DJVM), Code Transformation, Bytecode Instrumentation, and Agent-based Systems.

Distributed JVM

A distributed Java Virtual Machine (DJVM) spans the whole cluster and provides a parallel execution environment for multithreaded Java applications. Java/DSM [27], cJVM [26], JESSICA [25], JESSICA2 [24] all implements some kind of DJVM. In these systems, a modified, nonstandard JVM is installed on every node. Java/DSM, cJVM, JESSICA modify the Java interpreter. This can seriously affect the performance of parallel applications as JIT compiler can not be used. JESSICA2 distinguish itself by providing a dedicated JIT compiler. The main disadvantage of this approach is that it compromises portability, as they use non-standard Java Virtual Machine.

Code Transformation

In this approach, the source files or the compiled bytecode files (class files) are transformed into native machine code. Hyperion [28] includes a Java-bytecode-to-C translator and a runtime library for the distributed execution of Java threads. Java bytecode is translated to C source code and then compiled to native machine code. Portability is compromised by the hardware-specific runtime library. Jackal [29] consists of an extended Java compiler and a runtime system for distributed shared memory. The compiler compiles Java sources codes to Intel x86 code. The compiler stores Java objects in shared regions and augments the program it compiles with access checks. This machine dependent compiler also compromises portability.

Bytecode Instrumentation

Systems adopting this approach take bytecode as input, and output the instrumented bytecode. Portability of Java is kept as the instrumented bytecode can run on standard JVM. JavaSplit [30], JavaParty [54], JDSM [55] are this kind of systems. JavaSplit support the original multithreaded paradigm of Java, which distinguish it from the other two that both introduce unconventional programming constructs and style.

Agent-based Systems

In these systems, a software agent is running on every node in the cluster. Software agents have been used in many systems to enhance the performance and quality of services [57]. These software agents provide services include job distribution, monitoring, and controlling for the system. An agent based Infrastructure is proposed in [56] as showed in Figure 2. The whole structure is implemented in pure Java, so it keeps full-fledged portability of Java. VCluster adopts this approach.

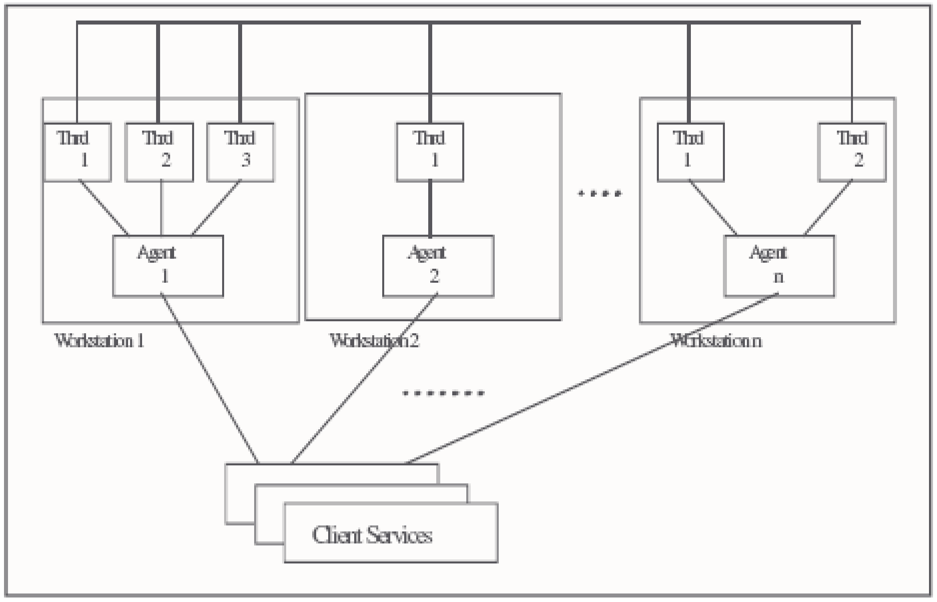


Figure 2: An Agent Based Infrastructure

Using agents has a lot of advantages [56]:

Portability: Despite the excellent portability of Java, it does not hide the architecture of the underlying computation platform. With agents running on a collection of nodes, a parallel application can run on these nodes regardless of how these nodes are connected.

Expandability: A new node can be added into the system by activating an agent on the machine. This can be totally transparent to the user and the parallel application. For those parallel applications that cannot be stopped, this is quite desirable.

Flexibility: Since the infrastructure is hidden from the user and the parallel applications, it is easy to modify and expand the underlying computation platform.

Security: Communication through the agents can be encrypted through a security module. This module can be removed, added, replaced, and upgraded transparently to the parallel application.

Resource management: Agents can collect the necessary information about the resources of their resident machines. This provides a distributed information base for all the system resources.

2.5 Thread/Process Migration

Thread/process migration can be used to improve the performance and robustness of a parallel application. It can be used in dynamic load balancing, or fault tolerance, e.g. to migrate a thread/process to another node when the original node is attacked. Also, for non-stoppable applications, threads/processes can be migrated without stopping when the original node is attacked or need to be shut down for maintenance. Some process migration systems are briefly introduced and we focus on thread migration in Java.

Process migration is implemented in [58-60]. [58] is based on Gobelins, which is a cluster operating system that provides a Distributed Shared Memory (DSM) system. [59] implements process migration based on the existing checkpoint/restart package, Epckpt. [60] provides a Java wrapper to MPI and support process migration through the Java Virtual Machine Debugger Interface (JVMDI) [65]. Process state is captured through JVMDI, and restored at the destination node. A MPI daemon is running on every node to deliver messages on behalf of the Java process. After a Java process migrates, the communication channels are automatically reconstructed so that processes can keep communicating.

What makes thread migration difficult is the machine dependent nature of threads. Same as processes, a thread context has much machine dependent information, such as the hardware program counter. This information cannot be simply transmitted to another machine as they will not work. To migrate a thread, machine dependent and machine independent information must be handled differently and a mechanism to continue the thread execution on the destination node must be provided.

JESSICA [25] is a multithreading S-DSM system in Java. It implemented a preemptive and transparent thread migration mechanism called Delta Execution [61]. The Delta Execution divides the execution context of a running thread into machine-independent sub-contexts and machine dependent sub-contexts. Machine-independent sub-contexts are migrated in a regular manner when a thread migrates. Machine-dependent sub-contexts will be left on the *home node* and execution involving these sub-contexts will continue to be performed there. Executions on the machine-dependent and machine-dependent sub-contexts will switch back and forth between the home and the remote node, making the migrated thread observed as in active execution. Delta Execution is transparent to the user and the parallel application as it is at the bytecode level, but it requires a Bytecode Execution Engine (BEE) which is part of JESSICA. Bytecode Execution Engine in turn requires modifications to the standard JVM, which compromises portability. Moreover, Bytecode Execution Engine cannot utilize the JIT compiler, unless a specialized JIT is developed, thus not suitable for high performance applications.

To tackle the JIT problem, JESSICA2 [24] first proposed a new migration mechanism through dynamic native code instrumentation inside the JIT compiler [62]. JESSICA2 uses a JIT compiler based execution engine (JITEE) compared to the BEE in JESSICA. The JIT compiler transform the *raw thread context* (RTC), which usually includes the virtual memory space, thread execution stack and hardware machine registers, to *bytecode-oriented thread context* (BTC), which consist of the identification of the thread, followed by a sequence of frames. Then RTC can be re-established using the BTC on the destination machine. Since this approach will cause extra space and time overheads, JESSICA2 later implemented another new migration mechanism through JIT recompilation [63]. A JIT compiler is used to recompile the Java methods in the stack context to aid the extraction and restoration of the thread context at the migration time. This approach achieves high-speed native thread execution without redundant code and thread mobility.

The three migration mechanisms used in JESSICA and JESSISA2 all requires a customized JVM, as either the interpreter or the compiler need to be modified. CEJVM [64], on the other hand, uses JVMDI [65] and JNI [66] to implement thread migration. However, JVMDI needs huge data structures and incurs large time overhead in supporting the general debugging functions. JVMDI also requires Java applications to be compiled with debugging information using specific Java compilers.

Visper [104] implements thread migration through the concept of remote thread, which is not a Java thread. Visper defines an *RTRunnable* interface, which extends the Java *serializable* interface and defines a *Run* function. Any class can become a remote thread by implementing this interface. A Java thread is allocated to run a remote thread by calling the *Run* function. A remote thread does not contain machine dependent context, so its state can be saved through serialization and transferred to another node. After that, its *Run* function is called again. The *Run* function must be coded by the user to handle the resumption of execution. In this mechanism, thread migration is not transparent to the user as in the cases of the previous mechanisms, but it is efficient and uses general JVM.

In VCluster, we use an approach which is totally different to the above approaches. The basic execution unit in VCluster is a virtual thread, and a virtual state is associated with a virtual thread. This virtual state contains all the computation data a virtual thread will need. In another words, a new virtual thread can continue the work of another virtual thread simply by given the virtual state of that virtual thread. Under this framework, to migrate a thread, we actually transmit the virtual state to another machine, create a new virtual thread, and let the new virtual thread continue the execution based on the virtual state. This is also why we call the threads *virtual* threads and hence the name of our library, VCluster. A virtual thread is not bound to a specific machine, and it can move around the whole cluster, for example, in case of load balancing or fault tolerance. This approach should be more effective and induce less overhead compared to the above approaches, as only the computation data, which is the necessary part, is actually migrated.

2.6 Comparison of libraries by features

Based on our discussion above, we provide a table that shows the comparison of all these libraries on features we discussed. We can see that most Java libraries that support multithreading and thread/process migration are based on the S-DSM model. Visper and VCluster are the few Java libraries that support multithreading and thread migration under a message-passing based architecture.

Table 1: A Comparison of Libraries for Cluster Computing

	Parallel Model		Programming Language		Support Multithreading	Support Thread Migration
	Message Passing	S-DSM	Non-Java	Java		
CCJ	X			X	X	
CEJVM		X		X	X	X
cJVM		X		X	X	
CLAM	X		X		X	X
Distributed Cilk		X	X		X	
Hyperion		X		X	X	
IVY		X		X		
Jackal		X		X	X	
Java/DSM		X		X	X	
JavaSplit		X		X	X	
JavaParty		X		X	X	
JDSM		X		X	X	
JESSICA		X		X	X	X
JESSICA2		X		X	X	X
JMPI	X			X		
jPVM	X			X		
MPICH	X		X			
mpiJava	X			X		
SilkRoad		X	X		X	
SilkRoad II		X	X		X	
TreadMarks		X	X			
Visper	X			X	X	X
VCluster	X			X	X	X

CHAPTER THREE: VCLUSTER – ARCHITECTURE AND IMPLEMENTATION

In this chapter, we first discuss the design of the VCluster architecture, a thread-based architecture that supports message passing and shared-memory in one framework. Then we describe the implementation of this architecture in Java.

3.1 Design Philosophy

We designed our parallel model with the following guidelines:

A Message Passing Framework

In cluster computing environment where computers have their own private memories and are loosely coupled through a slower network, message passing has been the dominating parallel programming diagram. Data sharing, coordination, and synchronization are done through explicit message exchanges. Message Passing Interface (MPI) [6] has been used as a de facto standard for message passing based parallel computing. MPI specifies the necessary point-to-point and advanced collective communication primitives for message passing. MPI and other message passing libraries such as Parallel Virtual Machine (PVM) [67] have been widely used in developing parallel applications, proving its effectiveness due to simplicity and portability over various parallel computing platforms.

Thread Based Message Passing

With the advance in microprocessor technologies, clusters of SMP machines have become available at a relatively low cost. Conventional message passing libraries, like MPI and PVM, are de facto parallel programming libraries for clusters which consist of usually homogeneous and uni-processor machines and lack the necessary technique, multithreading, to support this new type of cluster efficiently. Moreover, point-to-point and collective communications between threads must be supported. Multithreading can be imported into current message passing libraries using multithreading libraries, like PThread and OpenMP. But the later can be very difficult given that the communications are defined between processes in MPI. A thread based model will not have any of these problems. Multithreading will be natively supported, and communication will be defined between threads, which in turn facilitate collective communications between threads. Besides, as conventional message passing libraries are not designed to support multithreading, their code is not thread safe and will induce many problems when multithreading is incorporated. The first suggestion people should follow to develop MPI+PThread or MPI+OpenMP parallel applications is to make sure that only one thread is calling MPI functions at any time, otherwise unexpected exceptions might happen. This again will not be a problem for thread based systems.

Message Passing and Shared Memory in one Framework

Under the architecture of a cluster of SMP machines, it is desired that shared memory is used for intra-node communication and message passing is used for inter-node communication. This requires the parallel model to support both shared memory and message passing in one framework. We go one step further and propose an extra feature, merging local shared memory into global message passing and providing a uniform message passing inter-thread communication mechanism to the user. This feature unifies inter-node communication and intra-node communication into one type of communication, inter-thread communication, saving the user from dealing with two types of communications all the time. Also, thread migration and load balancing will be much easier with this feature, as the communication mechanisms between the migrated thread with other threads will be the same before and after the migration.

Portability and Maintainability

While scalability and better speedup have been the main driving motivations of parallel and distributed processing, we cannot emphasize the importance of portability and maintainability enough in more recent cluster computing environments where a cluster is composed of heterogeneous machines rather than homogeneous ones. The number representation formats can be different, big endian and little endian. Although it can be handled by the user who is aware of the heterogeneity and converts the number representations accordingly, heterogeneity is not hidden from the users and it reduces the portability of the developed parallel programs.

It is widely accepted that Object oriented programming can promote greater flexibility and maintainability in programming, especially for large-scale software engineering. However, the MPI standard is not designed toward an object oriented architecture.

Designed for Java

As an object-oriented programming language and a platform independent environment, Java has been intensively studied for its application in high performance computing and a variety of Java libraries for cluster computing have been developed. These libraries either provide a Java wrapper to native MPI libraries, or implement the MPI standard in pure Java. However, simply applying Java technologies to the existing message passing model is not an efficient solution. Those message passing models were designed to augment the existing sequential programming languages with additional message passing functionalities, and not designed towards the object-oriented paradigm.

Other specific features of Java also need to be taken into consideration when design a new parallel model. For example, the model should leverage the multithreading paradigm of Java to better support clusters of SMP machines.

Support Thread Migration

Thread migration can be used to improve the performance and robustness of a parallel application. It can be used in dynamic load balancing, or fault tolerance. Also, for non-stoppable applications, it can be migrated without stopping.

Security

Security seems to be neglected by current message passing libraries, as clusters nowadays tend to be a close collection of nodes in which nodes can trust each other. However, this mutual-trust between nodes is not always true, especially with the emergence of so called *mini-grid computing*, in which multiple clusters form a larger scale computation platform. The new parallel model must be able to enforce security mechanisms.

Recently, Java has also enforced its viability as a distributed computing tool by incorporating Java cryptography and security packages as a part of recent JDKs.

3.2 VCluster – The Architecture

Following the guidelines in the previous section, we designed our new parallel programming model, VCluster. VCluster provides a framework to support a cluster of heterogeneous uni- or multi-processor machines using a portable parallel programming library. VCluster combines the shared memory and message passing in a single framework. The initial design of VCluster was inspired by SCPlib and CRlib, which are both multithread message passing libraries with richer set of advanced functionalities such as dynamic load balancing, thread mobility, heterogeneous computing, replication, and fault tolerance [68, 69]. They were primarily developed for parallel C programs over a variety of parallel computing platforms.

VCluster is designed to act as a middle-ware on top of the operating system to facilitate development and execution of parallel, multi-threading programs over a cluster of SMP machines. This cluster architecture is reflected in VCluster as global architecture. Global architecture is composed of local architectures, which reside on every node in the cluster. The design and implementation of these architectures are described in the following sections.

3.2.1 Global Architecture

The global architecture is showed in Figure 3. A basic computing unit in the VCluster is a communicating virtual thread, which is built upon Java thread. One or more virtual threads can be running on every Java Virtual Machine (JVM), and one or more Java Virtual Machines can be running on every physical node.

Unlike the message passing model where communication resources are bound to a process, in VCluster, communication resources are associated with an individual thread. A virtual channel is created between any two threads that want to communicate with each other. The two threads can be on different node, or on the same node. This virtual channel provides message passing functions, and decide how to send the message according to the locations of the destination thread, through the underlying network or through shared memory. In this way, virtual channels hide the difference between inter-node communication and intra-node communication from the user.

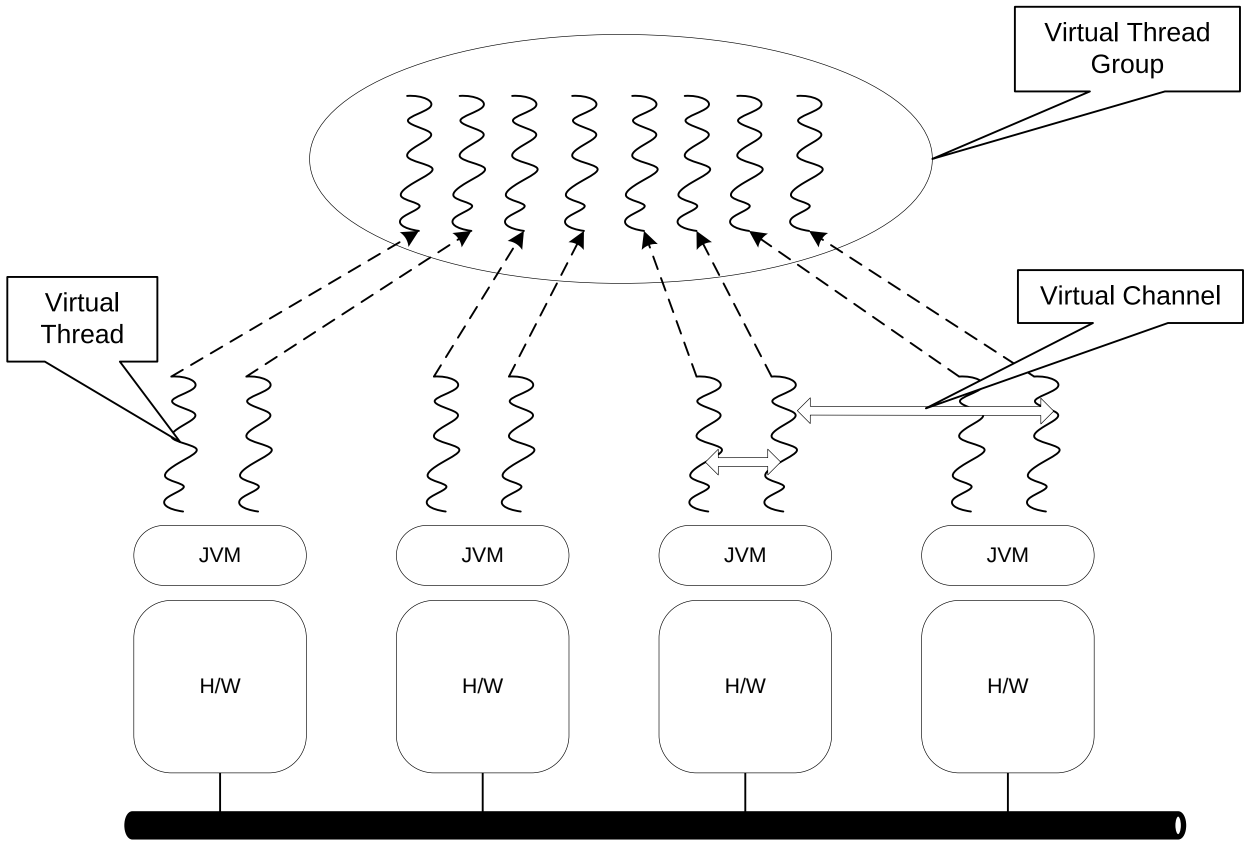


Figure 3: Global Architecture

All or part of the virtual threads can form virtual thread groups. A virtual thread can be a member of several thread groups. Collective communication is supported between threads in a thread group.

A virtual thread can migrate across the Java Virtual Machines and physical computers, to facilitate load balancing or fault tolerance through thread migration. After a thread migration, corresponding virtual channels and/or virtual thread groups will be automatically updated by the system so that threads can continue communicating with each other.

3.2.2 Local Architecture

By *local architecture,* we refer to the architecture of a VCluster parallel application on a Java Virtual Machine. This architecture is actually a collection of virtual threads. A thread creates a set of uni-directional communication channels to communicate with other threads. A thread also associates itself with a set of states of interest that are shared data and methods. The same state can be shared by multiple threads and the atomic and mutually exclusive accesses to the states are guaranteed. This gives another type of shared memory communication between threads.

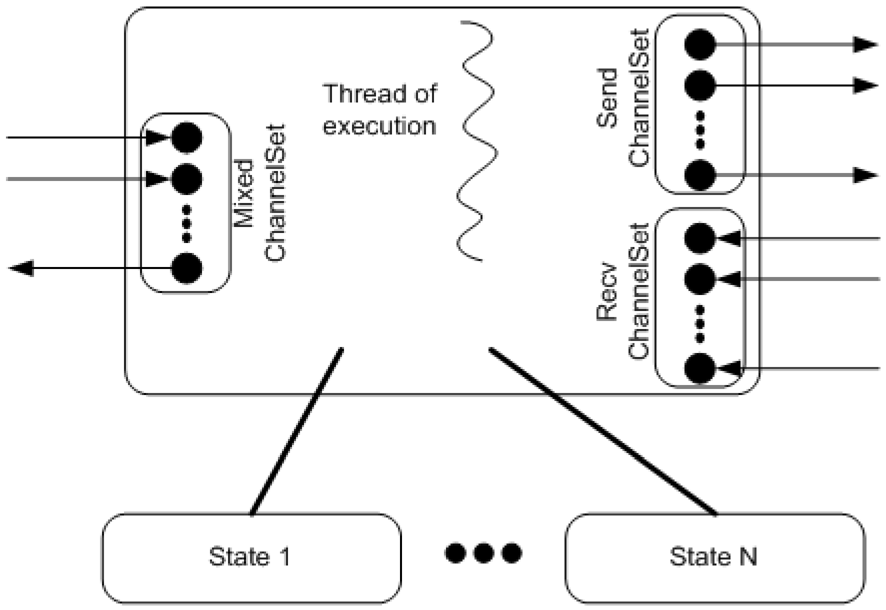


Figure 4: Thread Architecture

Java's object oriented paradigm and polymorphism provide a useful framework to hide the complexity of the software structure and make the parallel program modular and the layered software structure more effective. Programmer write the parallel applications by extending

VCluster thread, state, channel, and channel set classes and inheriting their properties, which hides the internal complexity of those abstract concepts.

3.2.3 Thread Migration

Thread migration is difficult because of the machine-dependent part of thread context. Approaches in JESSICA [25], JESSICA2 [24], CEJVM [64] are *strong migration* mechanisms, as thread migration is transparent to the user. However, they incur extra overhead on the execution after migration. Furthermore, they require customized or specific interpreters or compilers, which compromise portability. On the other hand, Visper [104] implements a *weak migration* mechanism. Its mechanism uses general JVM and compilers, but thread migration is not transparent to user applications.

VCluster also implements a *weak migration* mechanism, using the concept of virtual state. In VCluster, the concept of virtual state provides user applications with a direct way to define the computation state of a thread. The computation state of a remote thread contains information necessary for the resumption of the execution after migration. For example, it can contain computation data, intermediate results, etc. The computation state varies according to the specific application, and it must be handled with the cooperation of the user application.

Users define a virtual state by extending the virtual state class. This inheritance gives both users and the VCluster system access to a virtual state, while separating them into two levels. Users can store application level information in a virtual state, and the VCluster system can store system level information. They have the choice to share or hide certain information from each other. In Java, only serializable objects can be transmitted over the network connection, so all information included in a virtual state must be stored in primary data type or serializable objects.

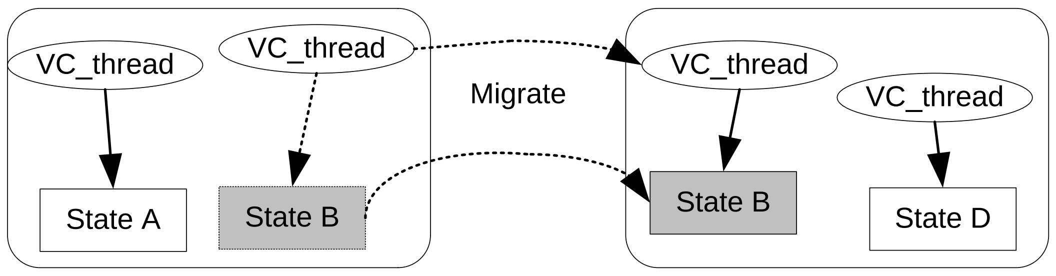


Figure 5: Thread Migration

The thread migration mechanism is illustrated in figure 5. To migrate a thread, the user thread is first notified to write the associated virtual state. After that, the user thread stops and is removed from the original node. The virtual state is then migrated to the destination node, where a new virtual thread is created based on the same class information and associated with the migrated state. This new thread reads the state, and continues the computation. To do this, the user thread must be coded to read the state before computation and start or continue the computation as desired. In thread migration, associated communication properties, e.g. the associated channels, are also copied and updated.

This approach is efficient as the system is only involved in the migration of the thread. Once a thread is migrated, it runs on its own just like normal threads and the system does not incur any overhead. Also, the application can run on any general JVM and with any general interpreters and compilers. However, this approach does require cooperation from the user applications and thus is not transparent to the user applications.

3.3 VCluster – Implementation

A prototype of VCluster was developed using 100% Java. The parallel run time system will run on any standard JVM, keeping full-fledged portability of Java. Details of the implementation of this run time system are described below.

3.3.1 Implementation of Global Architecture

The global architecture is implemented through an agent based architecture, as showed in Figure 6.

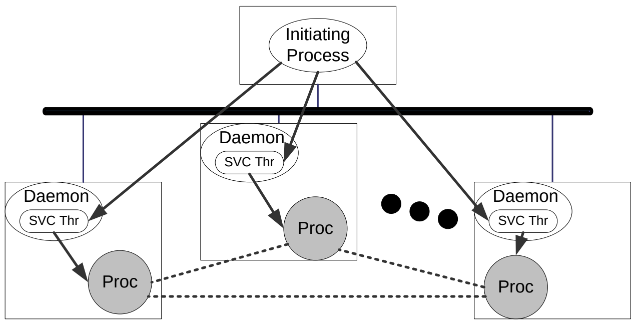


Figure 6: Distributed Run-Time System Architecture

In VCluster, a daemon process is running on every node in a cluster, and they talk through sockets. The user can start a parallel application from any node in the cluster (a daemon process is not necessary). Then the VCluster system started by this application will cooperate with the daemon processes to set up the distributed runtime environment. The daemon process manages process creation and termination and relays the standard output from the created processes to the initiating process. Daemon processes are multithreaded and can take multiple process creation requests concurrently. When a process creation request arrives at the daemon, a separate service thread is created that will actually spawn the requested process on the computer. This service thread replays the standard outputs to the initiating process and monitors the execution of the spawned process. When the spawned process is terminated, this service thread is destroyed as well. In VCluster, we are currently using a process group file to indicate the machines upon which processes are to be created. Once the requested processes are created, full connection is established among the processes that will be later used to create virtual channels among the threads.

MPICH [70] uses rsh [71] service provided by the operating system. Rsh service, however, has significant security vulnerabilities in that it does not perform secure authentication of the users and the communications are transmitted in plain text without encryption. SSH [72] service is a better choice in that it provides strong user authentication and communication encryption. Nevertheless, even SSH is used, the security is totally dependent on the SSH security mechanism, which has been facing challenges [73]. By using dedicated daemons, we can strengthen the overall structure of the distributed parallel run time system. Currently, we are not using any security services, but confidentiality and authentication will be later incorporated into the system. Also, dedicated daemons form an agent based architecture that delivers portability, expandability, flexibility and better resource management (Section 2.2.2).

3.3.2 Implementation of Local Architecture

The local system, a process serviced by a service thread, is a multithreaded system. Figure 7 shows the internal multithreaded structure of a process.

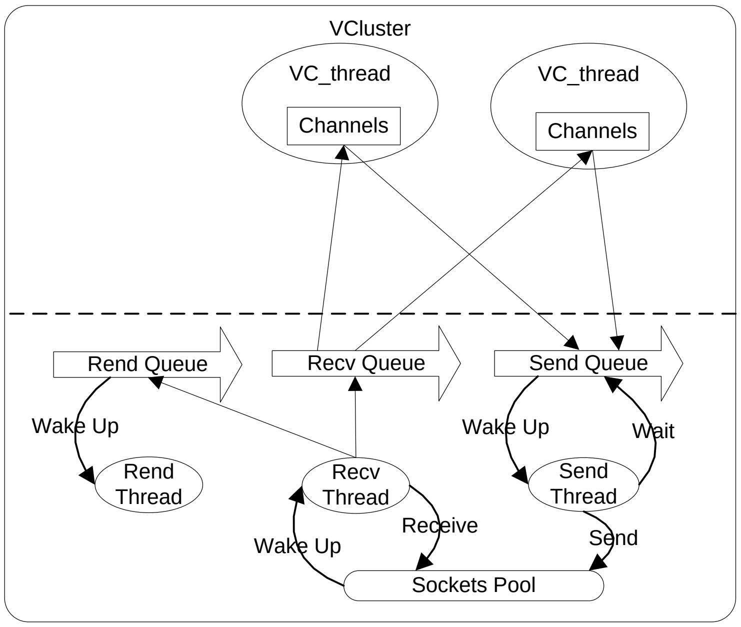


Figure 7: Internal Multithreading Structure of a Process

Each process consists of system threads that are not visible to the users, and user computing threads. User computing threads are created by user through extending the virtual thread class of VCluster, to represent a certain computation. System threads include rendezvous thread, send thread, and receive thread. Rendezvous thread is used to provide rendezvous style communication, a higher level communication mechanism for clients to issue remote procedure calls and for servers to accept them; In VCluster, thread migration takes place through this mechanism. Send and Receive threads maintain the associated message queues and perform send/receive operations on behalf of the virtual threads. Use of separate send and receive threads make I/O operations more efficient. They make the send and receive operation non-blocking, which allows the user computing threads to continue to run without being unnecessarily blocked. On SMP machines, user computing threads can keep running while send/recv threads are sending and receiving messages. Message requests can be processed and replied quickly. And it is possible that when a computation thread tries to receive a message, it is already in the receive message queue. It is shown in [74] that prompt responses of requests can improve performance and using separate send/recv threads yields better performance than interrupted communication on SMP machines, and slightly worse on uni-processor machines.

In our original implementation, the recv thread will be always running and polling for incoming messages. This approach turned out to decrease the performance seriously as the recv thread will compete with computation threads for CPU cycles, most time unnecessarily. It is desired that recv thread *sleeps* all the time and only *wake up* when there are messages coming. Java NIO [75], which is introduced in Sun JDK 1.4, gave us just what we needed. The Java NIO provides a new I/O mechanism through new classes called *channel* and *selector*. We replaced *ServerSocket* and *Socket* with *ServerSocketChannel* and *SocketChannel*. Then the recv thread uses a *selector* to monitor these channels. When there are no messages coming, the recv thread will *wait* on this *selector*, during which the recv thread is in a *sleep* state and will not compete for CPU cycles. Only when there are messages coming, the recv thread will be *woke up* by the *selector* and receive the message. The message is put into the receive message queue after received by the recv thread. After that, recv thread goes to *sleep* again.

Similarly, the send thread *sleeps* all the time an only *wake up* when there are outgoing messages. This can be easily implemented using the *wait* and *notify* keywords in Java. In Java, every object is a monitor and provides synchronization functions including *wait* and *notify*. Several threads can *wait* on an object until be *woke up*. Send Thread will *wait* on the send message queue, not using any CPU cycles at this time, until being *wake up* by outgoing messages. After sending out the messages, it goes to *sleep* again.

Communications between two threads can be either inter-node or intra-node. Figure 8 illustrates the intra-node communication between two threads. To send a message, the sender virtual thread calls the virtual communication channel with the data to be sent. The channel assembles a message with a copy of the data and other parameters, and puts the message into the receive queue. Since a copy of the data is put into the queue instead of the original one, the sender virtual thread can continue working on the data right after the call, without worrying about whether the receiver virtual thread has received the data. To receive a message, the receiver virtual thread also calls the virtual communication channel. The channel first checks whether the desired message is already in the receive queue. If it is, the channel returns the message to the virtual thread; if not, the channel causes the virtual thread to sleep on the queue, and be woken up when the message comes. Acting as a monitor, the queue also guarantees exclusive access to it and prevents race conditions.

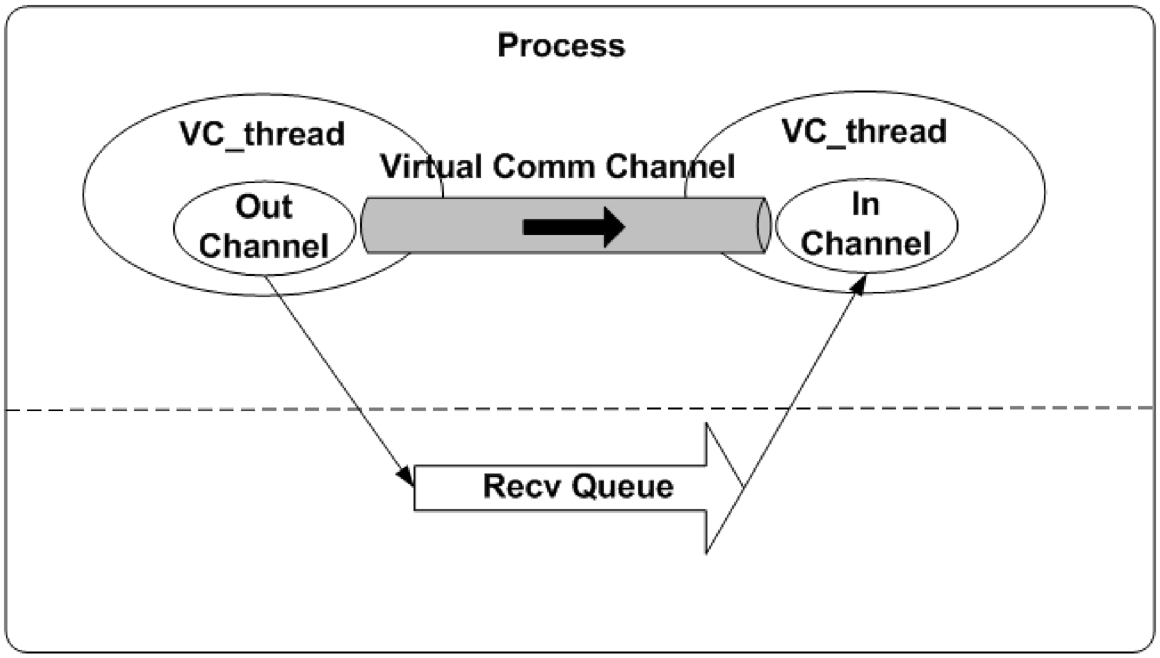


Figure 8: Intra-node Communication

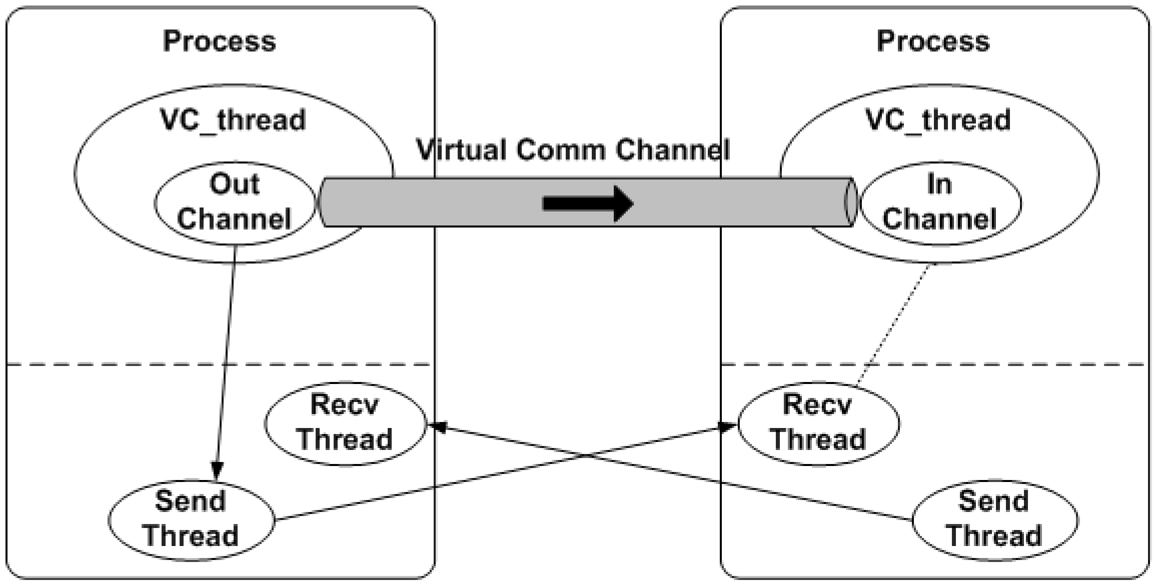


Figure 9: Inter-node Communication

Figure 9 illustrates the inter-node communication between two threads. To send a message, the sender virtual thread calls the virtual communication channel, which in turn assembles a message and puts it into the send message queue. Again, a copy of the data is included in the message, so that the sender virtual thread can continue working on the original data immediately, without worrying about whether the data has been sent out. The send message queue wakes up the send thread, which sends out the message through sockets.

When the message arrives at the destination node, the receive thread is woken up, which in turn receives the message and puts it into the receive message queue. If a computation thread is waiting on this message, it is woken up immediately.

To receive a message, the receiver virtual thread calls the virtual communication channel. This part of the communication is the same as in the intra-node communication.

3.3.3 The Initialization Process

To deploy a parallel application on a cluster, processes must be created on distributed nodes in the cluster, and inter communication channels must be created between every two processes, as showed in Figure 10. Outputs of all processes must be forwarded to the master process, where the user starts the application.

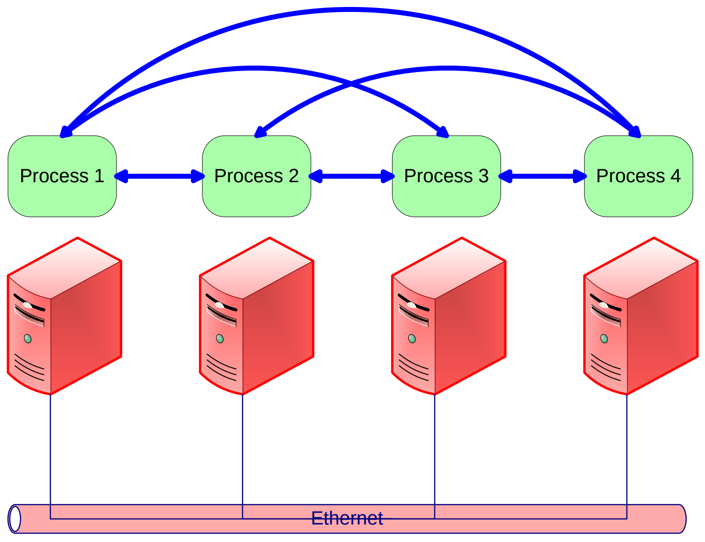


Figure 10: Parallel Application Architecture on Cluster

This could be a quite complex process. MPICH creates processes through rsh, using a “machinefile” to specify the nodes. JPVM creates processes through daemons running on every node, a similar approach to VCluster. However, its approach is much more complex to the user compared to the VCluster approach. JPVM provides an administrative console at the master node. To run an application, a user must first build up the parallel environment using this administrative console by inputting the information of every daemon, including the node address and the listening port. Then, in the parallel application, the user must code explicitly to spawn those processes. If arguments are used to run the application, the master process must transfer them to other processes through coding again. We want to simply this process, moving this complicate part from the user side into VCluster side. Currently, the user runs VCluster applications through a “procgroup” file. This “procgroup” files specifies the nodes to run a parallel application and other related information. Listing 1 shows the contents of a sample “procgroup” file. Every line specifies the information of a node. The first item is the name or IP address of a node; the second item is the number of processes to create on the node (currently not supported); following that are command line arguments to the ‘java’ command; the last item is the name of the main class to run.

```
#sample proc group file
sgenode01 1 -Xmx1000m VClusterApp.MD.AppMD
sgenode02 1 -Xmx1000m VClusterApp.MD.AppMD
sgenode03 1 -Xmx1000m VClusterApp.MD.AppMD
sgenode04 1 -Xmx1000m VClusterApp.MD.AppMD
```

Listing 1: A Sample ProcGroup File

Next, the initialization process of VCluster will be explained in detail. Figure 11 illustrates the initialization process in a UML sequence diagram.

Step 1: a user starts the application with a “procgroup” file. Let us call this node the master node, and the VCluster instance created by the user VCluster Master.

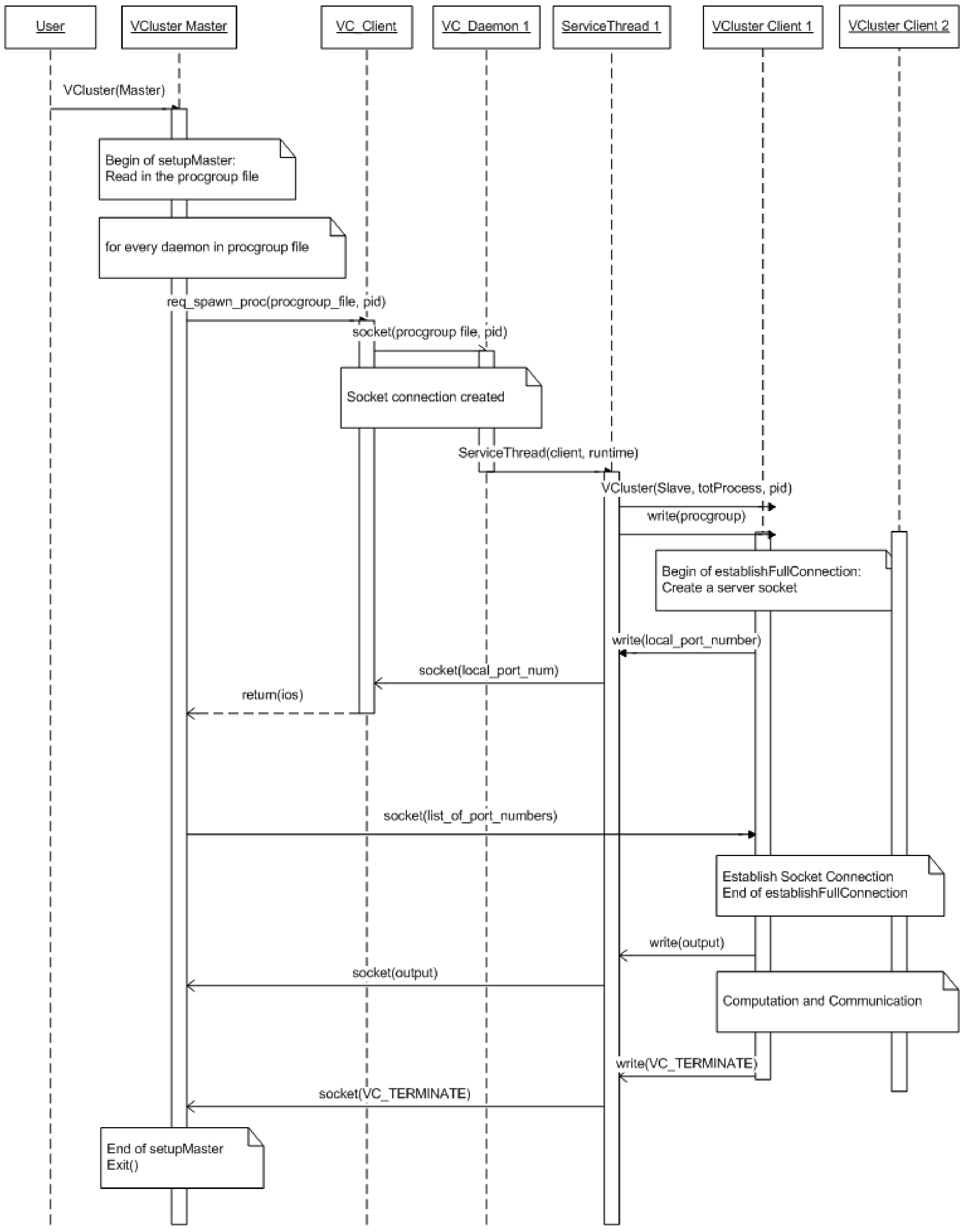

User
VCluster Master
VC_Client
VC_Daemon 1
ServiceThread 1
VCluster Client 1
VCluster Client 2
VCluster(Master)
Begin of setupMaster:
Read in the procgroup file
for every daemon in procgroup file
req_spawn_proc(procgroup_file, pid)
socket(procgroup file, pid)
Socket connection created
ServiceThread(client, runtime)
VCluster(Slave, totProcess, pid)
write(procgroup)
Begin of establishFullConnection:
Create a server socket
write(local_port_number)
socket(local_port_num)
return(ios)
socket(list_of_port_numbers)
Establish Socket Connection
End of establishFullConnection
write(output)
socket(output)
Computation and Communication
write(VC_TERMINATE)
socket(VC_TERMINATE)
End of setupMaster
Exit()

Figure 11: Initialization Process (Sequence Diagram)

Step 2: VCluster Master read in the “procgroup” file and spawn processes on client nodes. Daemons are located from the “procgroup” file and socket connections from VCluster master to these daemons are established. After that, a request is sent to every daemon. This request contains the contents of the “procgroup” file, and the process id assigned to the process that will be created by the daemon.

Step 3: daemons create service threads and VCluster processes. After receiving the request, a daemon creates a service thread. The service thread then creates a VCluster instance (VCluster Client) according to the specifications in the “procgroup” file with a Java Runtime. In this way, the printout of the VCluster Client will be read by this service thread, which in turn forward to the master node and printed out there to the user. Also, this communication channel between the service thread and the VCluster Client allows the service thread to pass the “procgroup” file to the VCluster Client, and the VCluster Client to pass the information of the server sockets it created to the service thread. This socket information will be again sent to the master node. Till this step, a VCluster process is created on every node in the “procgroup” file, but they can not communicate with each other as the connections between these processes are not established.

Step 4: connections between every two VCluster Clients are established. At this time, only VCluster Master has the socket information of every VCluster Client. This information is sent to the service threads, and then to the VCluster Clients. Using this information, the VCluster Clients can establish connections between each other. This step could be simplified if VCluster Clients all create sockets with a fixed port number. The problem is that in such a way, only one VCluster Client can be created on every node, and the port number might not be free on every node. By allowing more than one VCluster Clients on a single node, we can run several parallel applications on one cluster at the same time, e.g. fault tolerance applications or other not performance critical applications.

Step 5: computation are carried out by the VCluster Clients. Computations and communications are carried out by the VCluster Clients. Printouts of VCluster Clients are relayed by service threads to VCluster master, being printed out to the user consequentially.

Step 6: application is terminated. After a VCluster client finished, the corresponding service thread notifies VCluster Master the termination of this VCluster Client and terminates itself. After all clients have finished, VCluster Master terminates. We can see that during this process, VCluster Master does not carry out the computation.

3.3.4 Thread Migration

As discussed previously, to migrate a thread in VCluster, VCluster migrates the user defined computation state of the thread. After that, a new thread is created on the destination node and is associated with the migrated state. Users define the computation state of an application by extending the *VC_State* class. For example, the user has a very simple application that computes the sum of an integer array. In this application, the user thread contains a loop that sums up the array items one by one. Now, suppose the user wants to migrate a thread within the loop. To continue the loop after migration, the user needs the array, the intermediate sum, and the index of the last item added to the sum. The computation state of this thread can be defined as shown in listing 2.

```
Public class sumState extends VC_State
{
    public int[] aiArray; //the array
    public int iInterSum; //the intermediate sum
    public int iIndex; //the index of the last item added
}
```

Listing 2: An Example Virtual State Class

The *VC_State* class is implemented in VCluster. It currently only contains one variable, a flag to identify whether a state has migrated. This inheritance relationship allows users and VCluster to define a virtual state together while being separated into two levels.

Thread migration can be triggered either by user applications or by VCluster. We are currently working on the system triggered thread migration. It is still an ongoing work, so this paper only discusses user-triggered thread migration. The previous example is used. When the user application decides to migrate a thread, the thread writes its state to the associated virtual state, calls the *migrate()* function, which is implemented in the parent *VC_Thread* class, and stops the loop. The *migrate()* function does the following. First, the virtual state is marked as migrated so that the user application will resume execution at the destination node instead of start over. Second, the state, the channels, and the class names of the user thread and state are packed up as a message and put into the send message queue, which is in turn sent by the send thread. The receive thread on the destination thread receives this message and determines it is a system message. System messages are put into the rendezvous message queue and the rendezvous thread is woken up to process this message. The rendezvous thread creates a new user thread, associates it with the virtual states and channels extracted from the message, and let it run. The user thread must be coded to read the virtual state first and decide whether to start the loop from beginning or continue the loop. In this case, it should continue. With the information in the state, it is easy to continue the loop. Channels between this thread and other threads must also be updated. Currently, this has to be done by user applications. We are currently implementing the idea of thread group. Thread group will manage channels between group members. It will create channels between threads, and automatically update channels after thread migration.

3.3.5 Object Serialization

Object serialization is one of the great features of Java. While conventional message passing libraries only support transmissions of arrays, object serialization enables transmission of objects. Compared to arrays, objects are more structured and can contain much more information than plain data. Any objects that implements the *Serializable* interface can be serialized, transmitted through the network to a destination node, and rebuilt through de-serialization.

Java provides *ObjectOutputStream* and *ObjectInputStream* to serialize and de-serialize objects. But as we use non-blocking *SocketChannels* in Java NIO, which is block-oriented instead of stream-oriented, this is not feasible. Objects must first be serialized to a stream, and then converted to a *ByteBuffer* which can be used by non-blocking *SocketChannel*. Listing 3 shows the code for sending and receiving objects through *SocketChannel*.

```
    //sending objects through SocketChannel
    baos = new ByteArrayOutputStream();
    oos = new ObjectOutputStream(baos);
    oos.writeObject(obj);         //convert object to stream, serialization
    ByteBuffer bfObj =
       ByteBuffer.wrap(baos.toByteArray());         //convert to ByteBuffer
    while (nBytes < bfObj.capacity())
       nBytes += SocketChannel.write(bfObj);       //write to SocketChannel


    //receiving objects through SocketChannel
    ByteBuffer bfObj = ByteBuffer.allocate(objLength);
    while (bfObj.remaining() > 0)
       SocketChannel.read(bfObj);                 //read from SocketChannel
    ObjectInputStream ois = new ObjectInputStream(new ByteArrayInputStream(
       bfObj.array()));                      //convert ByteBuffer to stream
    obj = ois.readObject();            //read from stream, de-serialization
```

Listing 3: Transmitting Object through SocketChannel

3.4 Conclusion

In this chapter, we discussed the architecture and implementation of VCluster. VCluster is a thread based parallel application model that support shared memory and message passing in a same framework. In the following chapter, we discuss how to develop parallel applications using VCluster.

CHAPTER FOUR: USE OF VCLUSTER

In this chapter, we describe how to write parallel applications on top of VCluster. The object-oriented paradigm of Java lets us hide the complexity within VCluster, and users write parallel applications through extending the virtual thread and virtual state classes.

4.1 An Example Application

Users develop parallel applications by extending the VC_Thread class and VC_State class. A user application normally contains three classes. The computation will be in the virtual thread class. The virtual state is used for thread migration and is not necessary if thread migration is not considered. Another class, the main class, is used to create and set up virtual threads, virtual states, and run them. Next we explain this with an example application.

This simple example application runs on two nodes, one thread running on each node. Thread 0 is the sender and thread 1 is the receiver. Thread 1 receives the messages and prints them out.

The main class contains the *main()* function, and is the class that is run by the user. Basically the task of main class is to set up the execution environment; create virtual threads, virtual states and the communication channels between threads. Listing 4 shows part of the code in this class. Code section 1 creates a virtual thread and adds it to VCluster. Line 4 and line 5 create a virtual state and associate it with the virtual thread. Section 2 creates a send channel on thread 0 and a receive channel on thread 1. A channel must be within a channel set. Send and receive channels usually belong to two separate channel sets, but are not required to. The send channel and the matching receive channel must have the same name.

The virtual state class contains computation data and/or methods. Users implement their application specific states through extending the virtual state class in VCluster. In this example, this class only contains one integer member, x. While it is not visible to the users, virtual state class provides inherited methods for state transfer necessary for thread migration.

```
1.1      // Create a VC_Thread
1.2      vthr = new applThread();
1.3      // Associate a state with this thread
1.4      vState = new applState();
1.5      vthr.addVCState(vState);
1.6      //add this thread to VCluster
1.7      vc.addThread(vthr);

2.1      // Setup the communications (channelSets and channels)
2.2     if (vc.getMyPid() == 0) {
2.3        vchannelSet = new VC_ChannelSet("sendChannelSet");
2.4        vchannel = new VC_Channel(1,"ch", vc.VC_WRITE);
2.5     }
2.6     else {
2.7        vchannelSet = new VC_ChannelSet("recvChannelSet");
2.8        vchannel = new VC_Channel(0,"ch", vc.VC_READ);
2.9     }
2.10    vchannelSet.addChannel(vchannel);
2.11    vthr.addChannelSet(vchannelSet);
```

Listing 4: Example Application, the Main Class

```
public class applState extends VC_State
{
   public int x;

   public applState ()
   {
      x = 999;
   }
}
```

Listing 5: Example Application, the State Class

The most important class is the virtual thread class, which carries out all the computation. Listing 6 shows part of the code of this class. Code section 1 shows how to get the process id and thread id, and how to print out messages; code section 2 shows how to access state; code section 3 shows how to send messages; code section 4 shows how to read messages.

```
1.1      //get my process id and thread id
1.2      myPid = getPid();
1.3      myTid = getTid();
1.4      VC_Println("** Hello World : pid = "+myPid+", tid = "+myTid);


2.1      //access state
2.2      state = (applState)getState();
2.3      VC_Println("** x = "+state.x);

3.1      //retrieve channel
3.2      vchannelSet = getChannelSet("sendChannelSet");
3.3      VC_Channel vchannel = vchannelSet.getChannel("ch");
3.4      //send messages
3.5      vchannel.writeInt(888);
3.6      vchannel.writeFloat((float)777.0);
3.7      vchannel.writeDouble(666.0);
3.8      int arrInt[] = {1,2,3,4};
3.9      vchannel.writeObject(arrInt);
3.10     double arrD[] = {1.0,2.0,3.0,4.0};
3.11     vchannel.writeObject(arrD);

4.1      vchannelSet = getChannelSet("recvChannelSet");
4.2      VC_Channel vchannel = vchannelSet.getChannel("ch");
4.3      //receive messages
4.4      int    a = vchannel.readInt();
4.5      float  b = vchannel.readFloat();
4.6      double c = vchannel.readDouble();
4.7      VC_Println("** recved a = "+a);
4.8      VC_Println("** recved b = "+b);
4.9      VC_Println("** recved c = "+c);
4.10     int arrInt[];
4.11     arrInt = (int []) vchannel.readObject();
4.12     double arrD[];
4.13     arrD = (double []) vchannel.readObject();
```

Listing 6: Example Application, the Thread Class

Listing 6 shows a “procgroup” file for this example application.

```
sgenode01 1 VClusterApp.Example.Appl
sgenode02 1 VClusterApp.Example.Appl
```

Listing 7: Example Application, the ProcGroup file

Then the user executes the following command for compilation and execution:

```
javac –d . Appl.java, applState.java, applThread.java
java VClusterApp.Example.Appl procgroup2
```

The complete listing of this sample application can be found in Appendix A.

4.2 A Multithreading Application

In the previous section, only one thread is created on every node. To use SMP machines efficiently, we explain how to create multiple threads on a single node and how to deal with the communication between these threads in VCluster. We first introduce the single thread version, in which only one thread is created on every node. Then we show how to extend this single thread version to support multithreading.

BPNN [77] is one of the most popular neural network training algorithms and has shown robust performance in many diverse applications. However, computational complexity of the BPNN makes its use challenging especially when the training data set size is huge. BPNN can be parallelized in many different ways depending on the underlying parallel computing architecture and programming models [78, 79]. We use a training set partition method that reduces the amount of communication needed for distributing data and synchronization. BPNN is trained iteratively until an acceptable mean squared error rate is achieved. In the beginning, the partitions of the data set are distributed to each worker thread. Each iteration is called an epoch, in which each worker thread works on the fraction of the computation corresponding to the distributed partition and the master thread aggregates the partial computation results. Then, the updated weight vectors are distributed to all the worker threads for the next epoch.

Implementation of this algorithm in VCluster is composed of two classes, the main class and the thread class. Note that there is no state class used in this example since issues on thread migration are not discussed in this paper.

Listing 8 shows the code of the main class. Only one thread is created on every node. Thread 0 is the master, so a send channel and a receive channel are created from this thread to all other threads. The other threads simply create a send channel and a receive channel to thread 0. The real computation is carried out in the thread class, which is shown in listing 9. The computation part here is the same in both VCluster and MPICH. In the communication part, thread 0 will collect new weight vectors from all other threads, average it, compute the new weight vectors, and distribute the new weight vectors to all other threads. The code in listing 9 shows how this communication part is implemented in VCluster. Every thread has a set of weight vectors. As Java is object-oriented, this set of vectors can be packaged into a class, which is called BPNNWeights. Objects of this class, instead of sets of vectors, can be transferred between threads to simplify the communication. Thread 0 first enters a loop to receive new weight vectors, and all other threads send its new weight vector. The send and receive are carried out by retrieving the corresponding channel from the channel set, and then calling write or read method of the channel. After all weight vectors are collected, new weight vectors are calculated by calling functions of the BPNNWeights class. Then the new weight vectors are distributed to all other threads.

```
public class AppBPNN {
   public static void main(String args[])
   {
      ...

      vc = new VCluster(args);
      myPid = vc.getMyPid();
      nprocs = vc.getTotProcs();

      // Create VC_Threads
      myThr vthr = new myThr();
      vc.addThread(vthr);

      // Setup the communications (channelSets and channels)
      //channelsets
      vSendSet = new VC_ChannelSet("sendChannelSet");
      vRecvSet = new VC_ChannelSet("recvChannelSet");
      //channel from all the other threads to thread 0
      if (myPid == 0)
      {
       for (i=1;i<nprocs;i++)
       {
          vchannelr = new VC_Channel(i, WEIGHT_FROM+i, vc.VC_READ);
          vRecvSet.addChannel(vchannelr);
          vchannels = new VC_Channel(i, WEIGHT_TO+i, vc.VC_WRITE);
          vSendSet.addChannel(vchannels);
       }
      } else {
         vchannels = new VC_Channel(0,WEIGHT_FROM+myPid, vc.VC_WRITE);
         vSendSet.addChannel(vchannels);
         vchannelr = new VC_Channel(0, WEIGHT_TO+myPid, vc.VC_READ);
         vRecvSet.addChannel(vchannelr);
      }
      //add channel set to thread
      vthr.addChannelSet(vSendSet);
      vthr.addChannelSet(vRecvSet);

      // Start the computation
      vc.start();
      vc.finalize();
   }
}
```

Listing 8: Main Class of BPNN in VCluster without Multithreading

```
public class myThr extends VC_Thread {
   //constants
   private final int outputNode = 1;
   ...

   //class variables
   private SampleRecord[] sampleRecords;
   ...

   public void run() {
      ...

      if (myPid != 0) {
         vSendSet.getChannel(WEIGHT_FROM+myPid)
            .writeObject(newBPNNWeights);
      } else {
         summaryBPNNWeights = new BPNNWeights(inputNode, hiddenNode,
outputNode);
         summaryBPNNWeights.add(newBPNNWeights);
         for (i=1;i<nprocs;i++) {
            readBPNNWeights = (BPNNWeights)vRecvSet.getChannel
               (WEIGHT_FROM+i).readObject();
            summaryBPNNWeights.add(readBPNNWeights);
         }
         summaryBPNNWeights.mean(nprocs);
         summaryBPNNWeights.add(bpnnWeights);
         newBPNNWeights = summaryBPNNWeights;
      }
      //thread 0 send bpnnWeights to all other threads
      if (myPid != 0) {
         newBPNNWeights = (BPNNWeights)vRecvSet.getChannel
         (WEIGHT_TO+myPid).readObject();
      } else {
         for (i=1;i<nprocs;i++) {
            vSendSet.getChannel(WEIGHT_TO+i)
               .writeObject(newBPNNWeights);
      }

      ...

      finalize();
   }
```

Listing 9: Thread Class of BPNN in VCluster without Multithreading

To support multithreading, we need to create more threads. We added a loop to create more threads, and a loop to create channels to all threads on a node instead of just one thread.

```
    //set up the threads
    for (myTid=0;myTid<Thread_Per_Node;myTid++) {
       // Create VC_Threads
       vthr = new myThr();
       vc.addThread(vthr);

       // Setup the communications (channelSets and channels)
       //channelsets
       vSendSet = new VC_ChannelSet("sendChannelSet");
       vRecvSet = new VC_ChannelSet("recvChannelSet");

       if ((myPid == 0) && (myTid == 0)) {
          //establish channels to other threads
          for (i=0;i<nprocs;i++) {
             for (j=0;j<Thread_Per_Node;j++) {
                if ((i != 0) || (j != 0)) {
                   vchannelr = new VC_Channel(i, WEIGHT_FROM+i+j,
                                              vc.VC_READ);
                   vRecvSet.addChannel(vchannelr);
                   vchannels = new VC_Channel(i, WEIGHT_TO+i+j,
                                              vc.VC_WRITE);
                   vSendSet.addChannel(vchannels);
                }
             }
          }
       } else {
          //establish channel to thread 0 on node 0
          vchannels = new VC_Channel(0, WEIGHT_FROM+myPid+myTid,
                                     vc.VC_WRITE);
          vSendSet.addChannel(vchannels);
          vchannelr = new VC_Channel(0, WEIGHT_TO+myPid+myTid,
                                     vc.VC_READ);
          vRecvSet.addChannel(vchannelr);
       }

       //add channel set to thread
       vthr.addChannelSet(vSendSet);
       vthr.addChannelSet(vRecvSet);
    }
```

Listing 10: Main Class of BPNN in VCluster with Multithreading

Changes to the thread class are similar. The major change is the loop on thread 0 to communicate with all threads on a node instead of just one thread.

```
if ((myPid != 0) || (myTid != 0)) {
   vSendSet.getChannel(WEIGHT_FROM+myPid+myTid)
      .writeObject(newBPNNWeights);
} else {
   summaryBPNNWeights = new BPNNWeights(inputNode, hiddenNode, outputNode);
   summaryBPNNWeights.add(newBPNNWeights);
   for (i=0;i<nprocs;i++) {
      for (j=0;j<Thread_Per_Node;j++) {
         if ((i != 0) || (j != 0)) {
            readBPNNWeights = (BPNNWeights)vRecvSet
               .getChannel(WEIGHT_FROM+i+j).readObject();
            summaryBPNNWeights.add(readBPNNWeights);
         }
      }
   }
   summaryBPNNWeights.mean(nthrds);
   summaryBPNNWeights.add(bpnnWeights);
   newBPNNWeights = summaryBPNNWeights;
}
//thread 0 send bpnnWeights to all other threads
if ((myPid != 0) || (myTid != 0)) {
   newBPNNWeights = (BPNNWeights)vRecvSet
      .getChannel(WEIGHT_TO+myPid+myTid).readObject();
} else {
   for (i=0;i<nprocs;i++) {
      for (j=0;j<Thread_Per_Node;j++) {
         if ((i != 0) || (j != 0)) {
            vSendSet.getChannel(WEIGHT_TO+i+j)
               .writeObject(newBPNNWeights);
         }
      }
   }
}
```

Listing 11: Thread Class of BPNN in VCluster with Multithreading

4.3 Conclusion

In this chapter, we describe how to develop parallel applications using VCluster, with or without multithreading. Detailed documentation of classes and their functions in VCluster are in Appendix B.

CHAPTER FIVE: PERFORMANCE, EXPERIMENTS AND COMPARISON

In this chapter, we examine the performance of VCluster and compare it with other relevant systems. Three relevant systems were compared with VCluster: MPICH, mpiJava, and JPVM. To support multithreading, MPICH was combined with POSIX Thread and openMP. MPICH is a C implementation of MPI by Argonne National Laboratory [70], mpiJava by [31], and JPVM by [35]. MPICH was chosen as the base case that extends the C programming language with message passing functions. It is most widely used and would yield the best performance compared with other Java based libraries. mpiJava is simply a wrapper of MPICH with Java interface and is expected to be slower than MPICH but faster than JPVM and VCluster, which are implemented purely in JAVA. Our performance analysis was done in three aspects. First we measured the point-to-point communication overhead. Second, we investigated the real application performance without multithreading for several exemplary parallel algorithms that includes both communication and computation. Lastly, we explored the application performance with multithreading.

5.1 Communication Overhead

The compare the communication overhead of different systems, we use the popular ping-pong benchmark. There are two processes running on two nodes, one is the sender and the other is the receiver. The sender will send a message to the receiver, and the receiver will send the same message back right after receives the message. This roundtrip time is calculated by sending the message a number of times and taking average.

Since JAVA uses the neural byte format for the communication, the communication overhead was expected to be higher than that of C. Figure 12 shows the round trip communication overhead with respect to the varying message size from one byte to one mega bytes. As expected, Java based message passing libraries are slower than C based MPICH. For a larger message size closer to one mega byte, VCluster performed better than the JPVM. We believe that this is because of the VCluster’s sending and receiving mechanisms based on separate send/recv threads. Despite the higher communication overhead in VCluster and other Java based systems, it does not discourage the use of Java for cluster computing in that coarse grain parallelism is mostly used in cluster computing and the communication overhead can be compensated by the larger computation part of the application. In addition, when computation becomes the major part, the send/recv thread in VCluster can transmit messages while the computation thread is working on computation. The pure communication time, in which the computation thread is idly waiting for messages, can be greatly reduced.

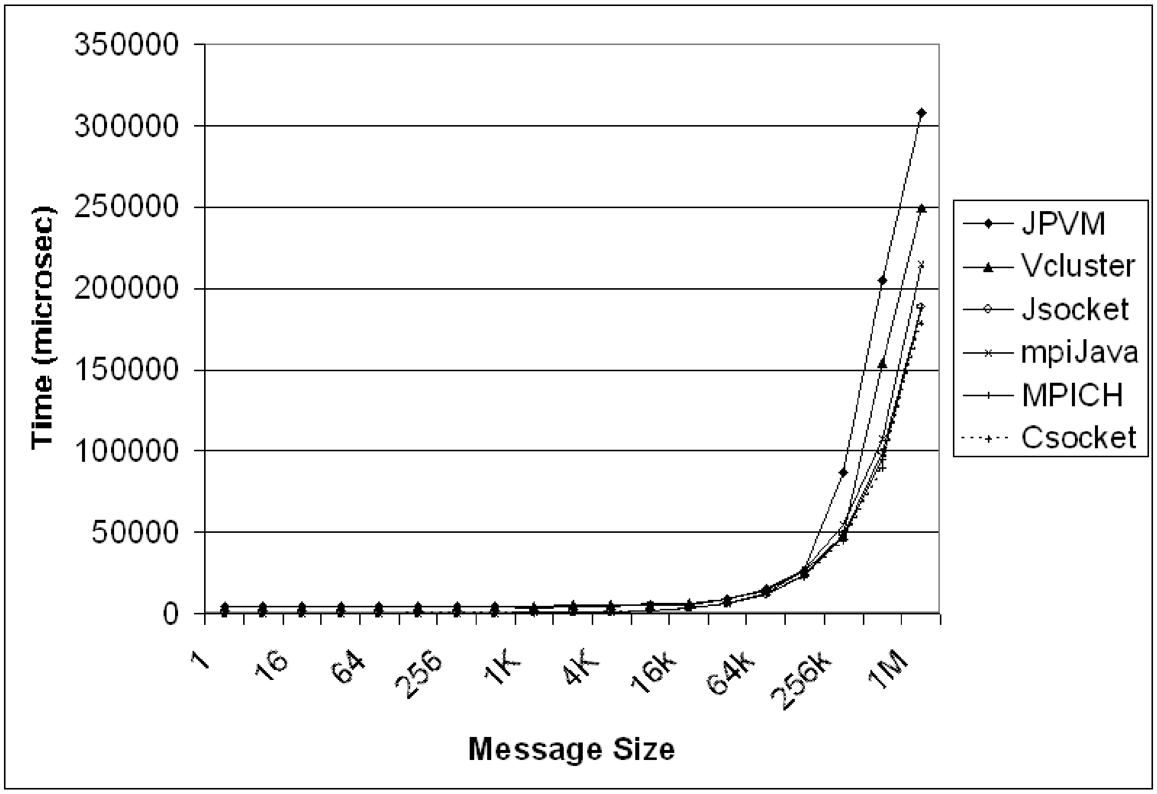


Figure 12: Communication Overhead

5.2 Experiments without Multithreading

In this section, we compare the performance of VCluster with MPICH, mpiJava and jPVM without using multithreading. VCluster will only create one thread on every node, and the other three libraries create one process on every node, as how they are usually used. The experiment was conducted on the Scerola cluster, which has 64 LINUX PCs equipped with 900MHz AMD Athlon processor, 1 GB memory, and 100BT networking.

5.2.1 Parallel Dirichlet Problem

The first application is a fluid dynamics application, Dirichlet boundary problem. The Dirichlet boundary problem is a simple numerical simulation problem on a two dimensional grid. Each point on the grid has a (x, y) location and a value representing the temperature of some material. At each time step, each point's temperature is averaged with its neighbors' temperatures to find the point's temperature at the end of the time step. This operates on all grid points that are not on the boundary. Boundary grid points are assumed to have a constant value. The workload is uniform in the Dirichlet problem. This allows the domain decomposition technique to be used in dividing up the workload among processes. We divided the grid by columns, as showed in figure 13. The communication will be between neighboring processes,

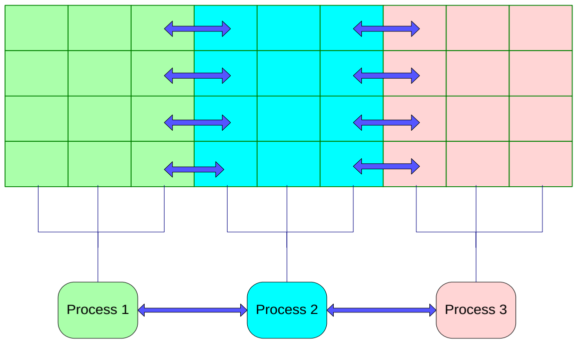


Figure 13: Illustration of Parallel Dirichlet Problem

Figure 14 shows their performance on this cluster with varying numbers of processors. Performance of three Java based message passing libraries was comparable and the MPICH outperformed them. With a small number of processors, the execution time difference was noticeable. However, with a larger number of processors, for example 32, the difference became ignorable. This results of the experiments indicate that with enough number of processors Java based message passing libraries can mitigate the associated overhead compared with C based libraries and be an attractive tool for a large scale parallel application development. mpiJava was expected to be faster than VCluster and JPVM as it still uses C codes in it. However, its performance was close to the other Java implemented libraries.

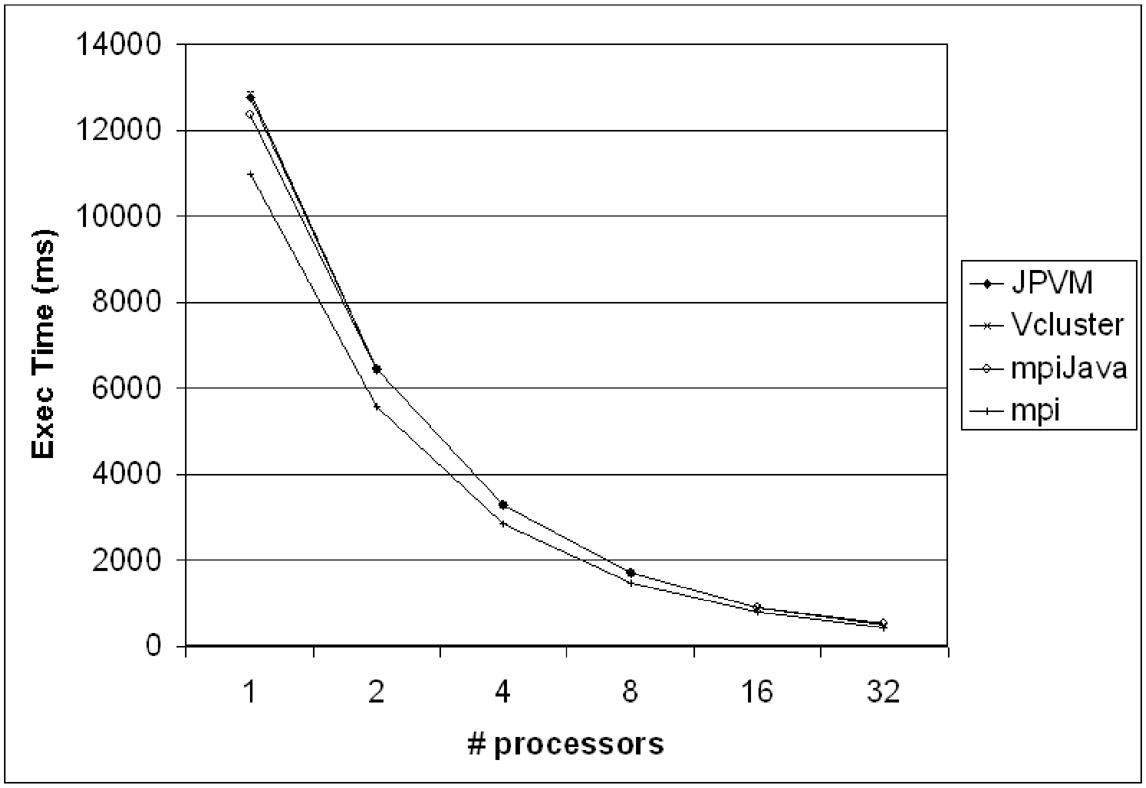


Figure 14: Dirichlet Problem without Multithreading

5.2.2 Parallel Back-propagation Neural Network

The second application was a parallel Back Propagation Neural Network (BPNN) algorithm for network based intrusion detection. BPNN [76] is one of the most popular neural network training algorithms and has shown robust performance in many diverse applications.

However, computational complexity of the BPNN makes its use challenging especially when the training data set size is huge. BPNN can be parallelized in many different ways depending on the underlying parallel computing architecture and programming models [77, 78]. We used a training set partitioning method that reduces the amount of communication needed for distributing data and synchronization. BPNN is trained iteratively until an acceptable mean squared error rate is achieved. In the beginning, the partitions of the data set are distributed to each worker thread. Each iteration is called an epoch. In each epoch, each worker thread works on the fraction of the computation corresponding to the distributed partition and the master thread aggregates the partial computation results. Then, the updated weight vectors are redistributed to all the worker threads for the next epoch.

Figure 15 shows the performance of the parallel PBNN on the same cluster. Once again, the three Java based libraries showed comparable performance. Note that the gap between VCluster and MPICH converges as the number of processors increases. The required amount of computation is much higher than that of the Dirichlet boundary problem and it is expected that with use of more processors the gap will further converge as the previous application did.

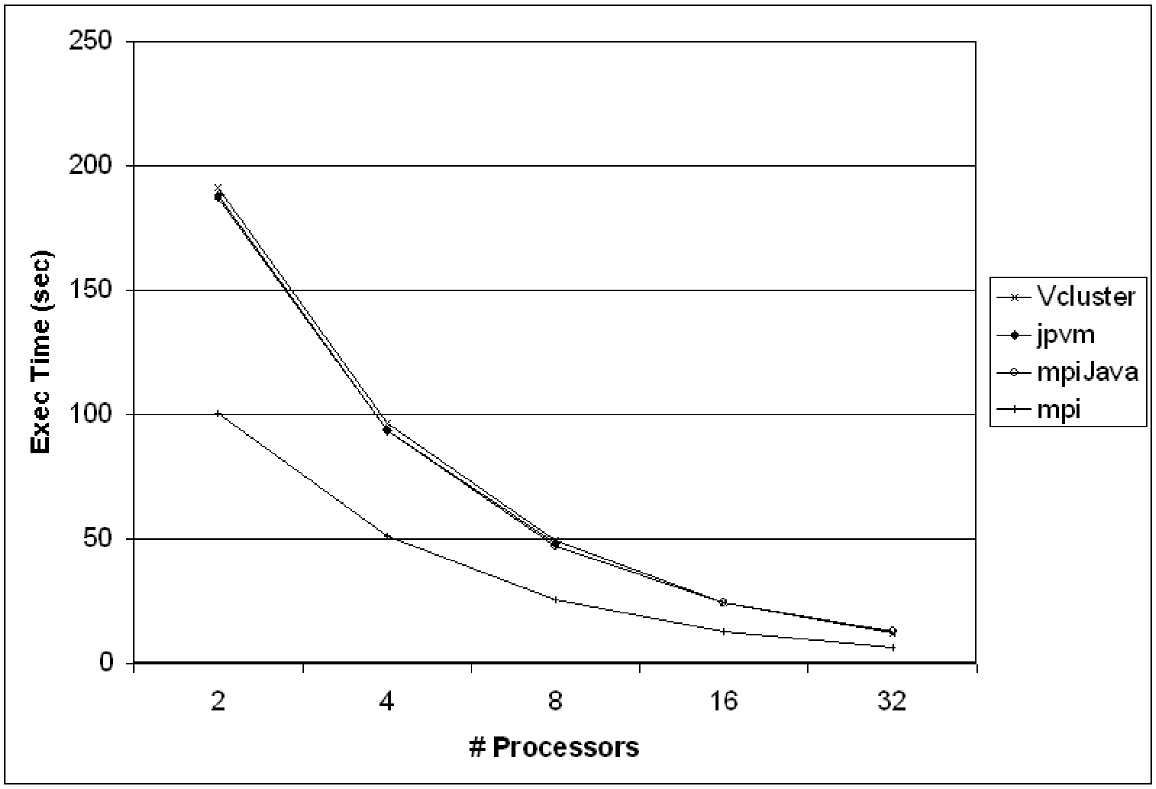


Figure 15: Parallel Neural Network without Multithreading

5.3 Experiments with Multithreading

Previous experiments were conducted on a cluster of uni-processor computers. While VCluster can support a cluster of uni-processor computers, it can also be used with a cluster of multiprocessors using the same programming model without causing significant changes to the parallel program. The experiments are on a clusterof 32 dual processor computers. Each computer is equipped with dual AMD Opteron 242 1.6GHz processors, 2 GB RAM, and a Gigabit Ethernet. The operating system is SunOS 5.9. The two problems used are again parallel dirichlet problem and parallel back propagation neural network problem.

5.3.1 Coding with VCluster and Other Libraries

We revised the MPICH program with POSIX thread and openMP to support multithreading, and mpiJava and JPVM programs were revised using Java threads. During this process, we found VCluster the easiest one to change. We have shown how to write multithreading applications in VCluster in the previous chapter. Next we will discuss the implementation of multithreading applications with MPICH, mpiJava and JPVM.

Multithreading with MPICH + PThread

Listing 12 shows the basic architecture of a MPICH + PThread program. *Mutex* and *Conditioners* are used for thread synchronization. And the communication between threads on the same node must be handled by the user. A thread is explicitly created on a function, “*thr()” as in Listing 12. Thread ID can be passed to the thread at this time. After that, the main thread will be waiting for all thread to finish, only at that time should the *MPI_Finalize* function been called.

The *thr* function shows how intra-node communication and inter-node communication should be carried out. Functions, *setMsg* and *readMsg* are written for intra-node communication. These functions must be protected by *mutex* to ensure exclusive entrance. To write a message, we must first make sure the last message has been read, otherwise it will be overwritten. This can be done through a *conditioner*. After that, the *new message flag* is set. To read a message, we must first make sure a new message has been written, otherwise the same message will be read again. After that, the *new message flag* is cleared. For different messages, specific functions should be written and specific *flags* should be used.

Inter-node communication is still through MPICH functions. Since MPICH is not thread-safe, calls to MPICH functions must be exclusive. One way to do this is through a *mutex* , as showed in Listing 7; another way is only allow one thread to call MPICH functions, other local threads will send their data to this thread if they want to send through MPICH. Another problem in inter-node communication is that as the message passing functions in MPICH are defined between processes, users must program carefully to make sure the message is sent to or received from the right thread on the other node. This can be done through tags, assigning a special tag for every thread. Collective communications are far more complicate. As only one thread can attend the collective communication, a specific thread must be selected as a representative. This thread needs to collect data from other local threads through intra-node communication, call MPICH collective function to get global data, and distribute the new data to all local threads.

These problems will not be present in VCluster. Intra-node communication and inter-node communication will be handled in the same way, through virtual channels. Collective communications between threads are also supported, which is easy to implement in VCluster as communications are defined between threads.

Another problem we encountered in revising a single thread MPICH program to support multithreading is about the global variables and functions. To develop a program, functions are often necessary and will make the program architecture more structural. Global variables are used to share data between these functions, and there is only one copy of every variable. However, when multithreading is incorporated, these global variables must be duplicated so that every thread can operate on a copy of the variables. And since functions can be called by different threads, they must determine by which thread they are called so that they can operate on the right copy of global variables.

Again this is not a problem for VCluster, thanks to Java's object-oriented architecture. Every thread is an instance of a class, and a copy of the class global variables will be automatically created for every thread. The functions do not need to changed, as Java will make sure they operate on the right copy of global variables.

```
//compile on ariel cluster, which is running SUN OS5.9
mpicc -O1 -o hp hp.c -D_POSIX_C_SOURCE=199506 -lm -lrt -lpthread

//compile on scerola cluster, which is running Debian with kernel 2.4.18
//debian is a popular linux distribution
mpicc -O1 -o hp hp.c -lpthread
```

A MPICH + PThread program can still be compiled using *mpicc*. Nevertheless, as the implementation of PThread library is highly dependent on the underlying operating system, this compiling process can be highly different, as illustrated

above.

```
#include "mpi.h"
#include <pthread.h>

//mutex and conditioner are used for thread synchronization
pthread_mutex_t mutex_mpi = PTHREAD_MUTEX_INITIALIZER; //exclusive call to
mpi functions
pthread_mutex_t mutex_cond = PTHREAD_MUTEX_INITIALIZER; //mutex for
condition
pthread_cond_t cond_msg = PTHREAD_COND_INITIALIZER; //gurantee the order of
read and write
float *msg[Thread_Per_Node]; //storage of messages
int fNewMsg[Thread_Per_Node]; //flags for new messages

/**************************************************************************
***/
int main(int argc, char *argv[])
{
   pthread_t threads[Thread_Per_Node]; //an array of threads
   int *tids[Thread_Per_Node]; //thread ids to pass to the threads

   //MPI_Init(&argc,&argv);
   MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &thread_level);
   MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myPid);

   //create threads
   for (i=0;i<Thread_Per_Node;i++) {
      *tids[i] = i; //set thread id
      //create thread
      pthread_create(&threads[i], NULL, thr, (void *) tids[i]);
   }

   //wait until all threads end
   for (i=0;i<Thread_Per_Node;i++) {
      pthread_join(threads[i], NULL);
   }

   MPI_Finalize();
}

//thread function
void *thr(void *ptr) {
   //get thread id
   myTid = *((int *)ptr);

   //communicate with threads on the same node
   setMsg(myTid-1,msgr); //send
   readMsg(myTid, revr); //receive
```

Listing 12: Multithreading with MPICH + PThread

```
   //communicate with threads on other nodes
   pthread_mutex_lock(&mutex_mpi); //exclusive call to MPI functions
   {
      MPI_Send(...);
      MPI_Recv(...);
   }
   pthread_mutex_unlock(&mutex_mpi);
}

//communicate between threads on the same node
void setMsg(int destTid, float *p_msg) {
   pthread_mutex_lock(&mutex_cond);
   {
      //make sure the last one has been read
      while (fNewMsg[destTid]) pthread_cond_wait(&cond_msg, &mutex_cond);
      //copy p_msg to msg
      for (i=0; i<N; i++) msg[destTid][i] = p_msg[i];
      //set new message flag
      fNewMsg[destTid] = 1;
      //someone might be waiting, wake them up
      pthread_cond_broadcast(&cond_msg);
   }
   pthread_mutex_unlock(&mutex_cond);
}

void readMsg(int myTid, float *p_msg) {
   pthread_mutex_lock(&mutex_cond);
   {
      //make sure new values have been written
      while (!fNewMsg[myTid]) pthread_cond_wait(&cond_msg, &mutex_cond);
      //copy msg to p_msg
      for (i=0; i<N; i++) p_msg[i] = msg[myTid][i];
      //clear new message flag
      fNewMsg[myTid] = 0;
      //someone might be waiting, wake them up
      pthread_cond_broadcast(&cond_msg);
   }
   pthread_mutex_unlock(&mutex_cond);
}
```

Listing 13: Multithreading with MPICH + PThread (continue)

Listing 14 shows the basic architecture of a MPICH + OpenMP program**.** The architecture is similar to that of MPICH + PThread, except that thread creation and thread synchronization are implemented through different ways. Threads are created by define a parallel architecture, by using primitive *#pragma omp parallel.* The number of threads to be created is by the environment variable, *OMP_NUM_THREADS*.

The *thr* function shows how intra-node communication and inter-node communication should be carried out. Functions, *setMsg* and *readMsg* are written for intra-node communication. Since *mutex* and *conditioner* are not supported by OpenMP, thread synchronization must be implemented in different ways. We use the primitive, *#pragma omp barrier*. By putting a barrier between the write and read functions, we guarantee that no message will be read before written. This may looks convenient than the MPICH + PThread approach, but it requires all threads to be synchronized, which is unnecessary and in-efficient as only the two communicating threads should be synchronized with each other.

Exclusive calls to MPICH functions are implemented using the *#pragma omp critical* primitive, which guarantees that only thread will be in the following block at one time.

Other problems like point-to-point communications and collective communications between threads, global variables and functions, still apply here.

```
ompicc -DUSE_STDARG -DHAVE_STDLIB_H=1 -DHAVE_STRING_H=1 -DHAVE_UNISTD_H=1 -
DHAVE_STDARG_H=1 -DUSE_STDARG=1 -DMALLOC_RET_VOID=1 -I/home/hzhang/mpich-
1.2.6/include -L/home/hzhang/mpich-1.2.6/lib -lmpich -lsocket -lnsl -lrt -
lnsl -O1 hp.c -o hp
```

Since OpenMP programs can only be compiled by specially designed, OpenMP support compilers, *mpicc* can not be used to compile the program. We use *ompicc* [81] compiler, the compilation command is like above.

```
#include "omp.h"
#include "mpi.h"

float *msg[THREAD_PER_NODE]; //exchange border value
double msg_time[THREAD_PER_NODE]; //exchange time

/**************************************************************************
***/
int main(int argc, char *argv[])
{
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myPid);

   //create threads
#pragma omp parallel
   hp(omp_get_thread_num());

   MPI_Finalize();
}

void hp(int tid) {
   myTid = tid;

   //communicate with threads on the same node
   setMsg(myTid-1, msgl);
   //make sure all have written to left
#pragma omp barrier
   readMsg(myTid, revr);

   //communicate with threads on other nodes
#pragma omp critical
{
      MPI_Send(...);
      MPI_Recv(...);
}

//communicate between threads on the same node
void setMsg(int destTid, float *p_msg) {
   //copy p_msg to msg
   for (i=0; i<N; i++) msg[destTid][i] = p_msg[i];
}

void readMsg(int myTid, float *p_msg) {
   for (i=0; i<N; i++) p_msg[i] = msg[myTid][i];
}
```

Listing 14: Multithreading with MPICH + OpenMP

Multithreading in mpiJava and jPVM

Since Java multithreading mechanism is used in both mpiJava and jPVM to support intra-node communication and thread synchronization, the multithreading part is the same. We use mpiJava code to show how to develop multithreading applications on these two systems. The code is showed in Listing 15.

The computation is moved into the *run* method of a class that extends the *Thread* class. To create a thread is equal to create an instance of such class. The rest part is the same as in MPICH + PThread. The main class starts the threads, waits for them to finish, and then call the *MPI.Finalize* method.

The algorithm of intra-node communication and inter-node communication is the same as in MPICH + PThread, although they have to be implemented in different ways. *Mutexes* and *conditioners* are replaced with *monitors*. In Java, every object is itself a monitor. Using a static object, threads will share a same monitor. The *synchronized* keyword on this *monitor* guarantees exclusive calls to MPICH functions. For intra-node communications, a special class, *myBuffer* is created. An instance of *myBuffer* will be associated with every thread. Communications are implemented by writing or reading from these buffers. For example, to send a message, a thread will write to the buffer that associated with the destination thread. Every buffer is a *monitor*, which guarantees exclusive access to critical sections through the *synchronized* keyword and support thread coordination through the *wait* and *notifyAll* functions.

Although problems like point-to-point communications and collective communications between threads still exist, global variables and functions will not be a problem anymore. Global variables are declared as class members of the *worker* class and every thread will naturally have its own copy of global variables.

mpiJava and jPVM programs can be compiled with the standard *javac* command.

```
import mpi.*;

class appl {
   static public void main(String[] args) throws MPIException {
      MPI.Init(args);
      nprocs = MPI.COMM_WORLD.Size();
      myPid = MPI.COMM_WORLD.Rank();

      //create threads
      for (i=0;i<Thread_Per_Node;i++) {
         workers[i] = new worker(N, nprocs, myPid, i);
         thrds[i] = workers[i];
      }

      //start threads
      for (i=0;i<Thread_Per_Node;i++) thrds[i].start();

      //wait for threads to finish
      for (i=0;i<Thread_Per_Node;i++) thrds[i].join();

      MPI.Finalize();
   }
}

class worker extends Thread {
   static Object mpiObject = new Object(); //exclusive call to mpi

   worker(int p_N, int procs, int Pid, int Tid) {
      //get thread id
      myTid = Tid;
   }

   public void run() {
      //communicate with threads on the same node
      leftBuf.setMsg(msgr);
      myBuf.readMsg(revr);

      //communicate with threads on other nodes
      synchronized (mpiObject) {
         MPI.COMM_WORLD.Send(...);
         MPI.COMM_WORLD.Recv(...);
```

Listing 15: Multithreading with mpiJava

```
      }
   }
}

class myBuffer {
   //communication between the other thread
   private float msg[];
   private boolean fNewMsg;

   //called by the neighbor to transmit the new border
   synchronized public void setMsg(float p_msg[]) throws
InterruptedException{
      //make sure the last value has been read
      while (fNewMsg == true) wait();
      //copy p_msg to msg
      for (i=0;i<N;i++) msg[i] = p_msg[i];
      //set new message flag
      fNewMsg = true;
      //someone might be waiting, wake them up
      notifyAll();
   }

   //called by self to read the new border
   synchronized public void readMsg(float p_msg[]) throws
InterruptedException{
      //make sure new value has been written
      while (fNewMsg == false) wait();
      //copy msg to p_msg
      for (i=0;i<N;i++) p_msg[i] = msg[i];
      //clear new message flag
      fNewMsg = false;
      //someone might be waiting, wake them up
      notifyAll();
   }
}
```

Listing 16: Multithreading with mpiJava (continue)

In conclusion, advantages of VCluster are:

1. VCluster support shared memory communication. In this case, threads on a same node communicate using channels, just like threads on different nodes. While in other systems, shared memory communication must be managed by the user, through techniques like monitors and semaphores.

2. As most message passing libraries are not thread safe, users must make sure that no two threads are calling libraries functions, especially communication functions, at the same

time. Combined with the monitors and semaphores for shared memory communication, this can easily lead to problems like deadlock.

3. VCluster supports communication between threads. In other systems, as the communication is between processes, users need to code carefully and make sure the message is received by the correct thread. In our code, tags are used to identify the source and destination threads.

4. In VCluster, shared memory or message passing is hidden from the user. In other systems, whenever two threads want to communicate, users must decide which model should be used and send messages appropriately. This again can be quite tedious and results in much longer code. Listing 17 shows the code for this in MPICH+PThread.

```
//recv
if (inodei == myPid) {
   //on the same node
   nrc = readNsd(tid);
} else {
   //on the other node
   //exclusive call to mpi
   sched_yield();
   pthread_mutex_lock(&mutex_mpi);
   MPI_Irecv(&nrc,1,MPI_INT,inodei,tagNsd-
             tid,MPI_COMM_WORLD,&rcvRequest[tid]);
   pthread_mutex_unlock(&mutex_mpi);
}
```

Listing 17: MPICH + PThread Code for Thread Communication

5. After joined a group, VCluster assigns each thread a unique ID, and the destination thread can be specified by this ID. In other systems, users need to set the thread IDs, for example, a combination of the process ID and thread ID. To communicate with a thread, process ID need to be parsed out from the thread ID, and again the user need to decide which model to use.

6. VCluster supports collective communication based on threads. While in other systems, the user must first collect data from threads on the same node to one thread, then ask the underlying library to carry out the collective communication at inter-node level on these selected threads, and finally, distribute the results to each thread.

5.3.2 Experiments and Results

Beside the advantages in the previous section, VCluster also present good performance. Figure 16 and 17 shows the performance of Parallel Dirichlet Problem and Parallel Neural Network with multithreading. We discussed these two problems without multithreading in the previous section. Now we create 2 threads on each node, with each threading working like a process in the previous case. The experiment is run on up to 32 nodes. VCluster gives a similar performance of mpiJava and a better performance than JPVM.

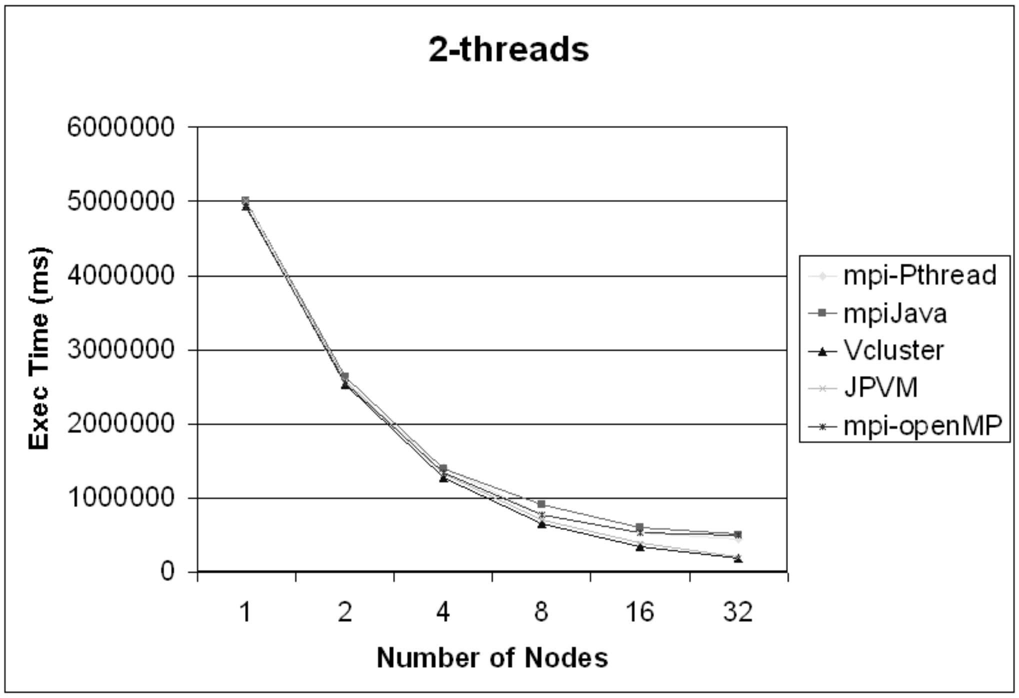


Figure 16: Parallel Dirichlet Problem with MultiThreading

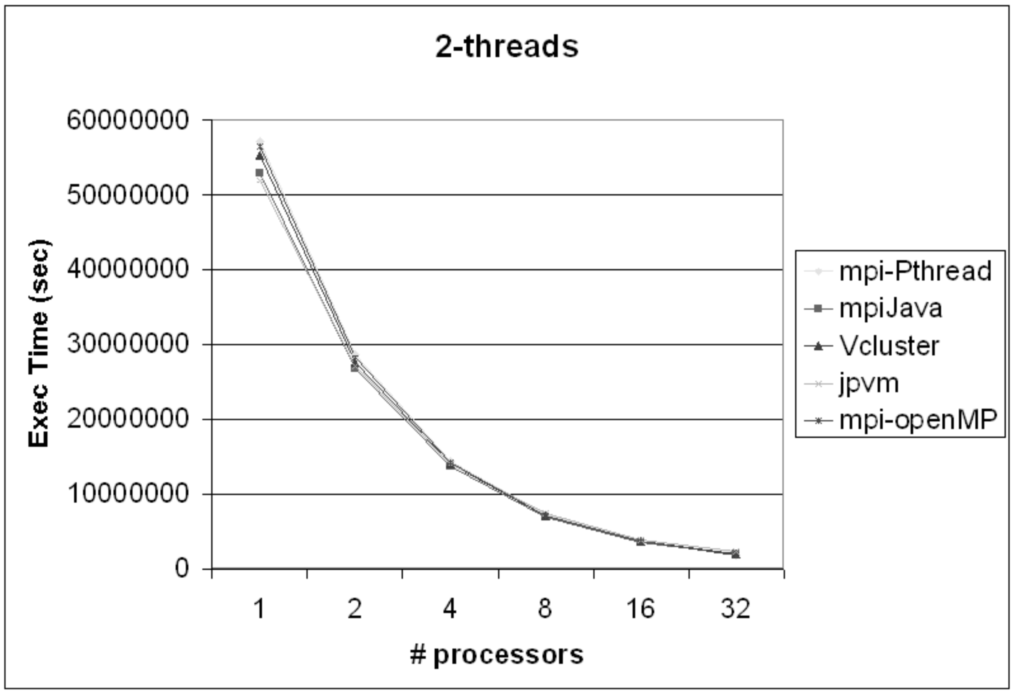


Figure 17: Parallel Neural Network with Multithreading

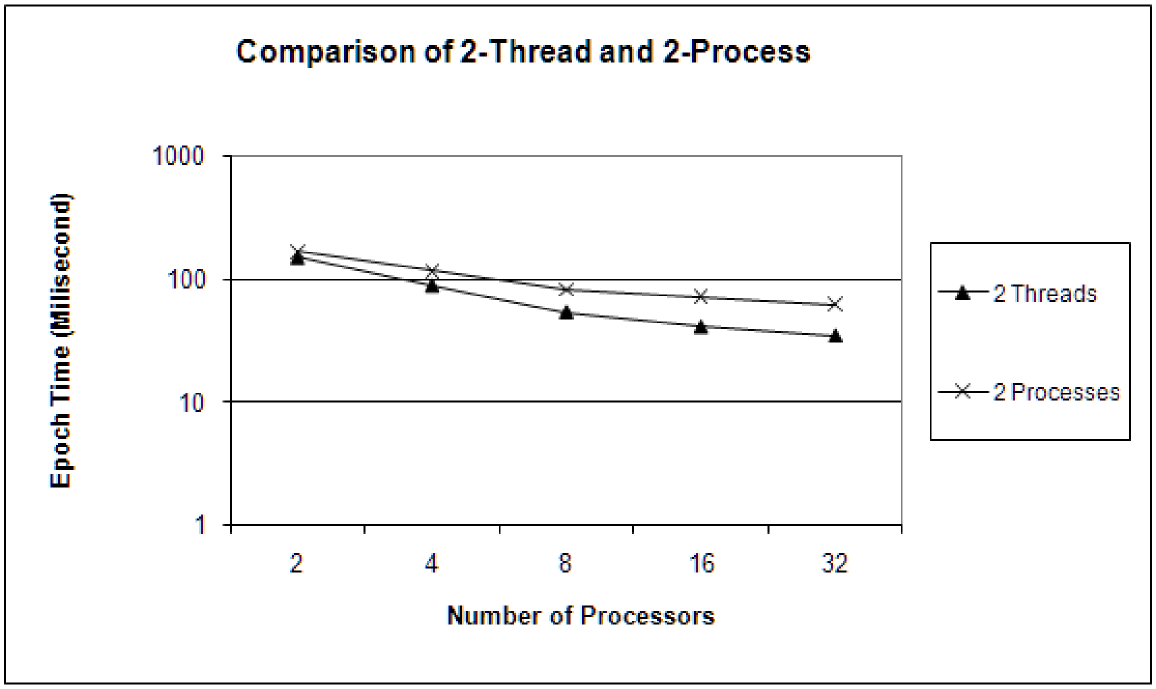


Figure 18: Comparison of 2 Threads and 2 Processes.

We also use the parallel Dirichlet problem to compare the communication performance of multithreading and multiprocessing. In multithreading, a thread is created for every processor. The communications between two threads on a same node are carried out through shared-memory by VCluster. In multiprocessing, a process is created for every processor. The communications between two processes on a same node are carried out through sockets by VCluster. The grid size is smaller than in the previous experiment to increase the communication to computation ratio. In this way, the difference in communication can affect the overall time more. Figure 18 shows the results of the experiment. Multithreading performs better than multiprocessing, and the difference increases when more processors are used.

5.4 Molecular Dynamics

Our third application is molecular dynamics. Molecular dynamics is a technique to simulate a set of interacting atoms, or particles. It is widely used in chemical and physical fields, like drug design and material design. It is also used in computer graphics, for example, to simulate the explosion of a gas bomb in computer games. Compared to the first two applications, this application requires much more computation. The algorithm is primarily based on [79], and we made some changes to suit our requirements.

In molecular dynamics, a set of atoms in a defined space are simulated. Every atom has an initial position and speed. The atoms exert forces to each other, thus incur acceleration. As the time evolves, all atoms will move and interacting with each other. The interacting of atoms is computed according to Newton's second law. That is,

$$m_i a_i = f_i$$

The equation means that acceleration can be calculated by the mass of the atom and the force. The mass of atoms is known or can be assumed, and the force can be calculated by the potential energy. Practically, a threshold for distance is used, called RCUT. If the distance of two atoms is larger than RCUT, the force on each other is so small that it needs not to be computed. The acceleration between two atoms is a function of their distance and the threshold distance RCUT.

The algorithm is to simulate the evolution of time. At every time step, the acceleration of each atom is computed, and atoms are moved accordingly. Since the interaction between every two atoms need to computed, the main computation part is a two-level loop and the time complexity is $O(N^2)$, where N is the number of atoms.

To parallelize the molecular dynamics problem, we divided the space into subspaces, with each parallel process assigned a subspace. In each time step, a process carried out the computation of atoms in its subspace.

In VCluster and MPICH + POSIX Thread, we create 2 or more threads on each machine. This further divids the space. For example, if originally we use two machines and thus two threads to divide the space into two sub-spaces, now we have 4 threads and the space is divided into four sub-spaces. openMP is good at parallelizing loops, and our main computation part is loops of numerical operation. Naturally we want to use openMP to parallelize these loops.

We expect that VCluster and mpich+pthread will show similar performance, while mpich+openMP will be almost twice slower. Suppose originally we have n atoms on each node, according to our analysis above, the computation complexity is $O(n^2)$. In VCluster and MPICH+PThread, the sub-space is further divided. If we create two threads on each node, each thread will take care of a sub-space which is half the size of the original sub-space, and thus has $\frac{n}{2}$ atoms. The computation complexity for each thread will be $O(\frac{n^2}{4})$, which is ¼ of the original time. Since we have two processors on each node, these two threads will be able to run in parallel. This will result in a loop time which is ¼ of the original loop time. In MPICH+openMP, the space is not further divided, instead the computation is parallelized. If we create two threads on each node, the computation for each thread will be $O(\frac{n^2}{2})$. These two threads can run in parallel, and result in a loop time which is ½ of the original loop time.

We experimented this application with a total of 131,072 atoms being simulated. This is the largest amount of atoms that can be stored in the memory of one node. Two threads are created on every node. We ran the experiment on 1 to 32 nodes. When one node is used, which means a total of two threads, each thread will simulate 131,072 / 2 = 65,536 atoms. When two nodes are used, which means a total of 4 threads, each thread will simulate 131,072 / 4 = 32,768 atoms. We would expect the computation time to be reduced to ¼. However, when more threads are used, communication time goes up, and the overall time will not be reduced to ¼. We ran the results on 1, 2, 4, 8, 16, 32 nodes. The results of the experiments are shown in figure 19 and 20. We can see that VCluster is the fastest, followed by MPICH+POSIX Thread and MPICH+openMP. As we have expected, MPICH+openMP is twice slower than MPICH+POSIX Thread. But MPICH+POSIX Thread is much slower than VCluster. We calculated the time spent on each step and found out that this is because the computation, which is a 2-level loop, contains several function calls. These functions calls will be executed for extraordinary times when the loop is executed. C somehow does not handle these functions calls as efficiently as Java does. This may because we use gcc, instead of cc, under the Solaris environment.

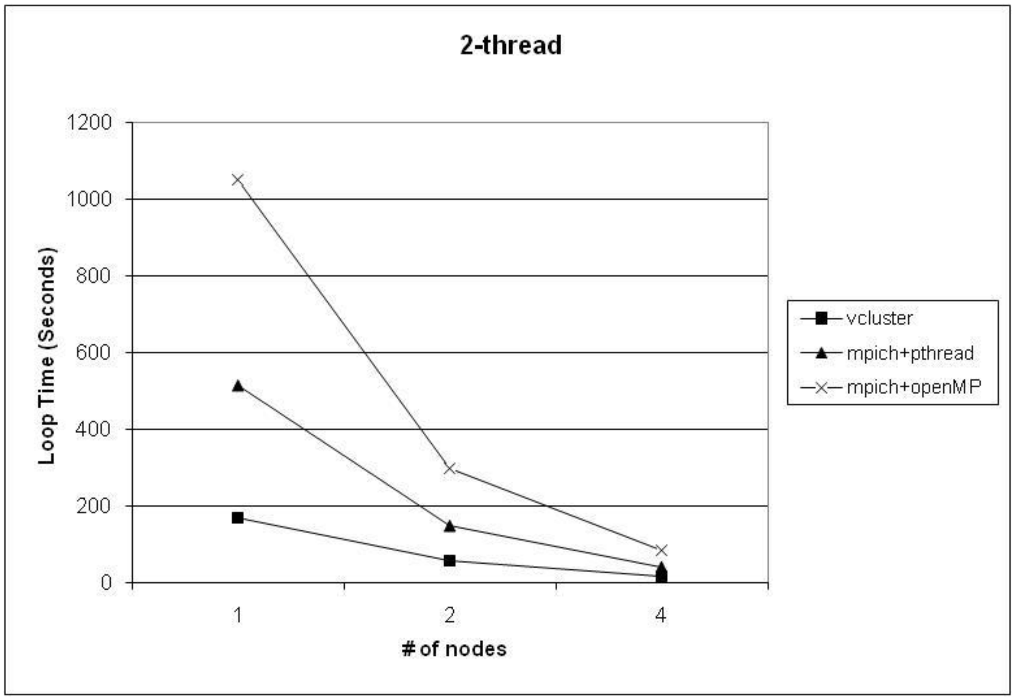


Figure 19: Parallel Molecular Dynamics with Multithreading - 1

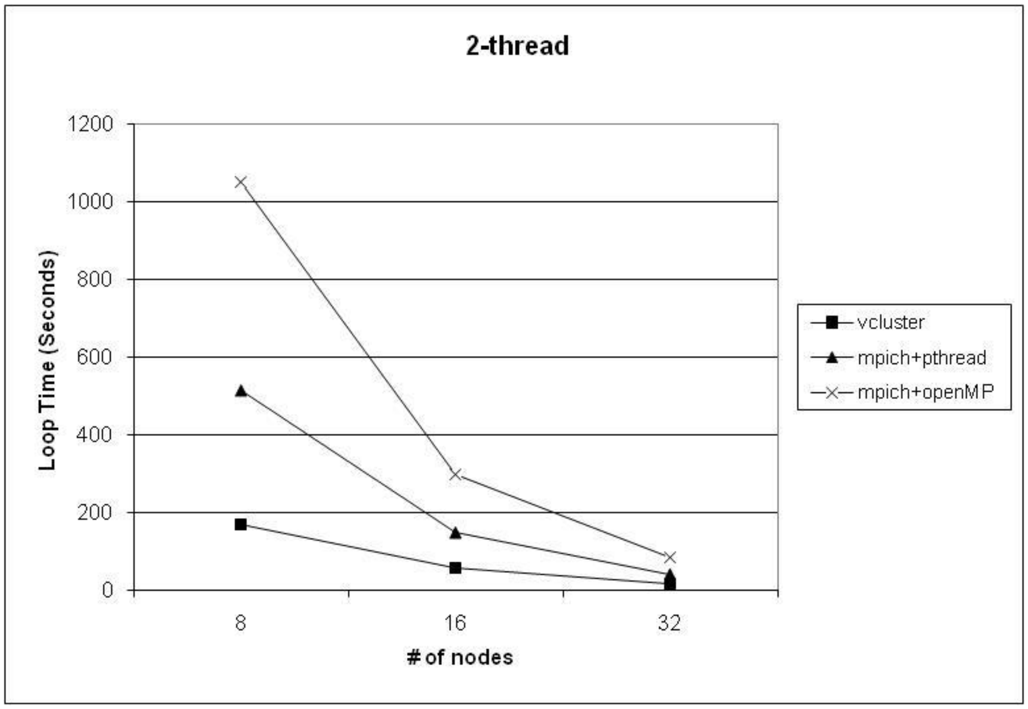


Figure 20: Parallel Molecular Dynamics with Multithreading – 2

5.5 Conclusion

In this chapter, we compare the performance of VCluster with other related libraries. The results of the experiments show that VCluster provides similar performance to Java related libraries, while making it much more easily to develop parallel applications on clusters of SMP machines.

CHAPTER SIX: PREDICTIVE MODELING USING COMPLEXITY ANALYSIS

The results of the experiments in the previous chapter indicate that when more parallelization is used (more threads and nodes), the computation time for each thread goes down, while the communication time goes up. When the decrease in the computation time exceeds the increase of the communication time, the overall time will goes down. After some point, when the decrease in the computation time is less than the increase of the communication time, the overall time will goes up. Beyond that point, using more threads will decrease the performance instead of increasing it. If we can find that point, we know how many threads we should use to achieve the best overall time. In this chapter, we describe a parallel runtime prediction model that uses an analytical modeling technique to investigate the performance of the parallel MD program. Using this model, we can predict the performance of the MD program on different number of nodes or different data sizes and see what the best run time is and how to get it. We validate the performance prediction models against the experimental data.

The analytical model is primarily concerned with predicting the performance and resource scaling characteristics on a variety of architectures. Analytical modeling techniques abstract the features of a parallel system as a set of parameters or parameterized functions in order to make the modeling task tractable [80]. Our analytical model models the behavior of a single time step of the parallel MD program, which includes concurrent computation, sequential computation, and communication overhead. Then, a linear equation is used to describe the gross behavior of the algorithm executed on a cluster.

During each time step, the following operations are executed on every thread.

1) update the speed of every atom to half the time step using the acceleration;

2) move the atoms according to their speed;
3) some atoms will move out to neighbor sub-spaces;
4) some atoms need to be copied to respective neighbors;
5) compute new acceleration;
6) update the speed of every atom to the other half of the time speed using the new acceleration;

We use these notations,

n, the number of atoms in each thread

p, the number of threads

$n_{neighbor}$, the number of neighbor atoms

n_{copy}, the number of atoms to copy to a neighbor thread

n_{move}, the number of atoms to move to a neighbor thread

Ci, i in 1, 2, 3 …, where each Ci is a constant

Operation 1 is a loop over all the atoms which updates the speed in three dimensions, the time is:

$$T_1 = n * 3 * C_1$$

Operation 2 is a similar loop, which updates the position of each atom instead of the speed, the time is:

$$T_2 = n * 3 * C_2$$

Operation 3 is more complex than the first 2 steps. It contains a loop over the three dimensions. For every dimension, all atoms are checked on whether it has moved out, the time is $n * 3 * C_3$. Then for every direction, atoms are moved out through network transportation. The time for network transportation consists of a start time $T_{startup}$, and the transportation time of each atom, noted as T_w. Since atoms move both in and out, the time is $2 * (T_{startup} + T_w n_{move})$. The whole loop over the three dimensions will take $3 * (n * 3 * C_3 + 2 * (T_{startup} + T_w n_{move}))$. After this loop is finished, the atom array needs to be compressed to exclude the atoms that have moved out. This is a loop over all the atoms, the time is $n*3*C_4$. Sum this time with the loop time, the overall time for operation 3 is

$$T_3 = n*9*C_3 + 6*T_{startup} + 6*T_w*n_{move} + n*3*C_4$$

Operation 4 is almost the same as operation 3. The difference is that it checks atoms on whether they need to be copied to neighbor threads and it does not have the final compress action. The time is

$$T_4 = n*9*C_5 + 6*T_{startup} + 6*T_w*n_{copy}$$

Operation 5 is the part that takes most of the time. First all the accelerations are reset to 0, the time is $n*3*C_6$. Next is a two-level loop that computes the force between every two atoms, including those neighbor atoms, the time is $(n*n_{neighbor} + \frac{n(n-1)}{2})*C_7$. After the acceleration is computed, all threads exchange the potential energy. That is, they send their local energy to every other thread and receive the potential energy from every other thread, the time is $2*(p-1)*(T_{startup} + T_w)$. Sum all this up, the time for operation 5 is

$$T_5 = n*3*C_6 + (n*n_{neighbor} + \frac{n(n-1)}{2})*C_7 + 2*(p-1)*(T_{startup} + T_w)$$

Operation 6 is the same as operation 1.

$$T_6 = T_1$$

The time for a single time step will be the sum of T_1 through T_6,

$$T = 2*T_1 + T_2 + T_3 + T_4 + T_5$$

The factors in the above equation of T is n, p, $n_{neighbor}$, n_{copy}, and n_{move}. The others are constants. After aggregating all these factors and simplifying them, we can obtain the following equation,

$$T = n * C_8 + n_{move} * C_9 + n_{copy} * C_{10} + n * n_{neighbor} * C_{11} + n^2 * C_{12} + p * C_{13} + C_{14}$$

According to our results of experiments, $n_{neighbor}$, n_{copy}, and n_{move} are all linearly related to n. That is, for example, $n_{move} * C_9 = n * b * C_9$, where b is another constant. Based on this assumption, the above equation can be further simplified to

$$T = n * C_{15} + n^2 * C_{16} + p * C_{17} + C_{18}$$

From the results of the experiments, we have a set of known n, p, and T. Using a linear regression method, we can get the coefficient values, $C_{15} = 0.00042125$, $C_{16} = 6.6229\text{e-}009$, $C_{17} = 0.038342$, $C_{18} = \text{-}2.3847$.

Figure 21 shows that the experimental results fit in the developed analytical model well. Using this analytical prediction model, we can predict the performance of the application. Figure 22 shows that to simulate 262,144 atoms, the loop time will be longer if 64 machines are used instead of 32. This means that we do not need more than 32 machines in this case. To utilize more than 32 machines efficiently, we can increase the number of atoms that are simulated. Again our prediction model can be used to predict the application performance. The advantage of prediction model over actual experiments is obvious. Carrying out the experiment will take much more time than using the prediction model, for example, months compared to minutes for this application. From Figure 22, we can see that when the number of atoms goes larger than 524288, we can use 64 machines efficiently. When the number of atoms goes to 2097152, 128 machines can be used efficiently.

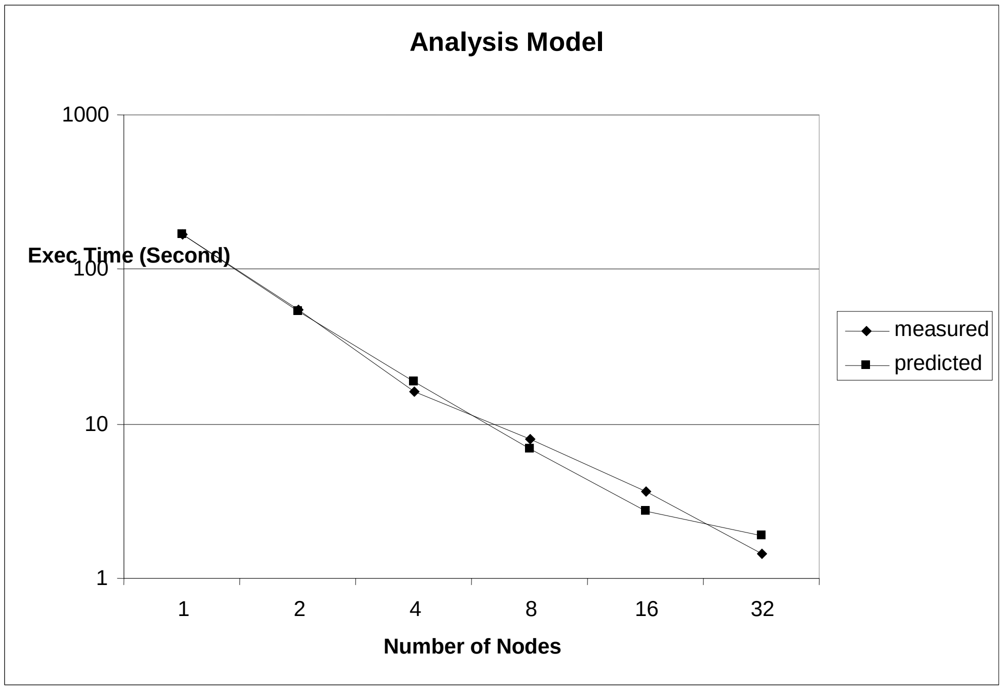


Figure 21: Analysis Model with Actual Numbers

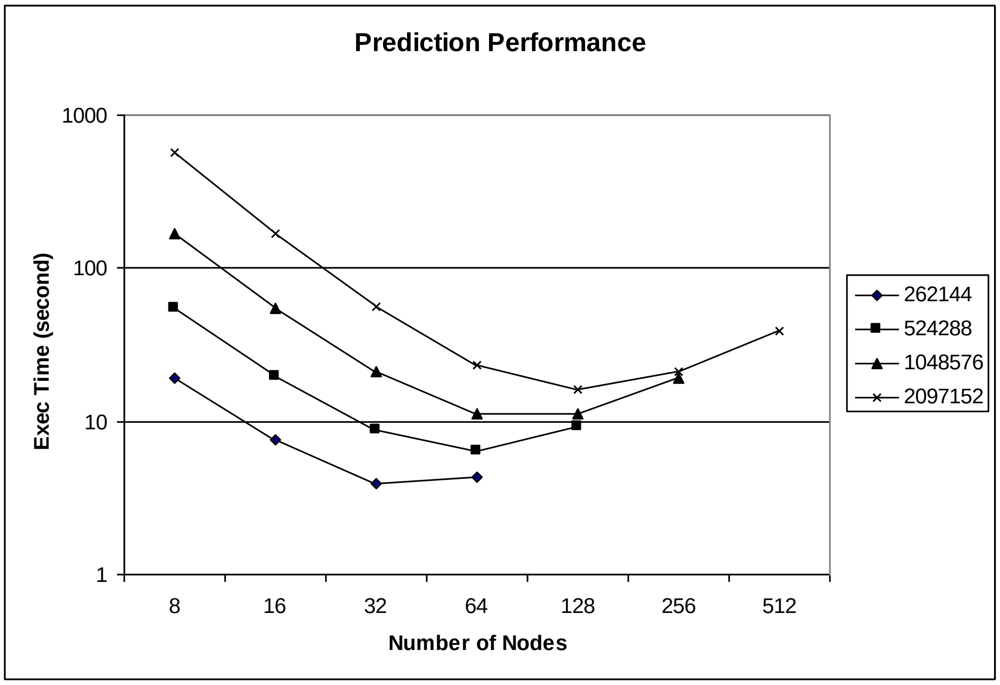


Figure 22: Prediction Model

CHAPTER SEVEN: EXPERIMENTS FOR LOAD BALANCING WITH VCLUSTER

Based on the basic functions, we want to implement more advanced features like load balancing. By migrating threads, loads in the cluster can be balanced dynamically.We show that dynamic load balancing through thread migration can improve the performance of the molecular dynamics application.

7.1 Introduction to Load Balancing

In cluster computing, load imbalances can be caused by many factors. It can be from the architecture of the application. For example, in molecular dynamics, particles can move around from one sub-space to another sub-space, potentially from one node to another node in the environment of cluster computing. It can also be affected by the configuration of nodes in a cluster. Nodes can be heterogeneous and have different computation power. Other factors, like other concurrently running applications, network latency, can also contribute to load imbalances during execution. Load imbalances often lead to performance degradation of applications, as the overall execution time is normally determined by the slowest node.

Dynamic load balancing can be utilized to improve application performance in cases of load imbalances. In conventional cluster computing libraries, dynamic load balancing is not supported and must be handled by users at application level. Many load balancing algorithms and schemes have been proposed and successfully employed in different application. These algorithms and schemes, although not proposed to be part of a cluster computing library, are worth studying as they were later adopted into different cluster computing libraries. There are two major categories in these systems. The first category falls into repetitive or iterative static partitioning. Data are repartitioned from time to time, according to historical statistics, to balance the load in the following computation. The second category is scheduling of parallel loops. Iterations of a loop are dynamically feed to idle workers to achieve an overall balanced load.

As the development of cluster computing libraries, recent libraries began to integrate dynamic load balancing as part of the library.

The rest of this chapter will be as following. In section 6.2.1 and 6.2.2 we discuss the two main categories in load balancing research. Section 6.2.3 is about the research on the exploit of thread/process migration in load balancing. In section 6.3 we discuss our implementation of load balancing. At last we show how load balancing can improve application performance through experiments.

7.2 Related Research

7.2.1 Mesh Partitioning

To discuss repetitive or iterative static partitioning, we first introduce meshing and mesh partitioning. Meshing is the procedure of discretizing the problem domain [92]. A mesh can be represented by a graph, with vertices representing sub-domains and edges representing the communication between them. This graph architecture makes mesh based applications a natural source of parallelization. Through mesh partitioning, meshes can be partitioned into connected sub-domains. In a cluster computing environment, every process will be assigned a sub-domain and be responsible for the computation of the sub-domain. Communications between sub-domains are carried out between processors. Generally, mesh partitioning can be abstracted as graph partitioning problems.

Due to the nature of the application, or when the number of processors becomes large, the load on processors can be much unbalanced. The mesh must be re-partitioned and sub-domains must be re-distributed to processors at runtime. In this distributed environment, the meshing partition algorithm must take into consideration the capability of processors, the network latency, along with the application. In repetitive or iterative partitioning, meshes are re-partitioned from time to time to achieve load balance. So the partitioning algorithm should also be efficient, as it will be carried out periodically during the runtime.

In [92], Jian et al proposed a communication tool, PART, addressing the difference of local and remote communication. In some distributed computing environments, the communication latency between processors can be very different. For example, in a cluster of SMP machines, the communication between processors on a same node, local communication, will be much faster than communication between processors on different nodes, remote communication. Their experiments show that PART can reduce the total execution time by up to 12%.

S-harp [93] tackled the problem of computation latency. Based on the fast feature of inertial partitioning and the quality feature of spectral partitioning, S-harp implemented a fast mesh partitioning algorithm.

Re-partitioning of a mesh is based on the load distribution of the mesh. This load distribution can be either computed or predicted based on previous data. Since there is no load balancing operation between two re-partitioning, the effectiveness of load balancing is based on the accuracy of the load distribution. This may incur problems in some cases. For example, in some applications the load of a mesh might not only depend on the number of data points, but also depend on the properties of each data point. In this case, the computation of load distribution might be too expensive to be carried out periodically for re-partitioning. In some other cases, the load distribution can not be accurately predicted. Load prediction is based on the assumption, "slow rate of load variation", which is apparently not always true.

7.2.2 Scheduling of Parallel Loops

Parallel loops are loops in which iterations are independent to each other. Because parallel loops are abundant in scientific applications and can be easily parallelized, the parallelization of them has been extensively researched, mostly on scheduling. The scheduling problem, basically, is how to schedule parallel loops on to workers, e.g. nodes in a cluster computing environment, so that the loops can be finished in the shortest time. The fundamental trade-off in scheduling of parallel loops is maintaining balanced load on workers while minimizing the scheduling overhead. To discuss different scheduling algorithms, we first introduce two notations.

N: the number of iterations.
P: the number of workers.

One simple and na ïve scheduling algorithm is to schedule equal portion of iterations to workers at once. This algorithm is called *static chunking* (SC). In SC, each worker is assigned a chunk of size N/P. Scheduling overhead is minimized as only one scheduling operation is needed. However, equal portion of iterations does not always mean equal work load. A common example is the conditional statements used in iterations. Branches of a conditional statement can have different work load. If iterations assigned to a worker tends to have more heavy-load branches than iterations assigned to other workers, its work load will be higher than average and the overall performance might not be optimal.

Alternatively, we can schedule one iteration at a time, which is known as *self-scheduling* (SS). Workers request for a new iteration when they become idle. In this case, we are sure that the finish time of all workers will not differ by more than the execution time of one iteration. Load on all workers are perfectly balanced. However, N scheduling operations are needed and because of the scheduling overhead, the overall performance might again not be optimal.

The above are two extreme cases. SC favors finge-grained, constant-length loops, and SS favors coarse-grained, variable-length loops. Other scheduling algorithms lie in the middle. We discuss several of them below.

Fixed-Size Chunking

Instead of scheduling one iteration of N/P iteration at a time, Kruskal and Weiss proposed the idea of scheduling fixed-size chunks [85]. They believe that given an upper bound of execution time, an optimal number K can be found such that the execution time will be within the given bound if iterations are scheduled in chunks of fixed-size K. They also gave the algorithm to calculate K. However, their algorithm only works when P and K are large. Furthermore, their algorithm does not take into consideration the average execution time of an iteration, whose effect could be significant as shown in [87].

In their paper, Kruskal and Weiss also speculate that it may be good to schedule large chunks at first, and later use smaller chunks to smooth over the previous load imbalances. Many decreasing-size schemes have been proposed after that, including the famous *factoring* scheme [86].

Guided Self-Scheduling

This is a decrasing-size scheme proposed by Polychronopoulus and Kuck [88], which addresses the problem of uneven processor starting times. When a worker request for work, 1/P of the total work left will be assigned. For example, if N=16 and P=4, the first worker will get 16/4=4 iterations; the second worker will get (16-4)/4=3 iterations; the whole distribution is shown in the following line.

4, 3, 3, 2, 1,1,1,1

Polychronopoulus and Kuck proved that for constant-length loops, GSS guarantees that the finishing time of all processors will differ within the execution time of one iteration. They also showed in their experiments that GSS out-perform SS in variable-length loops.

Factoring

GSS is good, but a problem with GSS is that GSS might schedule too much work at first, and the later smaller chunks are not able to smooth over the previous un-evenness. Flynn and Hummel analyzed this problem and proposed that a scheduling algorithm should schedule as much work as possible while they must have a high probability to finish before the optimal time. We omit the induction process (which can be found in [86]), and present their algorithm directly. In *factoring*, iterations are distributed in batches of P equal size chunks. The total number of iterations in a batch is a fixed ratio of the remaining, normally half of it. Using the same example, N=16, P=4, the distribution is as following:

2,2,2,2,1,1,1,1,1,1,1,1

We can see that compared to GSS, fewer iterations are allocated at first, leaving enough left to compensate for the un-evenness when necessary.

Factoring has been widely researched and many algorithms based on *Factoring* have been proposed, like *Weighted Factoring,* A*daptive Factoring,* and *Weighted Adaptive Factoring* [89-91]. These algorithms have been widely adopted in distributed applications and also cluster computing libraries [94].

7.2.3 Thread/Process Migration in Load Balancing

Traditionally, load balancing is handled by the parallel applications. With the advance in cluster computing libraries, especially with the support of multithreading and dynamic thread/process migration, it is possible to incorporate load balancing into cluster computing libraries.

Although a lot of research has been done in the use of thread migration in load balancing, they are mainly focused on the new issues incurred by the distributed environment, for example, the network latency. In [95, 96], loads on threads are monitored by the systems. When the system determines that the load on a node is high, it will find a node with light load and decide which threads to be migrated from the high load node to light load node. The threads will then be stopped from computation, migrated to the destination nodes, and re-started for computation. In a distributed system, other resources associated with a thread, like the communication resources, data used by the thread, must also be handled with care. As we discussed in chapter two, most current libraries that support thread migrations are based on distributed shared-memory systems. The data shared between threads, and the communications incurred by this sharing to keep consistence of data, must be taken into consideration for the decision of migrating thread. If the increase of communications out-beat the gain of load balancing, load balancing is useless. The computation of communication cost must also be fast, to reduce load balancing overhead. Originally, communication cost is calculated as the total number of data pages shared by threads, which can be very expensive. In [97], Liang et al analyzed the communication cost of different sharing types and proposed a much faster algorithm for the computation of communication cost.

7.3 Implementation of Load Balancing in VCluster

As we have discussed, load balancing is implemented in VCluster through thread migration. Load balancing must be carried out among a group of threads. So, to implement load balancing, we first implement the thread group, called *VC_Group*, in VCluster. Periodically, applications in a thread group exchanges load information. The group analyzes the load distribution and determines whether thread migration is necessary. If thread migration is necessary, the group also determines which threads to migrate and where the threads should be migrated to. To describe the implementation of load balancing, we first describe the implementation of thread group.

7.3.1 Thread Group

Thread group is implemented in VCluster through the *VC_Group* class. All threads in a group join the group at the beginning of the computation. After that, they can exchange messages by their indexes in the group. *VC_Group* manages the channels between all the threads. This implementation further hides the physical position of threads from each other. Through the virtual channels, communication threads do not need to know the physical position of each other.

With thread group, they do not need to know the physical position even after some of the threads have migrated. *VC_Group* will update the related channels automatically.

Another powerful feature of thread group is collective communication between threads. In conventional cluster computing libraries, all or some of the processes can form groups. A group of processes can carry out collective communications to exchange data in a group-wide scope. For example, the broadcast function can send data from one process to all other processes in the group. The broadcast function may look simple, but collective communications can be rather complicate. Nearly all common group-wide data operations can be implemented as collective communication functions. Collective communication functions have been proven to be extreme useful in abundant scientific communications, like the experiment programs we have used.

Under a multithreading architecture, it is desirable to carry out collective communication between threads. This is not supported by conventional cluster computing libraries. As shown in our experiment programs, the user application must collect data from threads in a process, carry out collective communications between processes, and distribute the result to threads. In these libraries, the communications are between processes. Since collective communications are actually a series of two-way communications and group-wide operations, it will be very difficult, if not impossible, to implement collective communications between threads in these libraries. While in VCluster, as our model is based on threads, the implementation will be very straightforward. We have implemented several collective communications.

7.3.2 Load Balancing

VC_Group currently provides two load balancing functions, with each implementing a different load balancing algorithm. Each load balancing function takes in two parameters, the load and the threshold. Periodically, all threads in a group call the same load balancing function with their load and the same threshold. With the load information from every thread, *VC_Group* has a clear view of the current load distribution and can run different load balancing algorithms upon it. If a thread is to be migrated, the load balancing function will return the index of the destination node; otherwise -1 is returned. All channels that are related to this thread are updated.

As there are many load balancing algorithms, with probably more in the feature, it is impossible to write a system that provides all these algorithms. With this observation, we positioned VCluster as a platform to implement load balancing algorithms, and designed it so that additional load balancing algorithms can be easily added. The *VC_Group* class provides a series of functions to easy the implementation of load balancing algorithms. For example, the *updateMember (member_index, destination_index)* function makes all the necessary changes to migrate a thread. When a load balancing algorithm decides to migrate a thread, it simply calls this function.

7.4 Experiments and Results

To show that the load balancing in VCluster works, we modified the molecular dynamics application for experiment. In the distributed molecular dynamics application, every thread will be responsible for particles in a sub-space, and every node can have several threads. As particles can move around, it might happen that the number of atoms in one sub-space is much larger than the number of atoms in another sub-space. As a result, the load on some nodes can be much heavier than the load on other nodes, slowing down the overall performance. By re-distributing the threads, we can re-even the load on every node and thus improve overall performance.

To simulate these situations and show how load balancing can tackle these problems. We intentionally make the load un-balance and use load balancing to balance it.

7.4.1 Migrate One Thread at a Time

In our first experiment, we only migrate one thread at a time. This experiment includes two nodes. The load ratio is initially set to 4 : 1. With load balancing, the ratio will become 3 : 2 after one iteration. We would expect that the iteration time of the second iteration will be less than the time of the first iteration. And since no further balancing is needed from the third iteration, the time should be further less. Figure 23 and Figure 24 shows the comparison with and without load balancing. Figure 24 shows the expected performance. However, in Figure 23, the iteration time of the second iteration is the same as the first iteration. This is because the time spent on load balancing and thread migration is almost the same as the time saved. In Figure 23, since the data is much larger, the time saved is much larger than the time spent on load balancing.

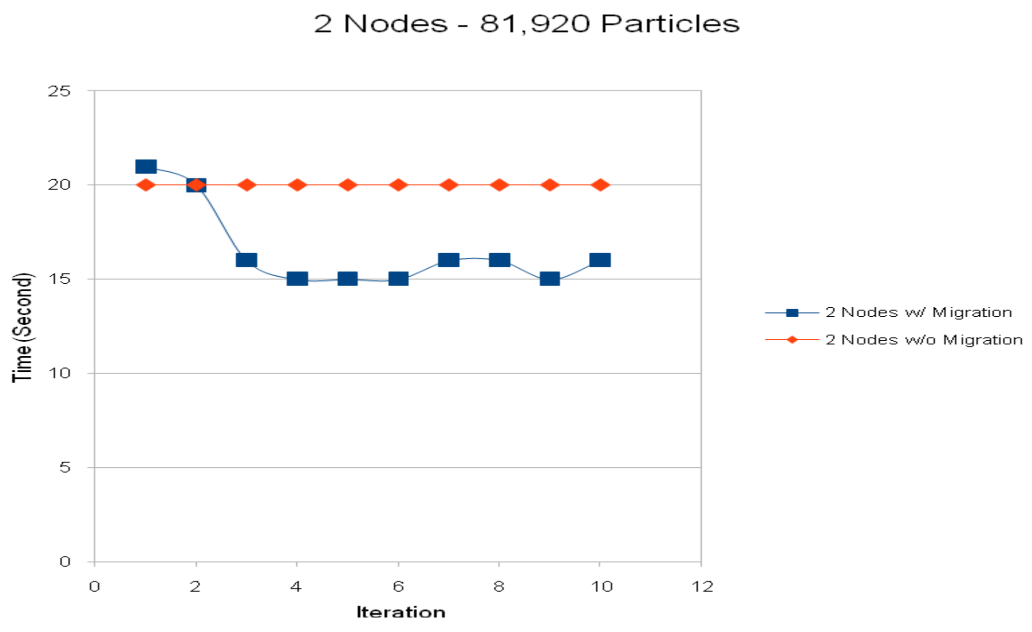


Figure 23: Load Balancing on Two Nodes 1

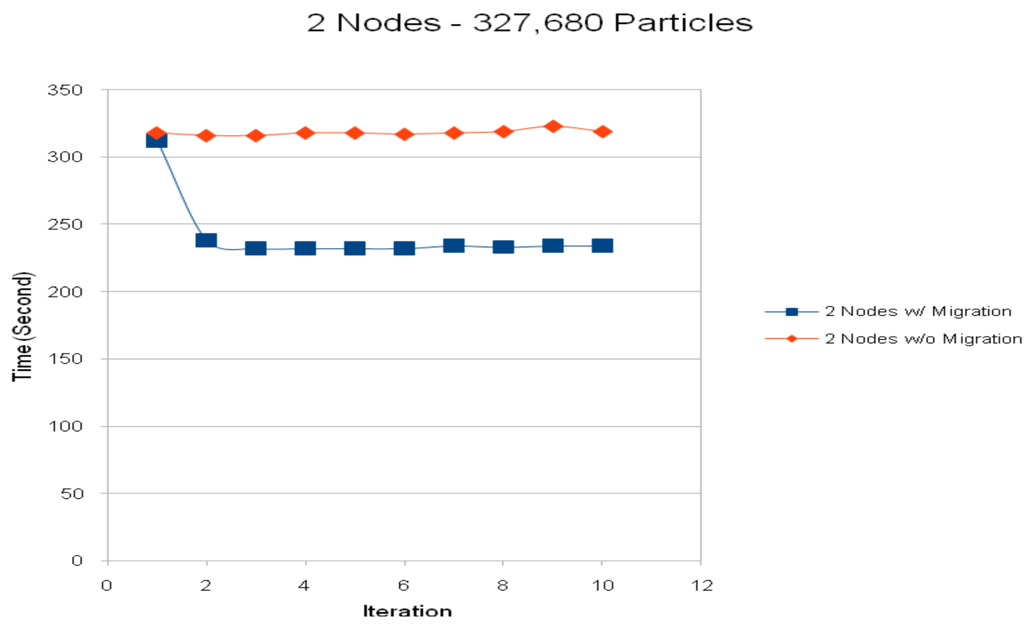


Figure 24: Load Balancing on Two Nodes 2

7.4.2 Migrate More than One Threads at a Time

In our second experiment, we migrate more than one thread at a time, to balance the load quickly. Four nodes are used in this experiment. The load ratio is initially set to 4:1:1:1. If one thread is migrated at a time, the ratio will become 3:2:1:1 after the first iteration, and 2:2:2:1 after the second iteration. If we make all necessary migrations at one time, the ratio will become 2:2:2:1 after the first iteration, saving more time. Figure 25 and 26 show the experiment results. Multi-migration balances the load after the first iteration, while single-migration takes two iterations. When the data size is small, the time spent on load balancing is larger than the time saved, resulting in longer overall time. This does not happen when the data size is large.

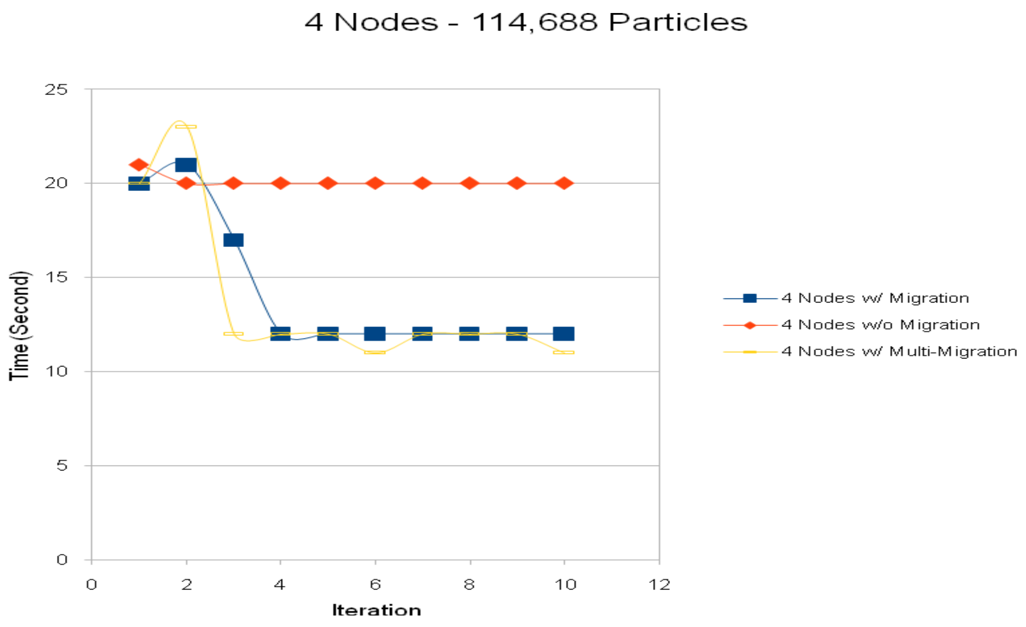


Figure 25: Load Balancing on Four Nodes 1

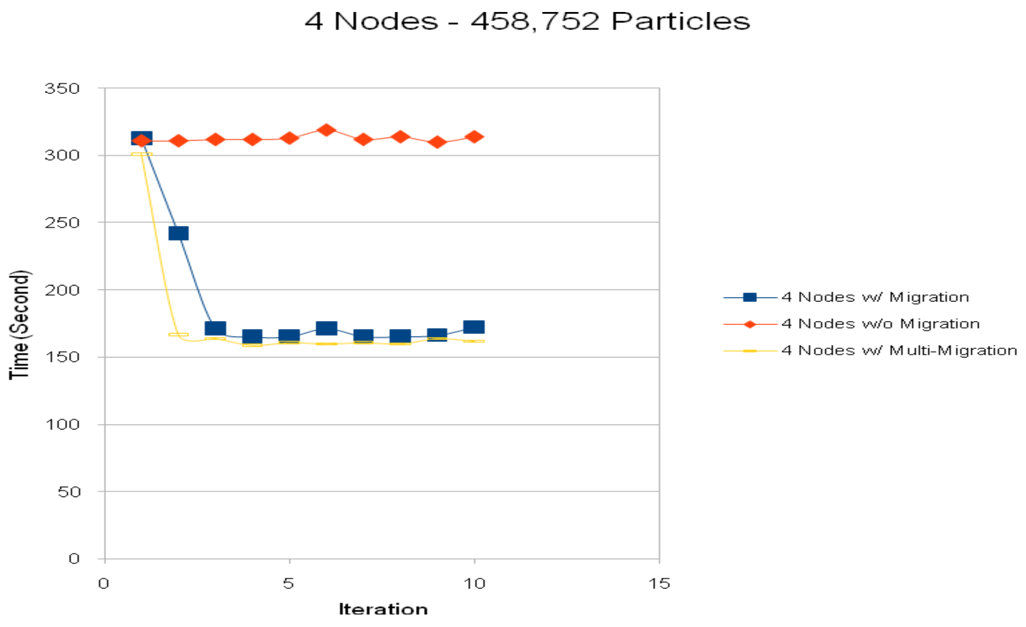


Figure 26: Load Balancing on Four Nodes 2

7.4.3 Migrate More than One Thread from More than One Node as a Time

In our third experiment, we try to migrate more than one thread from more than one node at a time. This case might occur in real applications and we want to make sure it can be handled by VCluster. Eight nodes are used in this experiment. The load ratio is initially set to 4:4:2:2:1:1:1:1. The balanced ratio is 2:2:2:2:2:2:2:2. Figure 27 and 28 show the results. The results are similar to the results in the previous section, which means VCluster can handle multi-migration from multi-nodes.

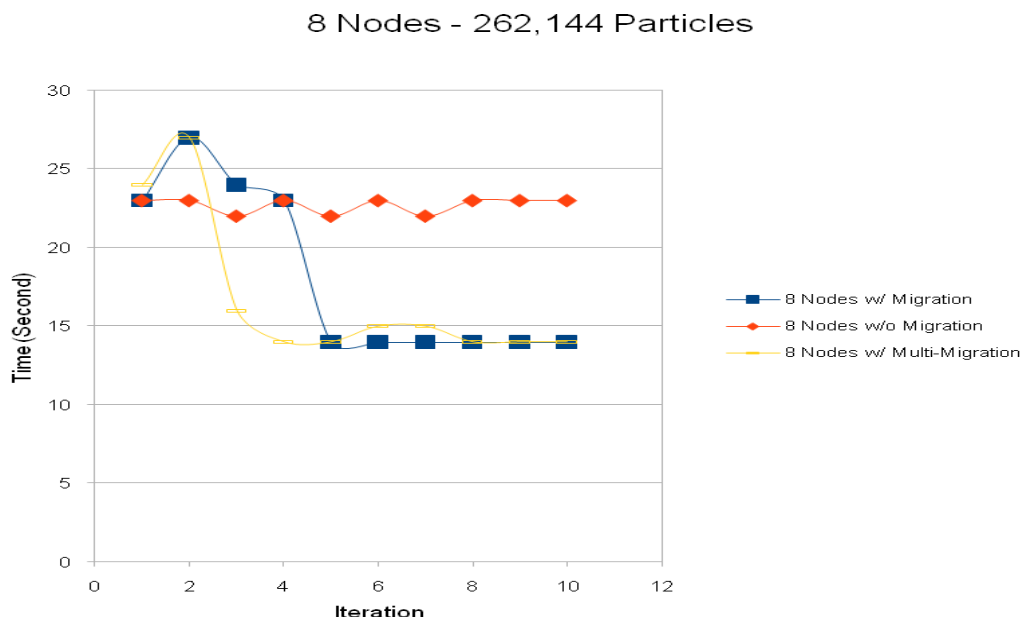


Figure 27: Load Balancing on Eight Nodes 1

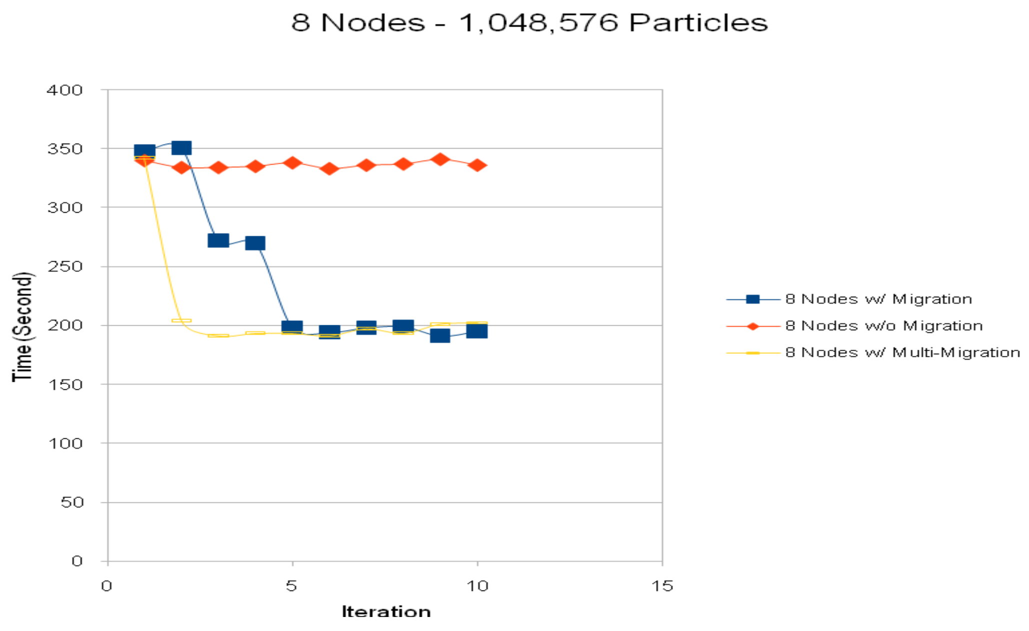


Figure 28: Load Balancing on Eight Nodes 2

7.4.4 Multi-Threading Issue with Linux Thread Library

When we were working on the above experiments, we ran into one issue with the Linux thread library. The issue is, when multi-threading is used, the performance of an application is very un-stable. It took us quite some time to figure out where is the problem and we finally find out the problem is within the Linux thread library, or the compatibility of Java and the Linux thread library. We never had this problem in Solaris.

We wrote a simple multi-threading application in Java to confirm that the problem is with the Linux thread library, or the compatibility of Java and the Linux thread, and is not incurred by VCluster. This simple application will have four threads. Every thread will do the same matrix multiplication. We ran the application 10 times and the time for every run is shown in Figure 29. We can see that the time varies from 300 seconds to 800 seconds.

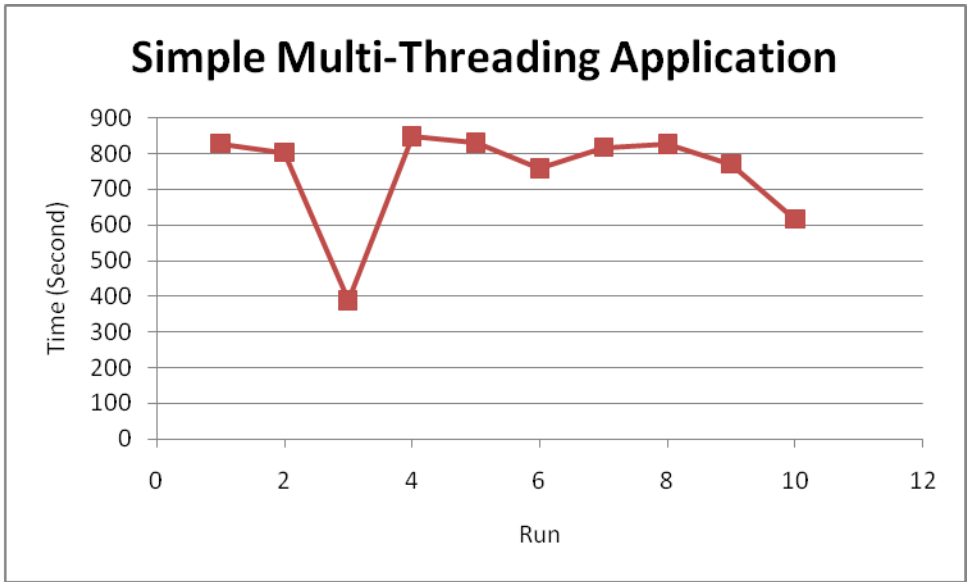


Figure 29: Multiple Run of a Simple Application

We have not figured out how to resolve this problem. Because of this issue, the data used in the pervious sections are all the best data we observed in many runs. These data are consistent with the data we observed when the cluster is installed with Solaris. Also because of this issue, we recommend run VCluster on clusters with Solaris.

7.5 Conclusion

In this chapter, we discussed the implementation of load balancing through thread migration in VCluster. Load balancing is built on top of thread group. The design of thread group makes VCluster a platform to develop load balancing algorithms easily. Thread group also support collective communication, which is desirable in multi-threading cluster computing but difficult to be implemented in conventional process-based libraries. We also show that load balancing can improve the performance of application through experiments.

CHAPTER EIGHT: FUTURE WORK AND CONCLUSION

In this chapter, we describe what can be done to further enhance VCluster and the conclusion. We want to extend VCluster to a higher level of parallel computing architecture, mini-grid.

8.1 Mini-Grid

Mini-grids are also called cluster of clusters. A mini-grid forms a larger computation platform by connecting a number of clusters together. This larger computation platform enables larger scale parallel applications. With an agent-based architecture, VCluster can be easily extended to support mini-grids.

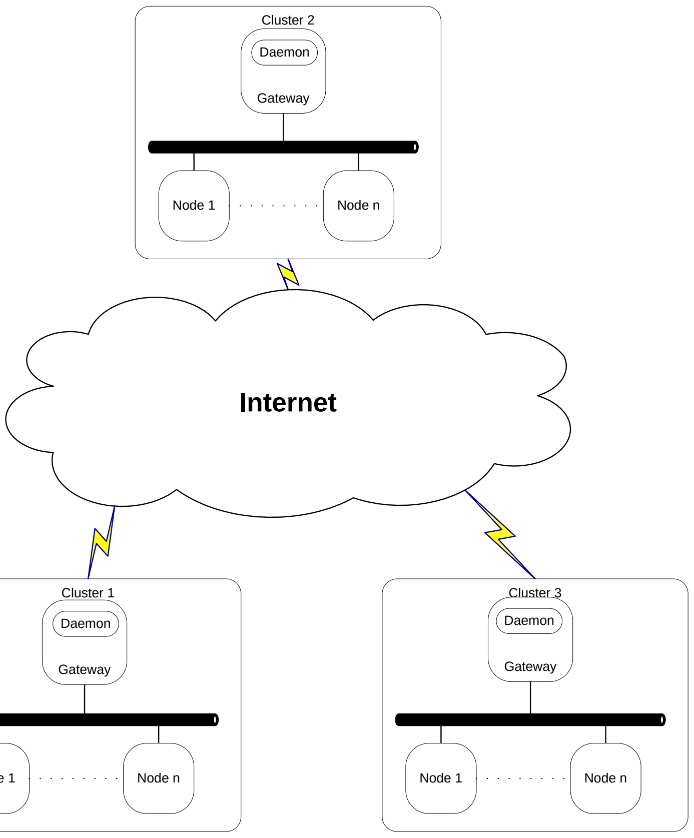


Figure 30: VCluster Architecture for Mini-Grid

Figure 30 shows the conceived architecture of VCluster on a mini-grid. A cluster normally has a gateway node that is connected to the outside world, while other nodes are protected in local network. In VCluster, a daemon is running on every node, including the gateway node. The daemons on these gateway nodes can be connected through the outside network, e.g. the internet. These daemons can relay messages between internal nodes in different clusters. However, communications between clusters should not be used too often as the communication latency is much higher than communications within a cluster.

After this mini-grid architecture is finished, we want to extend the load balancing scheme to this larger architecture.

Intensive research has been done in the area of load balancing on grid computing. Grid computing is different from cluster computing in that it is on a very larger scale, and the components are weakly connected and are especially heterogeneous. With this background, the load balancing research on grid computing focuses on the scheduling of tasks submitted by tremendous users and resource management [99-101]. [99] is of special interest to us as it uses a two-level architecture. The whole grid is divided into groups and every group will have a coordinator. This architecture is similar to that of a mini-grid. But for the load balancing on mini-grids, we focus on the load balancing of just one computation intensive application, as mini-grids are far more dedicated computing resource than grids and we can use it exclusively.

Mini-grid is a new area and so is the load balancing on mini-grid. We will identify the possible issues, design a load balancing scheme based on VCluster, and implement it.

8.2 Conclusion

In this research, we designed and developed a portable parallel runtime system for cluster computing, called VCluster. VCluster was designed to support both shared memory programming and message passing in a single framework, which is desirable for the new cluster computing environment, a cluster of symmetric multiprocessors. VCluster provides the concepts of virtual thread, states, and channels through which thread migration is possible. A prototype implementation was done using 100% Java codes for portability. Underlying architectural differences of the multiprocessor machines are hidden from the user applications by the developed library.

VCluster makes it much easier to develop multithreading parallel applications compared to conventional libraries like MPI. As multithreading is supported in VCluster, no extra thread libraries are needed. Communications between threads in VCluster are through virtual channels. Virtual channels automatically choose shared memory or message passing for the communication according to the relative position of the two threads, intra-node or inter-node, freeing the user applications from dealing with two communication modes all the time. Virtual channels also make it easy to implement thread migration. The relative position of threads are hidden from the user application and communications between two threads remain the same even if one or two of the threads have migrated. The communication system is also designed to be thread safe so multiple threads can call communication functions simultaneously.

We have developed three applications, parallel dirichelet problem, parallel back-propagation neural network, and parallel molecular dymanics, to compare the performance of VCluster with three other libraries, MPICH, mpiJava, and JPVM. OpenMP and POSIX Thread are used to implement multithreading in MPICH. The experimental results show that the performance of VCluster is comparable to MPICH with OpenMP or POSIX Thread. VCluster is different from those message passing tools in that it facilities the use of a cluster of multiprocessors more efficiently. VCluster utilizes a separate send and receive thread to send and receive messages. This implementation makes it possible for communication and computation to run simultaneously on multiprocessor machines.

The basic computing unit in VCluster is a virtual thread. The word virtual comes from the fact that virtual threads can migrate from one machine to another machine in a cluster, while a real thread is bound to a physical machine. Thread migration is implemented in VCluster through the concept of virtual state. To migrate a virtual thread, its associated virtual state is migrated to the destination machine, and then a virtual thread is created on the destination machine to continue the computation.

Virtual threads in VCluster can form a thread group. A thread group creates channels between member threads automatically and allows virtual thread to communicate with each other through their group ID. A thread group also provides collective communication methods to its member threads, which is hard to be implemented in conventional process-based message passing libraries. When member threads are migrated, a thread group also updates the related channels automatically.

With thread migration and thread group, dynamic load balancing is implemented in VCluster. Dynamic load balancing is desirable in applications where the load distribution can be very un-balanced, which results in bad performance. Performance can be improved by re-balancing the load distribution. In VCluster, threads are migrated to re-balance the load distribution. A user application periodically calls the load balancing functions provided by thread group. A load balancing function determines whether thread migration is necessary and if so which threads should be migrated and to which machines. We modified the molecular dynamics application for experiment. The results show that dynamic load balancing in VCluster can greatly improve the performance in some cases.

There are still a lot can be done to further improve and enhance VCluster. We plan to expand VCluster to support mini-grid, cluster of clusters. A mini-grid forms a larger computation platform by connecting a number of clusters together. This larger computation platform enables larger scale parallel applications. Since the architecture of a mini-grid is very different from a cluster, a lot of challenges are lying ahead.

APPENDIX A: SOURCE CODE

Appl.java

```
package VClusterApp.Example;

import VCluster.VCluster;
import VCluster.VC_Thread;
import VCluster.VC_Channel;
import VCluster.VC_ChannelSet;

public class Appl
{
   public static void main(String args[])
   {
      VCluster vc;
      VC_ChannelSet  vchannelSet;
      VC_Channel vchannel;
      applThread vthr ;
      applState vState

      // Initialize
      vc = new VCluster(args);

      // Create a VC_Thread
      vthr = new applThread();
      // Associate a state with this thread
      vState = new applState();
      vthr.addVCState(vState);
      //add this thread to VCluster
      vc.addThread(vthr);

      // Setup the communications (channelSets and channels)
      if (vc.getMyPid() == 0) {
         vchannelSet = new VC_ChannelSet("sendChannelSet");
         vchannel = new VC_Channel(1,"ch", vc.VC_WRITE);
      }
      else {
         vchannelSet = new VC_ChannelSet("recvChannelSet");
         vchannel = new VC_Channel(0,"ch", vc.VC_READ);
      }
      vchannelSet.addChannel(vchannel);
      vthr.addChannelSet(vchannelSet);

      // Start the computation
      vc.start();
      vc.finalize();
   }
}
```

applStateJava

```
package VClusterApp.Example;

import VCluster.VC_State;

public class applState extends VC_State
{
   public int x;

   public applState ()
   {
      x = 999;
   }
}
```

applThread.java

```
package VClusterApp.Example;

import VCluster.VC_Thread;
import VCluster.VC_State;
import VCluster.VC_Channel;
import VCluster.VC_ChannelSet;

public class applThread extends VC_Thread
{
   // User need to implement this run method for each different thread
   public void run()
   {
      applState state;
      VC_ChannelSet vchannelSet;
      int myTid, myPid;

      //get my process id and thread id
      myPid = getPid();
      myTid = getTid();
      VC_Println("** Hello World : pid = "+myPid+", tid = "+myTid);

      //access state
      state = (applState)getState();
      VC_Println("** x = "+state.x);

      if (myPid == 0) {
         //retrieve channel
         vchannelSet = getChannelSet("sendChannelSet");
             VC_Channel vchannel = vchannelSet.getChannel("ch");
             //send messages
         vchannel.writeInt(888);
         vchannel.writeFloat((float)777.0);
         vchannel.writeDouble(666.0);
             int arrInt[] = {1,2,3,4};
         vchannel.writeObject(arrInt);
             double arrD[] = {1.0,2.0,3.0,4.0};
         vchannel.writeObject(arrD);
      }
      else if (myPid == 1) {
         //retrieve channel
         vchannelSet = getChannelSet("recvChannelSet");
             VC_Channel vchannel = vchannelSet.getChannel("ch");
             //receive messages
             int    a = vchannel.readInt();
             float  b = vchannel.readFloat();
             double c = vchannel.readDouble();
         VC_Println("** recved a = "+a);
         VC_Println("** recved b = "+b);
         VC_Println("** recved c = "+c);
             int arrInt[];
             arrInt = (int []) vchannel.readObject();
```

```
              VC_Println("** recved arrInt =
"+arrInt[0]+arrInt[1]+arrInt[2]+arrInt[3]);
              double arrD[];
              arrD = (double []) vchannel.readObject();
              VC_Println("** recved arrD = "+arrD[0]+", "+arrD[1]+",
"+arrD[2]+", "+arrD[3]);
       }

       finalize();
   }
}
```

APPENDIX B: DOCUMENTATION

VCluster

Field Summary

public static final int VC_WRITE

public static final int VC_READ

Constructor Summary

void VCluster (String args[])

Method Summary

void addThread (VC_Thread vthr):	Add a virtual thread.
void finalize ():	Finalize the computaton.
int getMyPid ():	Return the process ID of this VCluster instance.
int getTotProcs ():	Return the total number of processes.
void start ():	Start the computation.
void VC_Println (String s):	Print out a message at the master node.

Field Details

VC_WRITE

public static final int VC_WRITE

Used in creating virtual channels to specify that data is read from the channel.

VC_READ

public static final int VC_READ

Used in creating virtual channels to specify that data is written to the channel.

Constructor Details

VCluster

public VCluster (String args[])

Initialize a VCluster instance.

args: command line arguments

returns: nothing

Method Details

getTotProcs

public int getTotProcs()

Return the total number of processes.

returns: total number of processes.

getMyPid

public int getMyPid()

Return the process ID of this VCluster instance.

returns: process ID.

start

public void start()

Start the computation. Called in the main class of an application.

returns: nothing

finalize

public void finalize()

Finalize the computaton. Must be called at the end in the main class of an application.

returns: nothing.

addThread

synchronized public void addThread(VC_Thread vthr)

Add a virtual thread. Called in the main class of an application.

returns: nothing.

VC_Println

public void VC_Println(String s)

Print out a message at the master node.

s: the message to be pritned out.

returns: nothing.

VC_Channel

Field Summary

Constructor Summary

void VC_Channel (int t_pid, String t_chName, int t_mode)

Method Summary

int getDestPid (): Return the destination process ID

int getMode (): Return the mode of this channel, read or write.

String getName (): Return the name of this channel.

boolean isAvailable (): Check whether a message for this channel has arrived.

double readDouble (): Read a double. Blocks until the message has arrived.

float readFloat (): Read a float. Blocks until the message has arrived.

int readInt (): Read a integer. Blocks until the message has arrived.

Object readObject (): Read an object. Blocks until the message has arrived.

void writeDouble (double vdouble): Write a double. Message might not have been sent or received.

void writeFloat (float vfloat): Write a float. Message might not have been sent or received.

void writeInt (int vint): Write an integer. Message might not have been sent or received.

void writeObject (Object o): Write an object. Message might not have been sent or received

Field Details

Constructor Details

VC_Channel

public VC_Channel(int t_pid, String t_chName, int t_mode)

Initialize a virtual channel.

t_pid: process id of the destiation thread

t_chName: name of the channel

t_mode: VC_READ or VC_WRITE

returns: nothing

Method Details

getDestPid

public int getDestPid()

Return the destination process ID.

returns: process ID.

getMode

public int getMode()

Return the mode of this channel, read or write.

returns: VC_READ or VC_WRITE.

getName

public String getName()

Return the name of this channel.

returns: name of this channel.

isAvailable

public boolean isAvailable()

Check whether a message for this channel has arrived. Return true is the message has arrived, otherwise return false

returns: true: message has arrived; false: message has not arrived.

readInt

public int readInt()

Read a integer. Blocks until the message has arrived.

returns: an integer.

readFloat

public float readFloat()

Read a float. Blocks until the message has arrived.

returns: a float.

readDouble

public double readDouble()

Read a double. Blocks until the message has arrived.

returns: a double.

readObject

public Object readObject()

Read an object. Blocks until the message has arrived.

returns: an object.

writeInt

public void writeInt(int vint)

Write an integer. Message might not have been sent or received.

vint: integer to be written

returns: nothing.

writeFloat

public void writeFloat(float vfloat)

Write a float. Message might not have been sent or received.

vfloat: float to be written

returns: nothing.

writeDouble

public void writeDouble(double vdouble)

Write a double. Message might not have been sent or received.

vdouble: double to be written

returns: nothing.

writeObject

public void writeObject (Object o)

Write an object. Message might not have been sent or received. A copy of the object is made by the system when this function returns. The desination thread will receive the copy of the object.

o: object to be written

returns: nothing.

VC_ChannelSet

Field Summary

Constructor Summary

void VC_ChannelSet (String t_channelSetName)

Method Summary

void addChannel (VC_Channel channel):   Add a channel to this channel set.

VC_Channel getChannel (String channelName): Return a channel by name.

String getName ():   Return the name of this channel set

void removeChannel(String channelName): Remove a channel from this channel set.

int size ():   Return the size (number of channels) of this channel set.

Field Details

Constructor Details

VC_ChannelSet

    public VC_ChannelSet(String t_channelSetName)

    Initialize a channel set.

    t_channelSetName: name of the channel

    returns: nothing

Method Details

getName

    public String getName()

    Return the name of this channel set.

returns: name of channel set.

size

public int size()

Return the size (number of channels) of this channel set.

returns: number of channels.

addChannel

public void addChannel(VC_Channel channel)

Add a channel to this channel set.

channel: a virtual channel

returns: nothing.

getChannel

public VC_Channel getChannel(String channelName)

Return a channel by name.

channelName: name of the channel

returns: a virtual channel.

removeChannel

public void removeChannel(String channelName)

Remove a channel from this channel set.

channelName: name of the channel

returns: nothing

VC_State

Field Summary

Constructor Summary

void VC_State ()

Method Summary

void clearMigrated ():	Set the state as not migrated.
boolean isMigrated ():	Return whether the state has migrated.

Field Details

Constructor Details

VC_State

public VC_State()

Initialize a virtual state.

returns: nothing

Method Details

clearMigrated ()

public void clearMigrated()

Set the state as not migrated.

returns: nothing.

isMigrated ()

public boolean isMigrated()

Return whether the state has migrated.

returns: true, if migrated; false if not migrated.

VC_Thread

Field Summary

Constructor Summary

void VC_Thread ()

Method Summary

void addChannelSet (VC_ChannelSet vcc): Add a channel set to this thread.

void addVCState (VC_State vcs): Associate a virtual state with this thread.

void finalize (): Finalize the computation.

VC_ChannelSet getChannelSet (String t_ChannelSetName): Get a channel set by name.

int getPid (): Returns the process ID of this thread.

int getTid (): Returns the thread ID of this thread in a process.

VCluster getVC (): Returns the VCluster instance this thread belongs to.

String getVCName (): Get the name of this thread in VCluster.

Object getVCState (): Get the virtual state associated with this thread.

void migrate (int destPid): Migreate this thread to a specified destition process

void setVCName (String t_name): Set the name of this thread in VCluster.

void VC_Println (String s): Print out a message at the master node.

Field Details

Constructor Details

VC_Thread

public VC_Thread()

Initialize a Virtual Thread.

returns: nothing

Method Details

setVCName

public void setVCName(String t_name)

Set the name of this thread in VCluster.

t_name: name.

returns: nothing.

getVCName

public String getVCName()

Get the name of this thread in VCluster.

returns: name.

getVC

public VCluster getVC()

Returns the VCluster instance this thread belongs to.

returns: a VCluster instance.

getPid

public int getPid()

Returns the process ID of this thread.

returns: process ID.

getTid

public int getTid()

Returns the thread ID of this thread in a process. This is not a globally unique thread id.

returns: thread ID.

addVCState

public void addVCState(VC_State vcs)

Associate a virtual state with this thread.

vcs: a virtual state

returns: nothing.

getVCState

public Object getVCState()

Get the virtual state associated with this thread.

returns: a virtual state.

addChannelSet

public void addChannelSet(VC_ChannelSet vcc)

Add a channel set to this thread.

vcc: a channel set

returns: nothing.

getChannelSet

public VC_ChannelSet getChannelSet(String t_ChannelSetName)

Get a channel set by name.

t_ChannelSetName: name of the channel set

returns: a channel set.

VC_Println

public void VC_Println(String s)

Print out a message at the master node.

s: the message to be pritned out.

returns: nothing.

finalize

public void finalize()

Finalize the computation. Must be called at the end of a thread.

returns: nothing.

migrate

public void migrate(int destPid)

Migreate this thread to a specified destition process. All information not save to the assocated virtual state will be lost.

destPid: ID of the destination process

returns: nothing.

LIST OF REFERENCES

[1] R. Buyya, *High Performance Cluster Computing: Architectures and Systems*, 1st edn., Prentice Hall Inc., 1999.

[2] M. Baker, R. Buyya. Cluster computing: the commodity supercomputer. *Software: Practice and Experience* May 1999; 29:6, 551-576.

[3] Top 500 Supercomputers, www.top500.org.

[4] T. Anderson, D. Culler, and D. Patterson. A Case for Networks of Workstations. *IEEE Micor*, Feb. 95. http://now.cs.berkeley.edu/

[5] M.A. Baker, G.C. Fox, and H.W. Yau. *Review of Cluster Management Software.* NHSE Review, May 1996, http://www.nhse.org/NHSEreview/CMS/

[6] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface,* MPI Press, 1995.

[7] C. Amza, A. L. Cox, S.Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. IEEE Computer, 29(2):18 ̇C28, February 1996.

[8] M. Campione, K. Walrath, A. Huml, *Java Tutorial, The: A Short Course on the Basics,* 3rd Edition, Addison Wesley Professional, 2000.

[9] Lindholm T, Yellin F. The Java Virtual Machine Specification (Sunsoft Java Series) (2nd edn). AddisonWesley Developers Press, 1999.

[10] Neffenger J. The Volano Report: Which Java platform is fastest, most scalable? A Java World exclusive!. http://www.javaworld.com/javaworld/jw-03-1999.

[11] Sun Microsystems, Inc. The Java HOTSPOT Virtual Machine. White Paper, September 2002.

[12] C. Mangione. Performance test show Java as fast as C++. Technical report, JavaWorld, 1998.

[13] http://www.debian.org/

[14] http://www.redhat.com/

[15] http://www.sun.com/software/solaris/

[16] Technical Committee on Operating Systems and Application Environments of the IEEE. Portable Operating System Interface (POSIX)-Part 1: System Application Program Interface (API), 1996. ANSI/IEEE Std. 1003.1, 1995 Edition, including 1003.1c: Amendment 2: Threads Extension [C Language].

[17] OpenMP Forum. OpenMP: A Proposed Industry Standard API for Shared Memory Programming. Technical report, October 1997.

[18] Javasoft, 'Java Remote Method Invocation - Distributed Computing for Java, a White Paper', http://java.sun.com/marketing/collateral/javarmi.html.

[19] Javasoft, 'Java Interface Definition Language', http://java.sun.com/products/jdk/1.2/docs/guide/idl/index.html.

[20] Javasoft, 'Java Object Serialization', http://java.sun.com/products/jdk/1.1/docs/guide/serialization/index.html.

[21] K. Li. Ivy: A shared virtual memory system for parallel computing. Proceedings of the International Conference on Parallel Processing (ICPP'88), The Pennsylvania State University, University Park, PA, August 1988;

[22] K. Li, P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems* 1989; 17:4, 321-359.

[23] Harry F. Jordan, Gita Alaghband, *Fundamentals of Parallel Processing*, Prentice Hall, 2003.

[24] Wenzhang Zhu, Cho-Li Wang and Francis C.M.Lau, ``JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support," IEEE Fourth International Conference on Cluster Computing (CLUSTER 2002) Chicago, USA - September 23-26, 2002, pp. 381-388, 2002

[25] M.J.M. Ma, C.L. Wang, F.C.M. Lau, ``JESSICA: Java-Enabled Single-System-Image Computing Architecture, Journal of Parallel and Distributed Computing, Vol. 60, No. 10, October 2000, pp. 1194-1222

[26] "cJVM: a Single System Image of a JVM on a Cluster" - International Conference on Parallel Processing 99 (ICPP 99), September 1999.

[27] W. Yu and A. L. Cox. Java/dsm: A platform for heterogeneous computing. Concurrency - Practice and Experience, 9(11):1213 ̇C1224, 1997.

[28] G. Antoniu, L. Bouge, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. The Hyperion system: Compiling multithreaded Java bytecode for distributed execution. Parallel Computing, 27(10):1279--1297, 2001.

[29] R. Veldema, R. A. F. Bhoedjang, and H. E. Bal. Jackal, a compiler based implementation of java for clusters of workstations. PPoPP 2001, 2001.

[30] Michael Factor, Assaf Schuster, Konstantin Shagin, "JavaSplit: A Runtime for Execution of Monolithic Java Programs on Heterogeneous Collections of Commodity

Workstations," *cluster*, p. 110, IEEE International Conference on Cluster Computing (CLUSTER'03), 2003.

[31] M. Baker, B. Carpenter, G. Fox, S. Ko, and S. Lim. mpi-Java: an Object-Oriented Java Interface to MPI. In 1st Int. Workshop on Java for Parallel and Distributed Computing (IPPS/SPDP 1999 Workshop), San Juan, Puerto Rico, Lecture Notes in Computer Science, volume 1586, pages 748 ̈C762. Springer, 1999.

[32] http://www.hpjava.org/elements.html

[33] S. Morin, I. Koren, and C. M. Krishna. JMPI: Implementing the Message Passing Standard in Java. In 4th Int. *Workshop on Java for Parallel and Distributed Computing (IPDPS2002 Workshop)*, Fort Lauderdale, FL, 118-123, 2002.

[34] M. Baker and B. Carpenter. MPJ: a Proposed Java Message Passing API and Environment for High Performance Computing. *In 2nd Int. Workshop on Java for Parallel and Distributed Computing (IPDPS 2000 Workshop)*, Cancun, Mexico, Lecture Notes in Computer Science, volume 1800, pages 552-559. Springer, 2000

[35] Adam J. Ferrari. JPVM: Network Parallel Computing in Java. *ACM Workshop on Java for High-Performance Network Computing*, 1998.

[36] A. Nelisse, J. Maassen, T. Kielmann, and H. Bal. CCJ: Object-Based Message Passing and Collective Communication in Java. Concurrency and Computation: Practice & Experience, 15(3-5):341C369, 2003.

[37] G. L. Taboada, J. Tourino, and R. Doallo. Performance Analysis of Java Message-Passing libraries on Fast Ethernet, Myrinet and SCI Clusters. *IEEE Fifth International Conference on Cluster Computing*, Hong Kong, China Dec. 2003.

[38] N. Stankovic and K. Zhang. An Evaluation of Java Implementations of Message-Passing. *Software - Practice and Experience*, 30(7):741 ̇C763, 2000.

[39] Technical Committee on Operating Systems and Application Environments of the IEEE. Portable Operating System Interface (POSIX)-Part 1: System Application Program Interface (API), 1996. ANSI/IEEE Std. 1003.1, 1995 Edition, including 1003.1c: Amendment 2: Threads Extension [C Language].

[40] OpenMP Forum. OpenMP: A Proposed Industry Standard API for Shared Memory Programming. Technical report, October 1997.

[41] F. Cai, S. Wu, L. Zhang, and Z. Tang. Parallel Program Performance Evaluation And Their Behavior Analysis on an OpenMP Cluster. *IEEE International Symposium on Cluster Computing and the Grid*, Chicago, Illinois, USA, 2004.

[42] Yoshinori Ojima, Mitsuhisa Sato, Hiroshi Harada, Yutaka Ishikawa, "Performance of Cluster-enabled OpenMP for the SCASH Software Distributed Shared Memory System," *ccgrid*, p. 450, 3rd International Symposium on Cluster Computing and the Grid, 2003.

[43] Nikolaos Drosinos, Nectarios Koziris, "Performance Comparison of Pure MPI vs Hybrid MPI-OpenMP Parallelization Models on SMP Clusters," *ipdps*, p. 15a, 18th International Parallel and Distributed Processing Symposium (IPDPS'04) - Papers, 2004.

[44] Thomas Rauber, Gudula Runger, Sven Trautmann, "A Distributed Hierarchical Programming Model for Heterogeneous Cluster of SMPs," *ipdps*, p. 165a, International Parallel and Distributed Processing Symposium (IPDPS'03), 2003.

[45] T. Boku, K. Itakura, S. Yoshikawa, M. Kondo and M. Sato. Performance Analysis of PC-CLUMP based on SMP-Bus Utilization. *Proceedings of WCBC'00 (Workshop on Cluster Based Computing 2000)*, Santa Fe, May 2000.

[46] Olsson, Ronald A., Keen, Aaron W. The JR Programming Language, Concurrent Programming in an Extended Java, Springer, 2004.

[47] Cilk-5.2 Reference Manual. Available on the website http://supertech.lcs.mit.edu/cilk.

[48] Distributed Cilk - Release 5.1 alpha 1. Available on the website http://supertech.lcs.mit.edu/cilk/release/distcilkS.1.html.

[49] L. Peng, W. F. Wong, M. D. Feng, and C. K. Yuen. SilkRoad: A Multithreaded Runtime System with Software Distributed Shared Memory for SMP Clusters. In Proc. Of the 2nd IEEE International Conference on Cluster Computing (CLUSTER2000), Nov. 2000.

[50] L. Peng,W. F.Wong, and C. K. Yuen. SilkRoad II: A Multi-Paradigm Runtime System for Cluster Computing. In Proc. of the 4nd IEEE International Conference on Cluster Computing (CLUSTER2002), Sept. 2002.

[51] Philip Hatcher, Mathew Reno, Gabriel Antoniu, Luc Bouge, "Cluster Computing with Java," *Computing in Science and Engineering*, vol. 07, no. 2, pp. 34-39, Mar/Apr, 2005.

[52] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. Dag-consistent distributed shared memory. In Proceedings of the 10th International Parallel Processing Symposium (IPPS), pages 132-141, Honolulu, Hawaii, Apr. 1996.

[53] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Consistency for Software Distributed Shared Memory. In Proceedings of the 19th Annual Symposium on Computer Architecture, pages 13-21, May 1992

[54] M. Philippsen and M. Zenger. JavaParty - transparent remote objects in Java. Concurrency: Practice and Experience, 9(11):1225--1242, 1997.

[55] Y. Sohda, H. Nakada, and S. Matsuoka. Implementation of a portable software DSM in Java. In Java Grande/ISOPE'01, pages 163--172, Palo Alto, CA USA, 2001

[56] Jameela Al-Jaroodi, Nader Mohamed, Hong Jiang, David Swanson, "An Agent-Based Infrastructure for Parallel Java on Heterogeneous Clusters," *cluster*, p. 19, IEEE International Conference on Cluster Computing (CLUSTER'02), 2002.

[57] Hayzelden and Bigham, "Software Agents for Future Communication Systems", Springer-Verlag Berlin Heidelberg, 1999

[58] Geoffroy Vallée, Christine Morin, Renaud Lottiaux, Jean-Yves Berthou, Ivan Dutka Malen, "Process Migration Based on Gobelins Distributed Shared Memory," *ccgrid*, p. 325, 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02), 2002.

[59] Min Choi, DaeWoo Lee, and SeungRyoul Maeng, "Cluster Computing Environment Supporting Single System Image", IEEE International Conference on Cluster Computing (CLUSTER'04), 2004

[60] Ricky K. K. Ma, Cho-Li. Wang, Francis C.M. Lau, "M-JavaMPI: A Java-MPI Binding with Process Migration Support," *ccgrid*, p. 255, 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'02), 2002.

[61] M.J.M. Ma, C.L. Wang, and F.C.M. Lau; ``Delta Execution: A Preemptive Java Thread Migration Mechanism, ¡¯¡¯International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA ¡¯99), pp. 518-524, June 28 ¨C July 1, 1999, Las Vegas, Nevada, USA.

[62] Wenzhang Zhu, Cho-Li Wang, Francis C. M. Lau, "Lightweight Transparent Java Thread Migration for Distributed JVM," *icpp*, p. 465, 2003 International Conference on Parallel Processing (ICPP'03), 2003.

[63] Wenzhang Zhu, Weijian Fang, Cho-Li Wang, and Francis C.M. Lau, A New Transparent Java Thread Migration System Using Just-in-Time Recompilation, The 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2004), pp. 766-771, MIT Cambridge, MA, USA, November 9-11, 2004

[64] M.U. Janjua, M. Yasin, F. Sher, K. Awan, I. Hassan, "CEJVM: "Cluster Enabled Java Virtual Machine"," *cluster*, p. 389, IEEE International Conference on Cluster Computing (CLUSTER'02), 2002.

[65] http://java.sun.com/j2se/1.3/docs/guide/jpda/jvmdi-spec.html

[66] http://java.sun.com/j2se/1.3/docs/guide/jni/

[67] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience* Dec. 1990; 2:4, 315-339.

[68] J. Watt, S. Taylor, S. Nilpanich. SCPlib: A Concurrent Programming Library for Programming Heterogeneous Networks of Computers. *IEEE Information Technology Conference, EX 228, pp 153-6*, 1998

[69] J. Lee, S. Chapin, S. Taylor. Computational Resiliency. *Journal of Quality and reliability Engineering International*, 18:3, pp185-199, 2002

[70] MPICH-A Portable Implementation of MPI, http://www-unix.mcs.anl.gov/mpi/mpich

[71] http://en.wikipedia.org/wiki/Rsh

[72] http://en.wikipedia.org/wiki/Ssh

[73] Dawn Song, David Wagner, Xuqing Tian. Timing Analysis of Keystrokes and Timing Attacks on SSH. Usenix Security Symposium 2001.

[74] Weiwu Hu Gang Shi, Fuxin Zhang, "Communication with Threads in Software DSMs," *cluster*, p. 149, 3rd IEEE International Conference on Cluster Computing (CLUSTER'01), 2001.

[75] http://java.sun.com/j2se/1.4.2/docs/guide/nio/

[76] R., D. E., G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. Nature, vol 323, pp533-536, 1986.

[77] Z., S., W., M., Klauer, B., W., K. Pipelining and parallel training of neural networks on distributed-memory multiprocessors. IEEE World Congress on Computational Intelligence, 4(27), June 1994, 2052-2057.

[78] N., T. and B. Svensson. Using and designing massively parallel computers for artificial neural networks. Journal of Parallel and Distributed Computing, 14 (3), 1992, 260-285.

[79] Louisiana State University, http://sunmp.cclms.lsu.edu/cclms/teaching/csc7610/

[80] Jr. Wagner Meira. Modeling Performance of Parallel Programs. Technical report 589, University of Rochester, Computer Science Department, Rochester, New York 14627, June 1995.

[81] http://www.cs.uoi.gr/~ompi/

[82] J. Gomez, V. Rego, and V. Sunderam, "CLAM: Connectionless, Lightweight, and Multiway Communication Support for Distributed Computing," Lecture Notes in Computer Science: Communication and Architectural Support for Network-Based Parallel Computing, pp. 227-240. Springer-Verlag, 1997.

[83] Juan Carlos Gomez, Edward Mascarenhas, Vernon Rego, "The CLAM Approach to Multithreaded Communication on Shared-Memory Multiprocessors: Design and Experiments," IEEE Transactions on Parallel and Distributed Systems, vol. 09, no. 1, pp. 36-49, Jan., 1998.

[84] E. Mascarenhas and V. Rego, “Ariadne: Architecture of a Portable Threads System Supporting Thread Migration,” Software–Practice and Experience, vol. 26, no. 3, pp. 327-357, Mar. 1996.

[85] Kruskal, C. and Weiss, A. Allocating independent subtasks on parallel processors, IEEE Trans. Softw. Eng., SE-11, 10 (Oct. 1985).

[86] S.F. Hummel, E. Schonberg, L.E. Flynn, Factoring: a method for scheduling parallel loops, Communications of the ACM 35 (8) (1992) 90–101.

[87] Flynn, L.E. and Hummel, S.F. Scheduling variable-length parallel subtasks. IBM Research Report RC 15492, Feb. 1990.

[88] Polychronopoulos, C. and Kuck, D. Guided self-scheduling: A practical scheduling scheme for parallel computers. IEEE Trans. Comput. C-36, 12 (Dec. 1987), 1425-1439.

[89] S.F. Hummel, J. Schmidt, R.N. Uma, J. Wein, Load-sharing in heterogeneous systems via weighted factoring, in: Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures, 1996, pp. 318–328.

[90] I. Banicescu, V. Velusamy, Load balancing highly irregular computations with the adaptive factoring, in: Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2002)—Heterogeneous Computing Workshop, IEEE Computer Society Press, 2002, p. (CDROM).

[91] I. Banicescu, V. Velusamy, J. Devaprasad, On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring, Cluster Computing: The Journal of Networks, Software Tools and Applications 6 (3) (2003) 215–226.

[92] J. Chen, V.E. Taylor, Mesh partitioning for distributed systems: exploring optimal number of partitions with local and remote communication, in: Proceedings of 9th SIAM Conference on Parallel Processing for Scientific Computing, 1999, p. (CDROM).

[93] A. Sohn, H. Simon, S-harp: a scalable parallel dynamic partitioner for adaptive mesh-based computations, in: Supercomputing 98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, IEEE Computer Society, 1998, p. (CDROM).

[94] I. Banicescu, R. L. Cari ño, J. P. Pabico, M. Balasubramaniam, Design and Implementation of A Novel Dynamic Load Balancing Library for Cluster Computing, Journal of Parallel Computing, Vol. 31, Issue 7, Pg. 736-756, July 2005

[95] B. Dimitrov and V. Rego. Arachne: A portable threads system supporting migrant threads on heterogeneous network farms. IEEE Concurrency Magazine, 7(1):60-69, January 1999.

[96] A. Itzkovitz, A. Schuster, and L. Shalev. Thread migration and its applications in distributed shared memory systems. Systems and Software, 42(1):71-87, 1998.

[97] T. Liang, C. Shieh, J. Li, Selecting Threads for Workload Migration in Software Distributed Shared Memory Systems, Journal of Parallel Computing, Vol. 28, Issue 6, Pg. 893-913, June 2002

[98] T.T.Y. Suen, J.S.K. Wong, Efficient Task Migration Algorithm for Distributed Systems, IEEE Transactions on Parallel and Distributed Systems, vol. 03, no. 4, pp. 488-499, Jul., 1992.

[99] F. O. Lucchese, E. J. Huerta Yero, F. S. Sambatti, and M. A. A. Henriques, An Adaptive Scheduler for Grid, Journal of Grid Computing, vol. 04, no. 1, pp. 1-17, March, 2006

[100] F. Desprez, and A. Vernois, Simultaneous Scheduling of Replication and Computation for Data-Intensive Applications on the Grid, Journal of Grid Computing, vol. 04, no. 1, pp. 19-31, March, 2006

[101] S. Huang, E. Aubanel, and V. C. Bhavsar, PaGrid: A Mesh Partitioner for Computational Grids, Journal of Grid Computing, vol. 04, no. 1, pp. 71-88, March, 2006

[102] G. W. Finger, J. Kapat, and A. Bhattacharya, Analysis of Tangential Momentum Accommodation Coefficient Using Molecular Dynamics Simulation. Microfluidics Session at 44th AIAA Aerospace Sciences Meeting & Exhibit, 2006

[103] http://www.centos.org/

[104] Stankovic, N., and Zhang, K., A Distributed Parallel Programming Framework, IEEE Transactions on Software Engineering, 28(5), May 2002, pp.478-493