Message Passing Library for Java

Rajesh Babu

August 21, 2009

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2009
Abstract

Message Passing Interface (MPI) is the standard message passing API for parallel computing. The original standard defines bindings for C, FORTRAN and C++. Several research efforts have extended the MPI standard for Java. But none of them really performed well enough compared to standard MPI implementations. In this project we designed a new message passing library for Java, called MPLJava, taking some of the best practices followed in the existing libraries and improvising them with the latest improvements in Java technology. We discuss the architecture of our implementation, its performance compared to standard MPI implementations, and propose some future research direction.
Contents

Chapter 1 Introduction........................................................................................................1

Chapter 2 Background........................................................................................................2
  2.1 Message Passing Model ...........................................................................................2
  2.2 Java for Message Passing Model .............................................................................5
    2.2.1 Single-JVM Programming Models .................................................................6
    2.2.2 Multi-JVM Programming Models ..................................................................6
    2.2.3 Existing Libraries ............................................................................................9
    2.2.4 Problems with the Existing Library ................................................................10

Chapter 3 Design and Implementation.............................................................................11
  3.1 Desired features ......................................................................................................11
  3.2 System Architecture ...............................................................................................13
    3.2.1 Task Manager ..................................................................................................14
    3.2.2 MPLJava Client API .......................................................................................15
    3.2.3 Queuing Mechanism .......................................................................................16
    3.2.4 Abstract Device Interface ...............................................................................18
    3.2.5 Datatypes .........................................................................................................24
  3.3 Workflow Diagram .................................................................................................26

Chapter 4 Testing and Performance.................................................................................28
  4.1 HPCx .......................................................................................................................28
  4.2 Correctness Test .....................................................................................................29
  4.3 Performance Test ...................................................................................................30
    4.3.1 Latency Test ....................................................................................................30
List of Tables

Table 1 - Latency ..............................................................................................................31
Table 2 - Inter-Node Latency with only IP communication............................................31
Table 3 - Initial Risk Assessment.....................................................................................46
List of Figures

Figure 1 - MPLJava System Architecture........................................................................13
Figure 2 – code snippet of MatchMaker ........................................................................17
Figure 3 - Connector Interface .........................................................................................18
Figure 4 – creation of Selector and ServerSocketChannel ..............................................20
Figure 5 - Select Loop ......................................................................................................21
Figure 6 - create and register Read-SocketChannel.........................................................21
Figure 7 - create Write-SocketChannel...........................................................................21
Figure 8 - register Write-SocketChannel .......................................................................22
Figure 9 - pseudo code of write routine .........................................................................22
Figure 10 - pseudo code of read routine .........................................................................23
Figure 11 - IntegerArray datatype ..................................................................................25
Figure 12 - using IntegerArray .......................................................................................25
Figure 13 - Workflow Diagram .......................................................................................27
Figure 14 – Correctness Test – Tasks in a Ring ...............................................................29
Figure 15 - pseudo code of ring program to compute global-sum of ranks .................29
Figure 16 - pseudo code of pingpong ............................................................................30
Figure 17 - MPLJava Shared Memory Connector Bandwidth ..........................................32
Figure 18- MPLJava Shared Memory Connector Bandwidth after Improvement .........33
Figure 19 - MatchMaker code snippet with ReentrantLock ..........................................33
Figure 20 – Singlepair Intra-Node Bandwidth ...............................................................34
Figure 21 - Multipair Intra-Node Bandwidth .................................................................34
Figure 22 - SinglePair Inter-Node Bandwidth ...............................................................35
Acknowledgements

I would like to thank my supervisor Dr. Stephen Booth for his advice and assistance throughout this project. I would also like to thank Dr. David Henty for supporting the project in the absence of Stephen.
Chapter 1

Introduction

MPI (1) is the standard message passing API for parallel computing. The original standard defines bindings for C, FORTRAN and C++ languages. As Java started becoming popular there was immediate interest in exploring its possible uses for parallel computing. As a result numerous MPI like Java bindings have been developed since late nineties. The general consensus is that the performance of Java implementations is much slower than standard MPI implementations. However, work performed in this domain has been performed on primitive Java versions and on relatively outdated machines.

The aim of this project is to design a new message passing library in Java for high performance computing with a competitive performance compared to standard MPI implementations. Java technology has seen revolutionary changes in the last decade. The project experiments various approaches to build an efficient library. MPI specification defines extensive message passing concepts. Being an experimental project, implementing all the functionalities makes no sense unless the library is scalable and is performing well. Hence the scope of the dissertation is bound to implementing point-to-point API for shared memory communication and IP based communication, and compare its performance with standard MPI implementations and some of the existing libraries like mpiJava.

This report is structured in the following sequence,

In Chapter 2 we describe some background knowledge of Message Passing Model and the various Java technologies that can be used to implement the Message Passing Model.

In Chapter 3 we propose the design and explain the system architecture, implementation and workflow.

Chapter 4 discuss the testing and performance results of the library.

Chapter 5 proposes some future extensions to improve the performance of the new library.

Chapter 6 concludes the report.
Chapter 2

Background

In this chapter we describe some background knowledge of Message Passing Model and the various Java technologies that can be used to implement the Message Passing Model.

2.1 Message Passing Model

Programming models can be categorized into two based on how memory is used. In the shared memory model each process accesses a shared address space, while in the message passing model an application runs as a collection of autonomous processes. In the message passing model symmetric processes communicate with other by sending and receiving messages. This symmetric model of communication is captured in the successful Message Passing Interface standard (MPI). MPI directly supports the Single Program Multiple Data (SPMD) model of parallel computing, wherein a group of tasks cooperate by executing identical copies of program on local data values. The MPI standard defines a set of functions that support Point-to-Point Communication, Collective Communication, Process groups, Process topologies and Datatype manipulation. The following sections explain some of the important concepts that might be used in this report.

Point-to-Point Communication

Point to Point Communication involves sending and receiving messages between two tasks. This is the simplest form of data transfer in a message passing model. The performance of point to point communication is generally measured in terms of latency and bandwidth. MPI supports multiple types of send routines like buffered, synchronous, ready and standard. Buffered-Send completes locally whether or not a matching receive has been posted. The data is ‘buffered’ somewhere until the receive is posted. Synchronous-Send can only complete when the matching receive has been posted. Ready-Send can only be started if the receiver has already posted the receive operation. Standard-Send can be either buffered or synchronous. MPI also defines blocking and non-blocking communication. A blocking subroutine returns only when the operation has completed. A non-blocking subroutine returns straight away and allows the program to continue with other work. At a later point the program can wait
for the completion of the non-blocking operation. Generally blocking communication is implemented on top of non-blocking communication. All these operations must also ensure messages are correctly ordered.

**Collective Communication**

Collective communication is a coordinated operation among multiple tasks. All tasks call the same operation with the same arguments. Thus when sending and receiving messages, one collective operation can replace multiple sends and receives, resulting in lower overhead and higher performance. The basic collective operations are `BARRIER` (for process synchronization), `BCAST` (for broadcasting a message from one process to multiple processes), `REDUCE` (to combine data from several processes to produce a single result), `SCATTER` (to send different portion of a single message to different processes) and `GATHER` (reverse of scatter). Collective routines are often built on top of point to point communication. These routines can simply be implemented by building a binary tree of processes where broadcast operations can be passed down the tree and reduction operation can be sent up the tree with partial combine at each step.

**Communicators**

A Communicator is an object that encapsulates a group of processes such that communication is restricted to processes within that group. Communicators that allow processes within a group to exchange data are termed as Intra-Communicators. Communicators that allow processes in two different groups to exchange data are called Inter-Communicators.

**Messaging Protocols**

All MPI Implementations need a mechanism for delivering messages to remote process. Whenever a process sends a MPI message to a remote process a corresponding initial protocol message (IPM) must be sent. This message should minimally contain the envelope information and may also contain some data.

MPI implementations use different underlying protocol depending on the size of the message. The basic protocol types are

**Eager Protocol**

In eager protocol IPM contains the full data of the corresponding message. If there is no matching receive posted when IPM arrives, then data must be buffered at the receiver.

**Rendezvous Protocol**

In rendezvous protocol IPM only contains the envelope information and no data. When an IPM is matched to a receive, a ready-to-send protocol message is returned to the sender. Sending process then sends the data in a new message. Send acts as a synchronous send unless the message is buffered on the sending process. For large
messages where receive is posted late, rendezvous can be faster than eager because the IPM will take less time than copying the data from the receive side buffer.

**Message Queues**

If the receiving process has already issued a matching receive the message can be processed immediately. If not then the message must be stored in a foreign send queue for future processing. Similarly a receive call looks for matching messages in the foreign send queue. If no matching message is found then receive message is stored in a receive queue. There could be many such queues for different communicators and/or senders. But mostly there is a single global queue which makes wildcard receive much simpler.
2.2 Java for Message Passing Model

Java has become increasingly popular as a general purpose programming language. With inbuilt mechanisms to handle concurrency and communication, such as threads and sockets, Java suits the needs of parallel computing. However, Java is not so popular in scientific and parallel community because of two main reasons; numerical problems, such as precision error, floating point reproducibility problem etc. and poor performance of Java compared to C or C++. The numerical problems in Java are already a known issue which is tracked by the Numerical Working Group of Java Grande Forum (8). Java technology has seen revolutionary changes in the last decade to improve the performance. To understand why Java is considered to be performing poor, we should understand how a Java program works.

The Java Virtual Machine (JVM) is the crucial component of Java Platform. The JVM is the instance of the Java Runtime Environment (JRE) which comes into action whenever a Java program is executed. A Java compiler compiles the Java source code and generates the Java bytecode. The JVM interprets the Java bytecode at runtime and generates the native instructions. The same bytecode can be used for all platforms, which makes Java a portable language. A vendor writes a JVM for their operating system.

Apart from being an Interpreter, The JVM can also have a Just-In-Time compiler. Most JVM include a JIT compiler which will compile heavily used code of the application into machine code at runtime. These parts will then typically run much faster than if they were simply interpreted. Java has an internal memory management system. The memory management does the allocation of memory through an Allocator, and collecting used objects through a Garbage Collector. When JVM cannot allocate an object from the current heap the garbage collector is invoked. Hence the JVM by itself is a heavy process which might consume lot of CPU and memory that are crucial parameters for High Performance Computing.

In order to design a message passing library in Java, it is essential to understand how SPMD model can be implemented in Java in a single-JVM environment and multiple-JVM environment (17). A single-JVM environment provides the infrastructure for running a single multi-threaded Java application. In a multiprocessor system the Java threads might be distributed across the processors. A multiple-JVM environment provides the infrastructure for running multiple JVMs, each capable of running a multi-threaded Java application, with some form of communication mechanism allowing interaction between the applications running on different JVMs. Multiple JVMs can coexist in the same system in the same memory. In a multiprocessor system the JVMs might be distributed across the processors similar to Java threads.
2.2.1 Single-JVM Programming Models

Java Threads

For single-JVM environment Java threads provide an appropriate mechanism for parallel programming, since threads are part of the standard Java libraries. Most current JVMs implement the thread class on top of the native OS threads. This means that the threads are visible to the operating system and are capable of being scheduled on different processors. For a SPMD model implemented in a single-JVM, communication between the threads running the tasks can be done via the shared address space. But careful measures must be taken to handle the concurrency issues that may arise when multiple threads try to access a shared data. The language offers mechanism to fork and join threads, and to synchronize the use of shared objects. From JDK 1.5 there is a mechanism in Java called CyclicBarrier that allows a set of threads to all wait for each other to reach a common barrier point. The Executor Framework can be used to maintain a thread pool and distribute the task to multiple threads. The Callable and Future interfaces can be used to define the task and return the results respectively. From JDK 1.5 Java also offers a rich set of concurrency mechanism to handle operations atomically. Some of the famous concurrency classes are ReadWriteLock, CountdownLatch, ConcurrentMap, CopyOnWriteList etc.

2.2.2 Multi-JVM Programming Models

A multi-JVM environment requires some mechanism for inter-JVM communication. Even if multiple JVMs are located in the same memory there are no direct mechanisms for them to communicate with each other via shared memory. The various approaches to provide inter-JVM communication are:

1. Use Java Native Interface (JNI) to use the existing standard MPI implementations in C or Fortran
2. To choose a readymade pure Java solution like Remote Method Invocation (RMI)
3. To use core Java API such as java.net, java.io and java.nio libraries as underlying communication layer.

RMI

Java RMI supports seamless remote method invocation on objects in different JVMs on a network. It is the programmer’s responsibility to provide the appropriate environment for the remote method to be invoked. RMI uses Java serialization for marshalling and un-marshalling of objects as streams. Serialization is the process of saving the current state of an object to a stream, and restoring an equivalent object from the stream. The fact that both the object and the methods manipulating the object can be sent over a network is an extremely powerful feature of RMI. When you invoke a method on a remote machine, the stubs and skeleton are used in the invocation. The stubs and skeletons are local code that serves as a proxy for the remote object. Fortunately, the RMI compiler takes care of creating the stubs and skeletons.
There are several drawbacks in using RMI for Message Passing Model in High Performance Computing. First, RMI was created to be used in a client-server paradigm in which a server application creates a number of objects and a client application then invokes methods on these objects remotely. The message passing model requires passing of data between processes, which is then processed locally, rather than invoking computation on remote data. Second, using RMI simply to pass data between multiple JVMs requires creation of additional threads to manage remote requests, which might incur context switching and synchronization overheads. Third, the serialization process used by RMI might be an inefficient solution for scientific computing where we are mostly interested in sending only the data rather than the entire object which unnecessarily increase the size of the message. Thus, RMI is a costly process to run on each and every node, resulting in unacceptable latency and resource consumption. Hence using RMI doesn’t seem to be a good solution for message passing model in high performance computing.

**Java Native Interface**

Java Native Interface (JNI) is a mechanism built into the JVM so as to invoke local system calls to perform input and output, graphics, networking, and threading operations on the host operating system. JNI can be used to provide wrappers for native MPI implementations, thereby enabling Java processes in different JVM to communicate with each other.

Though using JNI wrappers seem to be a simple and easy solution, there are several drawbacks. In a JNI implementation it is not possible to use MPI operations to communicate between Java threads because it is not safe to run more than one thread in a single JVM to perform MPI operation. This is because native implementations of MPI might not be thread-safe. Moreover, JNI calls inhibit JIT compilation because JIT compiler doesn’t have any control over the invoked native code. This can have an adverse impact on performance.

**Socket**

Java provides built-in support for lower level network communication. The `java.net` library provides a class, `Socket`, which implements one side of a two-way connection between a Java program and another program on the network. The `Socket` class sits on top of a platform-dependent implementation, hiding the details of any particular system from the Java program. By using the `java.net.Socket` class instead of relying on native code, Java programs can communicate over the network in a platform-independent fashion. However, the `java.io` and `java.net` libraries perform well enough for client-server codes based on Remote Method Invocation (RMI) in a WAN environment. The performance of these libraries is not suitable, however, for high performance communication in a LAN environment due to several key inefficiencies listed below.

1. The `java.io` library lacks a mechanism for a single thread to poll multiple sockets, and the ability to make non-blocking I/O requests on a socket. The workaround solution is to use a separate thread to poll each socket. This
introduces unacceptable overhead for a high performance application and simply does not scale well as the number of sockets increases.

2. The `java.io` operations work out of an array of bytes allocated in the Java heap. Java cannot pass references to arrays allocated in the Java heap to system-level operations, because objects in the Java heap can be moved by the garbage collector. Instead, another array might be allocated in the C heap and the data is copied back and forth \(^{(11)}\). Alternatively, to avoid this extra overhead, some JVM implementations might protect the byte array from garbage collection during system level operations.

**Java New I/O API**

The `java.nio` library, introduced in JDK 1.4, implements the new I/O APIs for the Java Platform providing a new set of abstractions for doing I/O. One of the most important aspects of `java.nio` library is its ability to operate in non-blocking mode, denied by the traditional `java.io` library. The following components in `java.nio` library make it an ideal candidate to implement message passing model over IP in Java.

1. **Buffers**

Starting from the simplest, the first improvement provided by `java.nio` library is the set of Buffer classes like `ByteBuffer`, `IntBuffer`, `FloatBuffer`, `DoubleBuffer` etc. These buffers provide a mechanism to store a set of primitive data elements in an in-memory container. They allow direct copies of primitives to/from buffers. The `ByteBuffer` can be allocated as direct buffer which are allocated in the C heap and therefore not subject to garbage collection. This allows I/O operations with no more copying than which is required by the operating system for any programming language.

2. **SocketChannel**

Channels are gateways through which I/O transfers take place, and buffers are the sources or targets of those data transfers. A Channel is like a tube that transports data efficiently between byte buffers and the entity on the other end of the channel. Some of the important Channels are `SocketChannel` and `ServerSocketChannel`. `SocketChannel` can read and write, while `ServerSocketChannel` listen for incoming connection and create new `SocketChannel`. All the `SocketChannels` create a peer `Socket` object when they are instantiated. The peer socket can be obtained from a channel by invoking its `socket` method. While every `SocketChannel` has an associated `java.net.Socket` object, not all sockets have an associated channel. If you create a `Socket` object in the traditional way, by instantiating it directly, it will not have an associated `SocketChannel`. `SocketChannels` can operate in non-blocking mode using `Selector` object.

3. **Selector**

It is no longer necessary to dedicate a thread to each socket connection. The `Selector` object enables a single thread to manage hundreds or even thousands of active socket connections with little or no performance loss. `Selectors` provide the ability to have a channel readiness selection, which enables multiplexed I/O.
Implementation of Java New I/O API differs for different applications based on number of threads to use. In the design chapter we have shown how java.nio is implemented for our library.

It is worth noting that usage of core Java APIs for underlying communication layer, such as java.net and java.nio implies the use of IP based communication. Hence for a closely-coupled parallel system with some fast processor interconnect, only a native MPI implementation is appropriate for performance and architectural reasons.

2.2.3 Existing Libraries

Existing approaches to MPI for Java can be grouped into two types:

1. Native MPI bindings where some native MPI library is called by Java programs through JNI
2. Pure Java implementation approach

mpiJava (2) is the most active Java wrapper project, provides an object oriented interface to standard MPI called MPJ. It consists of a collection of wrapper classes with C++ like interface that call a native MPI implementation through JNI.

JavaMPI (3) is a comparable approach with mpiJava. JavaMPI wrappers were automatically generated from C MPI header by a special purpose code generator. This eases the implementation work, but does not lead to fully object based API because it is very close to the C binding.

M-JavaMPI (4) is another wrapper approach with process migration support. Unlike mpiJava and JavaMPI, it does not use direct binding of Java programs and MPI. M-JavaMPI follows a client-server message redirection model that makes the system more portable, that is, MPI implementation-independent. It was not publicly available.

JMPI (5) is a pure Java implementation based on RMI following the mpiJava specification. It was developed for academic purposes at the University of Massachusetts.

CCJ (6) is a pure Java implementation with an MPI like syntax. It makes use of Java capabilities such as a thread based programming model or sending of objects.

MPIJ (2) is a pure Java MPI subset developed as part of the DOGMA project. MPIJ implements a large subject of MPI functionality except virtual topologies and user-defined datatypes. Objects must be manually serialized before communicated as a stream of bytes. To achieve better performance native types are first marshalled into Java byte array, and then sent over a TCP/IP channel.

MPJ Express (7) is the only acceptable pure Java implementation. It follows the mpiJava API specification. MPJ Express provides a very generic architecture with a framework to connect to any underlying communication layer. It already has interface to communicate through high speed interconnect such as Gigabit Ethernet and Myrinet. However, the architecture of MPJ Express is such that it creates a separate JVM for
each task. Hence it is not a scalable solution for SMP cluster because the overhead of running a separate JVM for each task in a shared memory node is much greater than running multiple tasks in conventional languages like C or FORTRAN. For Java we therefore require a mechanism that allows multiple processes to be run within the same JVM.

### 2.2.4 Problems with the Existing Library

Here we consolidate all the drawbacks in the existing libraries and justify the purpose to build a new message passing library for Java

- In a JNI implementation it is not possible to use MPI operations to communicate between java threads because it is not safe to run more than one thread in a single JVM to perform MPI operation. This is because native implementations of MPI might not be thread-safe.
- JNI calls inhibit JIT compilation because JIT compiler has no control over the invoked native code. This can have an adverse impact on performance.
- The C++ MPI bindings are very complex and much of the complexity like derived data-types is irrelevant to a strongly OO language like Java. We require a much simpler API.
- Running a pure Java implementation based on RMI is not an ideal solution for low level message passing. RMI’s client-server paradigm doesn’t suit the message passing model. Moreover, RMI is a costly process with unacceptable latency and resource consumption.
- Existing pure Java implementations are not suitable for SMP node because they create separate JVM for each task running in the same node. The overhead of running a JVM per task is much greater than running the task in conventional languages like C or FORTRAN. For Java we therefore require a library that allows multiple tasks running in the same node to use a single JVM. Also many of the existing pure Java implementations were intended for heterogeneous environment, such as grid computing, where one can take advantage of Java’s portability and security features however in a symmetric multiprocessor these features are going to be an overhead.

Considering the above problems and the absence of a good performing message passing library for Java is the motivation behind this project.
Chapter 3

Design and Implementation

The existing libraries typically followed either the JNI approach or the pure Java approach. The advancement in Java technology has enabled networking applications written in Java to rival their C counterparts at least for Ethernet. On the other hand, improvements in specialized networking hardware have continued, cutting down the communication cost to a couple of microseconds. However, most of these network hardware’s currently do not support Java directly. Keeping both in mind, the key issue at present is not to debate the pure Java approach versus the JNI approach, but to provide a flexible mechanism for applications to use different communication platforms.

The new Message Passing Library for Java (henceforth termed as MPLJava) proposed in this chapter provides a generic and extensible architecture to integrate different communication platforms.

3.1 Desired features

The proposed message passing library for Java should have the following features or at least should be capable of supporting them in the future.

Pure Java Implementation

The core library should be written in pure Java. This allows runtime JVMs such as HotSpot JVM to improve the performance by combining interpretation and JIT compilation. A pure Java implementation allows JIT compiler to perform runtime optimizations. It also allows HotSpot JVM to apply many optimization techniques such as inline expansion, loop unwinding, bounds-checking elimination, procedural analysis, and architecture dependent register allocation\(^{(18)}\).

Single JVM

Perhaps this is the most unique feature of the proposed library. The library should use a single JVM per node to run multiple tasks. A single JVM per node implementation allows the library to minimize the resource consumption by using a single virtual machine. It also enables the library to schedule tasks in multiple threads which in turn...
allow fast synchronization among tasks and fast communication between tasks via shared address space.

**Support multiple interconnects**

The proposed library should provide a flexible mechanism to use different communication platforms such as Ethernet, Infiniband, Myrinet etc. We can write a pure Java based connector for interconnects such as Ethernet and JNI based connector for non-Java compliant interconnects like Infiniband and special interconnects as in BlueGene. The library should be capable of integrating different interconnects through simple configuration.

**Integration with existing Job Submission Tools**

The library should be easy to run on any existing Job Submission Tools. The library should provide a generic configuration mechanism to integrate with any existing system and should not overwhelm the programmer with too many configurations.

**Queuing Mechanism**

The library should provide a well optimized queuing mechanism to buffer data wherever required. Proper care should be taken to cleanup used objects and handle concurrency issues.

**Datatype**

We saw that serialization process is a costly operation both in terms of resource consumption and increase in message size. The library should provide a mechanism to represent the data in the simplest form.
3.2 System Architecture

MPLJava offers a layered design that allows incremental development. It adheres to the Single Program Multiple Data (SPMD) model used by MPI. The system architectures consists of the following components,

1. Task Manager
2. Client API
3. Queuing Mechanism
4. Abstract Device Interface
5. Datatypes

The following diagram shows the layered design of MPLJava.

![MPLJava System Architecture Diagram]

Figure 1 - MPLJava System Architecture
3.2.1 Task Manager

Task Manager facilitates use of Single-JVM for all the tasks running in the same processing node. For example, in machines like HPCx where each processing node has 16 processors, MPLJava creates only one JVM per node common to all 16 processors. MPLJava accepts the fully qualified name of the task to be executed as a command line argument. Any arguments to the task can also be sent as command line argument following the task name. The total number of tasks is set in a system environment variable, MPLJAVA_TOTAL_PROCESSES. The library also uses a Config.properties file to specify the default values required by the library, such as the environment variable names used by the library, default IP port and default maximum size of eager protocol message.

Job Submission

MPLJava can be seamlessly integrated with any Job Submission Tool. But unlike a normal job submission, where the user specifies the total number of tasks in the job submission script, here the user has to specify the total number of nodes in the job submission script. The Task Manager of the MPLJava identifies the number of tasks to be executed in each node based on the total number of tasks entered in the MPLJAVA_TOTAL_PROCESSES environment variable and the total number of nodes entered in the Job Submission Script. For example, if we wanted to run a standard MPI program in 32 CPU’s in HPCx, we specify the CPUs=32 and Tasks_Per_Node=16. For MPLJava we specify CPUs=2, Tasks_Per_Node=1 and MPLJAVA_TOTAL_PROCESSES=32. A detailed example of how to run jobs using MPLJava is shown in Appendix A.

The Task Manager then creates a thread for each task and delegates a copy of the task to each thread.

Address

In order to provide a generic mechanism to identify nodes, MPLJava creates an abstract unique object called Address. The Address object abstracts all the information required to establish a connection with the node. The current implementation of MPLJava’s Address contains IP-Address and Port details for TCP/IP connection. In future when the library is extended for different interconnects such as Infiniband we can add Infiniband specific parameters to Address object. The Task Manager reads the node information from system variables, set in the job submission script, and creates the Address object. MPLJava provides a factory mechanism called HostnameModifier to identify the interconnect to be used. A DefaultHostnameModifier is provided to retrieve the default IP based connection. If we wish to use a different interconnect we can create a new class by extending the HostnameModifier and add the new class to the factory mechanism. Each HostnameModifier class is associated with a machine name. If we do not wish to use the default interconnect we can request the library to use a different HostnameModifier by specifying the corresponding machine name in the MPLJAVA_MACHINE_NAME environment variable. For HPCx the
DefaultHostnameModifier retrieves a slow Ethernet connection. In order to use the fast Ethernet connection we wrote a new class called HpcxHostnameModifier and specified the corresponding machine name, HPCx, in the MPLJAVA_MACHINE_NAME environment variable.

### 3.2.2 MPLJava Client API

MPLJava Client API consists of message passing programming interface for point-to-point communication. Any message passing operation can be invoked by calling MPI.xxx where xxx is the operation. This is just to keep the interface as simple as possible for existing MPI programmers. The initial set of APIs consists of MPI.send and MPI.recv for blocking communication, MPI.isend and MPI.irecv for non-blocking communication, Comm.getMpiCommWorld, Comm.getRank and Comm.getSize to retrieve the basic set of parameters of a MPI task, Request.iwait for a task to wait for non-blocking communication to complete, basic set of datatypes such as IntArray, FloatArray and DoubleArray and a interface Task to identify the task. We can design an object oriented approach by encapsulating these operations inside the communicator object, as done by mpiJava API specification, in the future release. More, client APIs can be provided in the future release as and when the other message passing concepts like collectives, communicators, process topologies etc. are implemented.

#### Correlation Id

MPLJava generates a unique Correlation-Id for every message. The correlation Id contains Source Rank (20 bit), Destination Rank (20 bit) and Message Counter (24 bit). The same Correlation-Id will be generated by both the sender and receiver. This Correlation-Id is used to match the messages on either side. The number of bits can be increased easily to accommodate future functionalities such as Communicator-Id.

#### Request

Request is the container of the message to be sent and received. It also contains the metadata of the message such as Correlation-Id. This is similar to MPI’s Request. It internally provides a ByteBuffer wrapper for the raw byte message. The Request object is also used as a semaphore to notify completion of the operation. The Request provides a public API, iwait, to wait for non-blocking communication to complete.
3.2.3 Queuing Mechanism

Queuing Mechanism handles buffering of messages. For Eager Protocol, send message received from remote node are stored in the foreign-send queue until a matching receive message is issued by the receiver. Similarly a receive message issued by the receiver is stored in the receive queue until a matching data messages is received in the foreign send queue. For Rendezvous Protocol, the sending process stores the send message in a queue until a ReadyToSend message is received. The following queues are defined in MPLJava’s queuing mechanism.

**RecvQueue**

RecvQueue maintains a map of RecvRequest indexed with their correlation-Id. If the RecvRequest uses eager protocol, the MatchMaker picks it. Otherwise if the RecvRequest uses rendezvous protocol, the Connector picks it and adds a ReadyToSend message to the SendReadyToSendQueue.

**SendQueue**

SendQueue maintains a FIFO (First In First Out) queue of SendRequest indexed by the node Address of the destination rank. There is only one SendQueue per remote node. All SendRequest destined for the local node are moved directly to the local ForeignSendQueue. Else if the SendRequest uses rendezvous protocol, it is moved to the WaitingSendQueue until a ReadyToSend message is received.

**ForeignSendQueue**

ForeignSendQueue buffers the SendRequest received from remote node for eager protocol.

**SendReadyToSendQueue**

SendReadyToSendQueue is used by rendezvous protocol on the receiver side to buffer ReadyToSend message to be sent to remote node. There is only one SendReadyToSendQueue per remote node.

**WaitingSendQueue**

WaitingSendQueue is used by rendezvous protocol on sender side to buffer SendRequest’s waiting to receive a ReadyToSend message.

**RecvReadyToSendQueue**

RecvReadyToSendQueue is used by rendezvous protocol on sender side to buffer the ReadyToSend message received from remote node.
MatchMaker

The MatchMaker is used by eager protocol. It matches the SendRequest in ForeignSendQueue with the corresponding RecvRequest in the RecvQueue based on the correlation-Id. It copies the bytes from SendRequest to RecvRequest and notifies completion of both the request.

```java
public static synchronized void addSendRequest(Request sendRequest) {
    long correlationId = sendRequest.getCorrelationId();
    if(recvQueue.containsKey(correlationId)) {
        Request recvRequest = recvQueue.remove(correlationId);
        byte[] recvData = recvRequest.getData();
        byte[] sendData = sendRequest.getData();
        System.arraycopy(sendData, 0, recvData, 0, recvData.length);
        sendRequest.notifyCompletion();
        recvRequest.notifyCompletion();
    } else {
        foreignSendQueue.put(correlationId, sendRequest);
    }
}

public static synchronized void addRecvRequest(Request recvRequest) {
    long correlationId = recvRequest.getCorrelationId();
    if(foreignSendQueue.containsKey(correlationId)) {
        Request sendRequest = foreignSendQueue.remove(correlationId);
        byte[] sendData = sendRequest.getData();
        byte[] recvData = recvRequest.getData();
        System.arraycopy(sendData, 0, recvData, 0, recvData.length);
        sendRequest.notifyCompletion();
        recvRequest.notifyCompletion();
    } else {
        recvQueue.put(correlationId, recvRequest);
    }
}
```

Figure 2 – code snippet of MatchMaker
### 3.2.4 Abstract Device Interface

Abstract Device Interface (ADI), also called *Connector*, is a simple generic interface to the communication platform. Based on the system where the library is deployed, the user can specify the matching Intra-Node Connector, Inter-Node Connector and Collective Connector properties in the Config.properties. The initial design of ADI provides an Intra-Node Connector based on shared memory, and an Inter-Node Connector based on *java.nio* library for communication via IP.

The Connector Interface provides API for initialization and completion. Any parameter required by the connector can be defined in Config.properties, and read from the property file during initialization.

```java
public interface Connector {
    void init(Properties configProperties) throws Exception;
    void finish() throws Exception;
    void send(byte[] buf, int destinationRank, Comm comm) throws Exception;
    Request isend(byte[] buf, int destinationRank, Comm comm) throws Exception;
    void recv(byte[] buf, int destinationRank, Comm comm) throws Exception;
    Request irecv(byte[] buf, int destinationRank, Comm comm) throws Exception;
    void bcast(byte[] buf, int root, Comm comm) throws Exception;
    void Barrier(Comm comm) throws Exception;
}
```

**Figure 3 - Connector Interface**

**Intra-Node Connector**

Intra-Node Connector in MPLJava is implemented by shared memory communication. Intra-Node Connector is based on eager protocol. Here instead of sending the message across the interconnect, the message is communicated between the tasks via the shared address space.

The sending task places the SendRequest in the ForeignSendQueue of the node where it resides. The receiving task places the RecvRequest in the RecvQueue of the node where it resides. The MatchMaker matches the SendRequest and RecvRequest based on the correlation-Id and copies the bytes from SendRequest to RecvRequest and notifies completion of both the requests.

**Inter-Node Connector**

Inter-Node Connector in MPLJava is implemented using *java.nio* library for communication via IP. Advantages of using *java.nio* library were already discussed in Section 2.2.2. The *SocketChannel* abstracts the underlying *Socket* to allow non-blocking communication. The *ByteBuffer*, which is the underlying datatype for all NIO operation, can be allocated as direct buffers for fast I/O communication. There are two models of implementing NIO Connector,

1. Exclusive Connector Model
2. Shared Connector Model
Ideally in SPMD model we schedule one task per processor. Creating more than one task per processor might result in over subscribing the processor resulting in context switching which might have a bad impact on the performance.

**Exclusive Connector Model**

In Exclusive Connector Model each task handles its own I/O operation. This model can be implemented in two ways. In the first approach, each task handles its I/O operations in its own thread using either `java.net.Socket` or `java.nio.SocketChannel` in a blocking fashion. This approach allows us to allocate as many numbers of tasks as the number of processors without over subscribing. In the second approach, each task handles its I/O operation in a separate thread using `java.nio.SocketChannel` in a non-blocking fashion. The thread running the task can delegate the I/O operations to its I/O threads and continue with the computation. Since each task requires an exclusive thread for communication, we can allocate only half the number of processors for the actual task, without over subscribing the processor. In case of SMP clusters the network adapter is likely to be shared by all the processors in a single node, hence at any instance only some tasks can perform the I/O.

**Shared Connector Model**

In Shared Connector Model one thread handles all the I/O operation of the node. This can be implemented in NIO using a pair of `SocketChannel` between any two nodes. One `SocketChannel` for read and one `SocketChannel` for write. Threads running the task can delegate the I/O operations to the shared I/O thread and continue with the computation. In this model one processor can handle the I/O thread while the remaining processors can be allocated for the computation task without over subscribing the processor. This model is better than exclusive connector model because it allows us to perform I/O in a non-blocking mode at the same time use only one thread per node to perform all the I/O operations. This model is efficient even in case of SMP clusters, where the network adaptor is likely to be shared, because threads are not going to compete with each other for accessing the network adaptor. However if the network adaptor is capable of handling concurrent channels there should be a mechanism in the shared connector model to utilize this benefit. Hence when a thread running the computation task complete its computation and wait for its I/O operation to complete, it can join the main I/O thread to perform I/O operations concurrently.

MPLJava implements the Shared Connector Model. But for this dissertation we limit ourselves to I/O operation done by a single thread per node. The idea of doing concurrent I/O operation is proposed in future work section 5.1.3.

**Implementation of NIO Connector**

Implementation of `java.nio` library was not discussed in the background chapter because implementation using `java.nio` library is specific to the problem we are trying to solve. It is driven by various parameters such as the number of threads to use, type of datatype to use etc. Hence it is important to understand how we used `java.nio` library in MPLJava library to implement the inter-node connector.
The NIO connector implementation starts with creation of a Selector. A Selector can be created by calling `SelectorProvider.provider().openSelector()`. A selectable channel such as `SocketChannel` registers with the Selector. Each registered selectable channel is represented by a `SelectionKey`. A Selector maintains three sets of selection keys:

1. The **key set** containing current channel registrations of this selector.
2. The **selected-key set** containing the keys of channels ready for I/O operation.
3. The **cancelled-key set** containing the cancelled but not unregistered keys.

A key is added to a selector’s key set by registering a channel via `Channel.register()`. A key is added to the selected-key set by selection process. Selection process is performed by the `Selector.select()` method. The selection process is a blocking operation, which queries the underlying operating system for an update as to the readiness of each channel. The blocked selector can be manually returned by calling `Selector.wakeup()`.

After we create a Selector, the next step is to create a `ServerSocketChannel`. The `ServerSocketChannel` binds to a port on which to accept connections. The `ServerSocketChannel` must then be registered with the Selector.

By registering a channel with a selector we also specify the operation on which this channel is interested. There are four types of operations,

1. **OP_ACCEPT** – The `ServerSocketChannel` should register with this operation. If the selector detects that the corresponding server-socket channel is ready to accept a new connection, it will add the corresponding key to the selected-keys set.
2. **OP_READ** – An accepted `SocketChannel` should register with this operation. If the selector detects that the corresponding channel is ready for reading, it will add the corresponding key to the selected-keys set.
3. **OP_WRITE** – A client `SocketChannel` should register with this operation whenever it is ready to write. The selector adds the corresponding key to selected-keys set when the channel is ready to write.
4. **OP_CONNECT** – A client SocketChannel should register with this operation if it wants to connect to a remote node in a non-blocking mode.

```java
initServerConnection(InetAddress address, int port) {
    Selector selector = SelectorProvider.provider().openSelector();
    ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
    serverSocketChannel.configureBlocking(false);
    InetSocketAddress isa = new InetSocketAddress(address, port);
    serverSocketChannel.socket().bind(isa);
    serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
}
```

**Figure 4 – creation of Selector and ServerSocketChannel**

At this point we have a `ServerSocketChannel` ready and waiting and we’ve indicated that we’d like to know when a new connection is available to be accepted. Now we
need to actually accept it. This brings us to our ‘select loop’. Select loop is where most of the action begins. Our selecting thread sits in a loop waiting until one of the channels registered with the selector is in a state that matches the operation we’ve registered for it. The pseudo code for select loop is shown in Figure-5.

```java
while(live) {
  this.selector.select();
  Iterator<SelectionKey> selectedKeys = this.selector.selectedKeys().iterator();
  while (selectedKeys.hasNext()) {
    SelectionKey key = selectedKeys.next();
    selectedKeys.remove();
    if (!key.isValid()) {
      continue;
    }
    if (key.isAcceptable()) {
      this.accept(key);
    } else if (key.isReadable()) {
      this.read(key);
    } else if (key.isWritable()) {
      this.write(key);
    }
  }
}
```

**Figure 5 - Select Loop**

The accept routine accepts the incoming connection and creates a read `SocketChannel`. Once we have accepted a connection it’s only of any use if we can read data on it. Hence the newly accepted `SocketChannel` is registered with selector for OP_READ operation.

```java
accept(SelectionKey key) {
  SocketChannel readSocketChannel = serverSocketChannel.accept();
  readSocketChannel.configureBlocking(false);
  readSocketChannel.socket().setTcpNoDelay(true);
  readSocketChannel.register(this.selector, SelectionKey.OP_READ);
}
```

**Figure 6 - create and register Read-SocketChannel**

In MPLJava we will not be using OP_CONNECT operation because we will be connecting all nodes at start-up in a blocking fashion. This is because non-blocking connection might throw a `ConnectException` if the remote node has still not started. The connection to remote node can be created as shown in Figure-7.

```java
initClientConnection(InetAddress address, int port) {
  SocketChannel writeSocketChannel = SocketChannel.open();
  writeSocketChannel.configureBlocking(true);
  writeSocketChannel.connect(new InetSocketAddress(address, port));
  writeSocketChannel.configureBlocking(false);
  writeSocketChannel.socket().setTcpNoDelay(true);
}
```

**Figure 7 - create Write-SocketChannel**

Note: We configured the `ServerSocketChannel`, read `SocketChannel` and write `SocketChannel` to non-blocking mode. This is because in shared connector model only
one thread handles all I/O operation. Hence having a blocking mode might cause indefinite blocking leading to deadlock and starvation.

We are yet to register the write `SocketChannel` with the `Selector`. We should register write `SocketChannel` for OP_WRITE operation only when we have data ready to write. If we register a channel for OP_WRITE and leave it set for ever, it leaves the selecting thread spinning because 99% of the time a socket channel is ready for writing unless the socket buffer is full. Hence we register for OP_WRITE only when there are messages in the `SendQueue` or `RendzSendReadyToSendQueue`. This should be done just before the select call in the select loop.

```java
if(sendQueue contains message || rendzSendReadyToSendQueue contains message) {
    writeSocketChannel.register(this.selector, SelectionKey.OP_WRITE);
}
```

Figure 8 - register Write-SocketChannel

Write operation first writes the ReadyToSend messages and then the SendRequest messages. Before a SendRequest message, a header containing the correlation-Id and message size is added. The write operation can be unsuccessful if the socket buffer is full. In this case the pending SendRequest message is temporarily held in the PendingSendQueue. Hence write operation should check for any message in the PendingSendQueue before proceeding to send the ReadyToSend messages otherwise data will be sent out of sequence. Figure-9 shows the pseudo code for write routine.

```java
write(SelectionKey key) {
    SocketChannel writeSocketChannel = key.channel();
    Address remoteAddress = get address corresponding to writeSocketChannel
    if(pendingSendQueue contains message for remoteAddress) {
        Request sendRequest = get pending message from pendingSendQueue
        writeSocketChannel.write(sendRequest);
        if (completed successfully) {
            notify completion of sendRequest
        } else {
            return;
        }
    }

    while(rendzSendReadyToSendQueue contains message for remoteAddress) {
        READY_TO_SEND message = get message from rendzSendReadyToSendQueue
        writeSocketChannel.write(READY_TO_SEND message);
        if (not successfully) {
            return;
        }
    }

    while (sendQueue contains message for remoteAddress) {
        Request sendRequest = get message from sendQueue
        Create HEADER message;
        writeSocketChannel.write(HEADER);
        if(completed successfully) {
            writeSocketChannel.write(sendRequest);
            if (completed successfully) {
                notify completion of sendRequest
            } else {
                add sendRequest to pendingSendQueue
                return;
            }
        }
    }

    key.interestOps(SelectionKey.OP_READ);
}
```

Figure 9 - pseudo code of write routine
If there are no more messages to write in the write SocketChannel, the operation is changed to OP_READ (even though we are not going to read through the write SocketChannel) in order to stop the select loop spinning.

Read operation reads two types of messages: ReadyToSend message and SendRequest from remote node. If at any point the read operation is incomplete the partially read SendRequest is temporarily stored in the PendingRecvQueue. Hence read operation always checks if any message is left in the PendingRecvQueue before proceeding to read a new message. The pseudo code for read operation is shown in Figure-10.

```
read(SelectionKey key) {
    SocketChannel readSocketChannel = key.channel();
    Address remoteAddress = get address corresponding to readSocketChannel
    if(pendingRecvQueue contains message for remoteAddress) {
        Request recvRequest = get message from pendingRecvQueue
        readSocketChannel.read(recvRequest);
        if(completed successfully) |
            if(rendezvous) notify completion of recvRequest
        else
            add recvRequest to foreignSendQueue in MatchMaker
    } else {
        return;
    }

    while(readSocketChannel contains more message) {
        if(READY_TO_SEND message) {
            correlationId = message.getCorrelationId()
            if(WaitingSendQueue contains correlationId)
                move sendRequest from WaitingSendQueue to SendQueue
            else
                put message in RendzRecvReadyToSendQueue
        } else if (remote SendRequest message) {
            if(rendezvous)
                get recvRequest from RecvQueue
            else
                create temporary recvRequest
                readSocketChannel.read(recvRequest);
            if(completed successfully) {
                if(rendezvous)
                    notify completion of recvRequest
                else
                    add recvRequest to foreignSendQueue in MatchMaker
            } else {
                add recvRequest to pendingRecvQueue
                return;
            }
        }
    }
}
```

Figure 10 - pseudo code of read routine

The send routine in NIO connector wraps the data along with its correlation-Id in a SendRequest. For eager protocol, this request is placed directly in the SendQueue. For rendezvous protocol, if a ReadyToSend message is already received, the SendRequest is placed in the SendQueue otherwise it is placed in the WaitingSendQueue.

The recv routine in NIO connector wraps the data along with its correlation-Id in a RecvRequest. For eager protocol, this request is placed directly in the RecvQueue. For rendezvous protocol, the request is placed in the RecvQueue and a ReadyToSend message is generated and placed in the RendzSendReadyToSendQueue.
3.2.5 Datatypes

Providing Java datatypes for High Performance Computing by itself is a dissertation topic. Traditionally Java Serialization is used for converting Java objects to and from byte stream. Serialization is the process of saving an object’s state to a sequence of bytes; De-serialization is the process of rebuilding the bytes into a live object. Serialization writes the data associated with the object along with the metadata of the class associated with the object. Then it recursively writes all the data and class information of its super class until it finds `java.lang.Object`. Hence the process of serialization and de-serialization is a costly operation for high performance computing because of high CPU utilization and unnecessary increase in the message size.

In Message Passing Model for High Performance Scientific Computing, our primary motivation is to transfer just the data. Hence the data has to be represented in the simplest form as possible. Java doesn’t provide any direct mechanism to convert primitive datatypes to bytes and bytes to primitive datatypes. However, from JDK 1.4 as part of `java.nio` library, Java provided a class called `java.nio.Buffer` and its subclasses namely `ByteBuffer`, `IntBuffer`, `FloatBuffer` and `DoubleBuffer`. These classes are containers for primitive datatypes and can be transformed to and from byte sequence by simple coding.

MPLJava provides a set of primitive datatypes for message passing such as `ByteArray`, `IntegerArray`, `FloatArray` and `DoubleArray` to show how `ByteBuffer` can be used to create a datatype for message passing model. The programmer writing the task can extend this idea to create his own datatypes.

The code in Figure-11 shows an implementation of `IntegerArray` where we are creating a byte array and providing an integer wrapper around the byte array. Using this approach, gives us the flexibility to convert primitive datatypes to byte sequence and byte sequence to primitive datatypes. This is similar to allocating memory in C language using `malloc()`. The `getLocation()` routine in `IntegerArray` can retrieve the starting location of any sub sequence of the `IntegerArray` irrespective of the number of dimension. The `getLocation()` routine is similar to pointer in C language. However, to get non-sequential bytes we have to copy the bytes to a separate array.

The code snippet in Figure-12 shows how to use the `IntegerArray`. In this code snippet we copy a sub sequence of ‘array’ starting from [1][0][0] into target[12]. Hence the process of converting from integer array of any dimension to integer array of any other dimension is much simpler. Moreover it is backed by the `ByteBuffer` which can used directly to send the data across network using `java.nio`.

However, these datatype cannot be compared to derived datatypes of MPI because, MPI derived datatypes have mechanism to send non-sequential bytes. It might be possible in Java to provide such functionality using a mechanism similar to `FileChannels` in `java.nio` library but as said earlier this is a research topic by itself hence it has to be studies properly before coming to any conclusion.
public class IntegerArray {
    ByteBuffer byteWrapper;
    IntBuffer intWrapper;
    int[] arraySize;

    public IntegerArray(int dimension, int... sizes) {
        arraySize = new int[dimension];
        int totalSize = 1, direction = 0;
        for (int i : sizes) {
            arraySize[direction++] = i;
            totalSize *= i;
        }
        byteWrapper = ByteBuffer.allocate(totalSize * 4);
        intWrapper = byteWrapper.asIntBuffer();
    }

    public int get(int... index) {
        if (index.length != arraySize.length)
            throw new MPLJavaException();
        return intWrapper.get(getLocation(index));
    }

    public void put(int value, int... index) {
        if (index.length != arraySize.length)
            throw new MPLJavaException();
        intWrapper.put(getLocation(index), value);
    }

    public int getLocation(int... index) {
        int offset = byteWrapper.limit() / 4;
        int location = 0;
        for (int i = 0; i < index.length; i++) {
            offset /= arraySize[i];
            location += offset * index[i];
        }
        return location;
    }

    public ByteBuffer getByteBuffer() {
        return byteWrapper;
    }
}

Figure 11 - IntegerArray datatype

int value = 0;
IntegerArray array = new IntegerArray(3,2,3,4);//same as array[2][3][4]
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
        for (int k = 0; k < 4; k++) {
            array.put(value++, i, j, k);//same as array[i][j][k]=value++
        }
    }
}
ByteBuffer src = array.getByteBuffer();
ByteBuffer target = ByteBuffer.allocate(12*4);
System.arraycopy(src.array(), array.getLocation(1,0,0)*4, target.array(), 0, 12*4);
IntBuffer subArray = target.asIntBuffer();
System.out.println(subArray.get(0)); //prints 12. Same as subArray[0]

Figure 12 - using IntegerArray
3.3 Workflow Diagram

The workflow diagram in Figure-13 explains the complete work flow of point to point communication. The communication starts when SendRequest and RecvRequest are posted and ends when they are notified for completion.

When a SendRequest is posted, if the destination task of the SendRequest is present in the same node then the SendRequest is added directly to the local ForeignSendQueue. Else the message size of the SendRequest is checked to identify the protocol to use. If the message size is less than MPLJAVA_EAGER_SIZE then eager protocol is used else rendezvous protocol is used. The default maximum message size for eager protocol is 64KB. For eager protocol, the SendRequest is added to the SendQueue. The Inter-Node Connector picks the SendRequest from SendQueue and sends it across interconnect to the Inter-Node Connector in the remote node. For rendezvous protocol, the SendRequest is placed in the SendQueue if a ReadyToSend message is already received in the RendzRecvReadyToSendQueue. Otherwise the SendRequest is placed in the WaitingSendQueue until a matching ReadyToSend message is received.

When a RecvRequest is posted, it is placed in the RecvQueue. The size of the RecvRequest is checked to identify the protocol to use. For rendezvous protocol, a ReadyToSend message is created and placed in the SendReadyToSendQueue. The Inter-Node Connector picks the ReadyToSend message from SendReadyToSendQueue and sends it across interconnect to the Inter-Node Connector in the remote node.

The Inter-Node Connector receives two types of messages from a remote node; ReadyToSend message and SendRequest message. When a ReadyToSend message is received, it checks if a matching SendRequest is present in the WaitingSendQueue. If a matching request is found, it moves the SendRequest from WaitingSendQueue to SendQueue. Otherwise, the ReadyToSend message is placed in the RecvReadyToSendQueue. When a SendRequest message is received, it checks the data size to find the type of protocol. If it is rendezvous protocol, it copies the SendRequest message directly to the RecvRequest in the RecvQueue and notifies completion of the RecvRequest. Else if it is eager protocol, it copies the SendRequest message to a temporary request and places the request in the ForeignSendQueue.

Whenever a SendRequest is added to the ForeignSendQueue, the MatchMaker finds for a matching RecvRequest in the RecvQueue. Similarly whenever a RecvRequest is added to the RecvQueue for eager protocol, it finds for a matching SendRequest in the ForeignSendQueue. It then copies the data from SendRequest to RecvRequest and notifies completion of both the requests.

Figure-13 shows the pictorial representation of the workflow.
Figure 13 - Workflow Diagram
Chapter 4

Testing and Performance

The testing and performance comparisons were done in HPCx server in order to test the library for both shared memory communication and IP based communication.

4.1 HPCx

HPCx is a cluster of 160 IBM eServer 575 servers which constitutes one of the national HPC service in the UK. Each frame (also called node) contains 16 Power5 processors, and has 32GB memory. Along with the 160 eServer 575 frames, there are 8 additional eServer 575 servers used for login and disk I/O.

Hardware

Each Power5 processor runs at 1.5 GHz with a theoretical peak performance of 6Gflop/s. Each chip has its own L1 cache consisting of 32KB data cache and 64KB instruction cache. The L2 cache of 2MB is shared between two processors on a single chip. The L3 cache of 36MB is shared between the two processors on a single chip. 16 Power5 processors make up one eServer frame. Each frame has 32GB of main memory accessible by all processors. The interconnect between frames is provided by IBM’s High Performance Switch. Each eServer frame has two network adapters and there are two links per adapter, making a total of four links between each of the frames and the switch network.

Software

Each eServer 575 frame in HPCx runs IBM’s AIX operating system. Standard MPI library, Java 1.5 and mpiJava library are already installed in the system.

Allocation Units

The total number of allocation units used for testing and performance comparison of MPLJava with standard MPI and mpiJava, is 3000 AU.
4.2 Correctness Test

Correctness tests for point to point communication were carried out using a basic Ring Program. In a ring program, tasks are considered to be in the form of a ring based on the rank. Each task has a previous task and next task. The previous of root will be the last and the next of last will be the root. The rank of the task is circulated from each task to the next task until all tasks receive their original rank. For testing blocking communication, all odd rank’s first send then receive and all even rank’s first receive and then send. For testing non-blocking communication, all ranks first send and then receive.

![Figure 14 – Correctness Test – Tasks in a Ring](image)

Neighbouring tasks across the node communicate via NIO connector, while neighbouring tasks within the node communicate via shared memory connector. Hence this program integrates the testing for both shared memory communication and NIO communication.

```
sendmsg = myrank;
total = sendmsg;

while(true) {
    Request srequest = MPI.isend(sendmsg, nextRank, comm);
    Request rrequest = MPI.irecv(recvmsg, prevRank, comm);
    srequest.iwait();
    rrequest.iwait();
    if(recvmsg == myrank) break;
    total = total + recvmsg;
    sendmsg = recvmsg;
}
System.out.println("Rank = " + myrank + " Total = " + total);
```

![Figure 15 - pseudo code of ring program to compute global-sum of ranks](image)
4.3 Performance Test

The performance of a message passing library is generally measured in terms of latency and bandwidth of point to point communication. A PingPong program was written, which sends increasing sized messages back and forth between tasks. To ensure that anomalies in message timings are minimised, the PingPong is repeated many times for each message size. The following code snippet shows the main loop of the PingPong program.

```java
half = size/2;
if(rank < half) {
    neig = rank+half;
    for(int i = 0 ; i < maxiter ; i++) {
        MPI.send(sarray.getBytes(),neig,comm);
        MPI.recv(rarray.getBytes(),neig,comm);
    }
} else if(rank >= half) {
    neig = rank-half;
    for(int i = 0 ; i < maxiter ; i++) {
        MPI.recv(rarray.getBytes(),neig,comm);
        MPI.send(sarray.getBytes(),neig,comm);
    }
}
```

Figure 16 - pseudo code of pingpong

4.3.1 Latency Test

The latency tests were carried out using the ping-pong program. The sender sends a message with a data size of zero or one byte to the receiver and waits for a reply from the receiver. The receiver receives the message from the sender and sends back a reply with the same data size. Many iterations of the ping-pong test were carried out and average one-way latency numbers were obtained. Blocking version of send and recv operations were used in the tests.

4.3.2 Multi-pair Latency Test

This test is very similar to the latency test. However, at the same instance multiple pairs are performing the same test simultaneously. The processes are divided into two equal blocks according to their ranks. Each process from one block forms a pair with a process from the other block. For example, process with rank '0' pairs with the process with rank 'np/2' and rank '1' with 'np/2 + 1' and so on. For intra-node testing all blocks reside in the same node, whereas for inter-node testing each block resides in a separate node. The code snippet for this test is shown in Figure-16.

The following table shows the latency figures for MPLJava, IBM MPI and mpiJava in HPCx.
Table 1 - Latency

<table>
<thead>
<tr>
<th>Latency</th>
<th>Intra-Node</th>
<th>Inter-Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPLJava</td>
<td>MPI</td>
<td>mpiJava</td>
</tr>
<tr>
<td>Single-pair</td>
<td>22 µs</td>
<td>3 µs</td>
</tr>
<tr>
<td>Multi-pair</td>
<td>59 µs</td>
<td>15 µs</td>
</tr>
</tbody>
</table>

It is clear that the latency of the IBM MPI implementation is the lowest of all. mpiJava follows next because it essentially is using the same messaging mechanism. The reason for such low latency is because the MPI library in HPCx uses a special low latency communication medium called user-space or US. To have a fair comparison we tried to force MPI and mpiJava to use IP communication by specifying `#@ network.MPI = csss,shared,IP` instead of `#@ network.MPI = csss,shared,US` in the job submission script. As a result the latency of MPI and mpiJava increased considerably as shown below.

Table 2 - Inter-Node Latency with only IP communication

<table>
<thead>
<tr>
<th>Latency</th>
<th>MPLJava</th>
<th>MPI</th>
<th>mpiJava</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-pair</td>
<td>187 µs</td>
<td>52 µs</td>
<td>89 µs</td>
</tr>
<tr>
<td>Multi-pair</td>
<td>652 µs</td>
<td>67 µs</td>
<td>117 µs</td>
</tr>
</tbody>
</table>

However, the latency of MPLJava for Inter-Node communication is still too high compared to its peers. The reason for higher latency is a combination of the use of thread-safe algorithms and a possible additional copying done by the JVM internally to copy data between the Java heap and C heap whenever a device level operation is invoked for I/O. We cannot avoid the thread-safe mechanisms but we can avoid the additional copying by using direct ByteBuffer. A direct ByteBuffer solution is not implemented in the current version of MPLJava due to time constraints. But a solution based on direct ByteBuffer is proposed in the future work section 5.1.1.

4.3.3 Bandwidth Test

The same ping-pong program is used to perform the bandwidth tests. The sender sends a fixed number of messages to the receiver and waits for the reply from the receiver. The receiver receives the message from the sender and sends back a reply with the same data size. The receiver sends the reply only after receiving the entire message. Many iterations of the ping-pong test were carried out and the bandwidth is calculated based on the elapsed time and the number of bytes sent by the sender. The elapsed time is the time taken by the sender to send the first message, to the time it receives the last reply back from the receiver. The objective of this bandwidth test is to determine the maximum sustained date rate that can be achieved at the network level by a single task. Blocking version of `send` and `recv` operation was used in the test.
4.3.4 Multi-pair Bandwidth Test

The multi-pair bandwidth is same as the bandwidth test. However, at the same instant multiple pairs are performing the same test simultaneously. The processes are divided among two blocks according to their ranks. Each process from one block forms a pair with the corresponding process from the other block. For example, process with rank '0' pairs with the process with rank 'np/2' and rank '1' with 'np/2 + 1' and so on. For intra-node test the blocks reside in the same node, whereas for inter-node test each block resides in a separate node. This process is repeated for several iterations. The objective of this bandwidth test is to determine the maximum sustained date rate that can be achieved at the network level collectively by all the tasks in a node. Blocking version of send and recv operation was used in the test.

4.3.5 MPLJava Shared Memory Connector Bandwidth

The graph in Figure-17 shows the Intra-Node Bandwidth achieved by MPLJava in HPCx.

![Figure 17 - MPLJava Shared Memory Connector Bandwidth](image)

From the above graph we see that the shared memory bandwidth for multi-pair is not scaling well as compared to single-pair mapping, inspite of allocating only one task per processor. Since we allocated only one task per processor we expect the memory copy to happen concurrently. Hence the time taken for a multipair memory copy should have been same as the time taken for a singlepair memory copy. On profiling the code we found that the MatchMaker, which is shared by all the tasks, was being locked exclusively by a single task when the memory copy happens between the SendRequest and RecvRequest. This was eliminated by replacing the intrinsic synchronization used by Java with a `java.util.concurrent.locks.ReentrantLock`. As a result the bandwidth for single-pair and multi-pair are almost equal as expected.
private static Lock lock = new ReentrantLock();

public static void addSendRequest(Request sendRequest) {
    long correlationId = sendRequest.getCorrelationId();
    lock.lock();
    if (recvQueue.containsKey(correlationId)) {
        Request recvRequest = recvQueue.remove(correlationId);
        lock.unlock();
        byte[] recvData = recvRequest.getData();
        byte[] sendData = sendRequest.getData();
        System.arraycopy(sendData, 0, recvData, 0, recvData.length);
        sendRequest.notifyCompletion();
        recvRequest.notifyCompletion();
    } else {
        foreignSendQueue.put(correlationId, sendRequest);
        lock.unlock();
    }
}

public static void addRecvRequest(Request recvRequest) {
    long correlationId = recvRequest.getCorrelationId();
    lock.lock();
    if (foreignSendQueue.containsKey(correlationId)) {
        Request sendRequest = foreignSendQueue.remove(correlationId);
        lock.unlock();
        byte[] sendData = sendRequest.getData();
        byte[] recvData = recvRequest.getData();
        System.arraycopy(sendData, 0, recvData, 0, recvData.length);
        sendRequest.notifyCompletion();
        recvRequest.notifyCompletion();
    } else {
        recvQueue.put(correlationId, recvRequest);
        lock.unlock();
    }
}

Figure 18 - MPLJava Shared Memory Connector Bandwidth after Improvement

Figure 19 - MatchMaker code snippet with ReentrantLock
4.3.6 Intra-Node Bandwidth Comparison

The graphs in figure-20 and figure-21 show the single-pair and multi-pair Intra-Node Bandwidth for MPLJava, MPI and mpiJava libraries in HPCx.

**Figure 20 – Singlepair Intra-Node Bandwidth**

**Figure 21 - Multipair Intra-Node Bandwidth**
From the above graph it is clear that intra-node bandwidth of MPLJava out-performs
the IBM MPI and mpiJava libraries. However for smaller message size, the overall
transmission time is dominated by latency, hence IBM MPI performs better than
MPLJava. As discussed in the previous chapters, the existing message passing libraries
in Java, like mpiJava, create a separate JVM for each task. The process of running
multiple tasks in a node implies running multiple JVMs in a single node. Whereas
MPLJava tasks run in threads that share a single JVM. Threads within a JVM share a
global address space. In general communication via global address space is faster than
communication via shared memory segment. Standard MPI implementation might be
using shared memory segment for communication. This is the reason why MPLJava’s
performance is better than standard MPI implementation and mpiJava. So the single
JVM approach used in MPLJava proved to be a good solution.

4.3.7 Inter-Node Bandwidth Comparison

The graphs in figure-22 and figure-23 show the single-pair Inter-Node bandwidth of
MPLJava, IBM MPI and mpiJava libraries in HPCx.

![Figure 22 - SinglePair Inter-Node Bandwidth](image)

It is clear that the single-pair bandwidth of IBM MPI is the best of all. However, the
performance of mpiJava is surprisingly low because one would expect mpiJava to
perform close to MPI because it is essentially using the same messaging mechanism.
The performance of MPLJava is three times slower than the standard MPI. MPLJava is
designed to be a generic solution, hence no local optimizations have been performed.
Whereas the standard MPI implementations are optimized to perform best with the
available hardware. For example, IBM MPI use a dedicated low latency US user-space
communications. To have a fair comparison we tried to force MPI and mpiJava to use
only IP communication by specifying `#@ network.MPI = csss,shared,IP` instead
of `network.MPI = csss.shared,US` in the job submission script. The outcome: MPLJava seem to have a better single-pair bandwidth utilization than MPI and mpiJava libraries. However for small messages MPLJava is slower because the overall transmission time is dominated by poor latency.

**Figure 23 - SinglePair Inter-Node Bandwidth via only IP**

The graphs in figure-24 and figure-25 show the multi-pair Inter-Node bandwidth (with and without user-space communication respectively) for MPLJava, IBM MPI and mpiJava in HPCx.

**Figure 24 - MultiPair Inter-Node Bandwidth**
The multi-pair bandwidth of MPI and mpiJava out-performs MPLJava in HPCx. However, the results show mpiJava is performing better than MPI beyond a certain message size which is strange because mpiJava is essentially using JNI to invoke the native MPI. This result is not a random noise but occur consistently. This might be related to some cache effects.

The next graph shows a fair comparison of results where all libraries are using the same messaging mechanism i.e. IP.

![Figure 25- MultiPair Inter-Node Bandwidth via only IP](image)

Inspite of using the same communication mechanism, MPLJava’s performance is 30% slower than MPI and mpiJava. This may be because, in HPCx each node has two network adapters and there are two links per adapter, making a total of four links between the node and the switch network. In MPI and mpiJava each process handles its own I/O operation hence MPI might be performing I/O concurrently via multiple links. Whereas in MPLJava’s shared connector model we handle all I/O operations sequentially in a single thread. This issue is already discussed under the shared connector model in the design chapter. A more efficient solution is proposed in the future work section 5.1.3, where the threads waiting for I/O operation to complete, will join the main I/O thread to perform I/O concurrently. The other possible reason for the performance difference might be because of soft tuning of TCP/IP parameters. We can try to boost the performance of socket by minimising the packet transmit latency, minimising system call overhead, adjust TCP window for the bandwidth delay product and dynamically tune the TCP/IP stack\(^{(14)}\). The standard MPI implementation might be optimized by tuning these parameters whereas MPLJava (in the current version) has only minimised the packet transmit latency by setting TCP_NO_DELAY.
Chapter 5

Future Work

5.1 Optimizations

This section proposes a few extensions to the current design which when implemented will improve the performance of MPLJava. These extensions were not implemented in the current version due to time constraints. The proposed extensions are

1. Use direct ByteBuffer for NIO connector
2. Use of object pool for buffering received eager messages in NIO connector
3. Multiplexing I/O in NIO connector

5.1.1 Direct ByteBuffer

In the previous chapter we saw that the latency of MPJava for Inter-Node communication is poor compared to standard MPI and mpiJava. Using a direct ByteBuffer is one of the mechanisms to improve the latency. In the current implementation we are passing normal ByteBuffer to the SocketChannels during read/write operation. This internally might cause copying the data from Java heap to C heap because most device drivers are written in C and hence Java library might internally use JNI to invoke the device. This layer of copying from Java heap to C heap can be avoided by using direct ByteBuffer.

A direct ByteBuffer can be created by invoking `ByteBuffer.allocateDirect()` factory method. The contents of direct buffers reside outside the normal garbage-collected heap. Hence the buffer returned by this method typically has higher allocation and de-allocation cost. Since direct buffers are not subject to garbage collection it is recommended that direct buffer should be used only for large, long-lived buffers that are subject to the underlying system’s native I/O operations.

Given the advantage of direct buffers, we can use it in NIO connector for SocketChannels read/write operation. But it is not simple to implement direct buffers because the library should internally handle copying the data between the user buffer and the direct buffer. For a non-blocking communication where streaming data can halt
at any point, this solution becomes more complex. The following pseudo code shows how to implement direct buffer in NIO connector of MPLJava.

```java
int writeBufferSize = SocketChannel.socket().getSendBufferSize();
int readBufferSize = SocketChannel.socket().getReceiveBufferSize();
ByteBuffer directReadBuffer = ByteBuffer.allocateDirect(readBufferSize);
ByteBuffer directWriteBuffer = ByteBuffer.allocateDirect(writeBufferSize);

read() {
    sc.read(directReadBuffer);
    copy data from directReadBuffer to RecvRequest
}

write() {
    copy data from SendRequest to directWriteBuffer
    sc.write(directWriteBuffer);
}
```

Figure 26 - direct buffer pseudo code

5.1.2 Object Pooling

In the current MPLJava’s Queuing Mechanism we saw that we are buffering the received SendRequest message for eager protocol in a temporary Request object and placing it in the ForeignSendQueue until the corresponding RecvRequest is posted. Once the data is copied the temporary Request object is garbage collected. At very high frequencies lot of such Request objects are created and garbage collection. This may incur a large CPU cost. In order to avoid repeated object creation and destruction we can create a pool of Request object and reuse the objects. But the solution is not simple because the incoming data message will be of varying size, so what should be the size of the pooled Request object? We can propose some interesting solutions for this problem like,

1. Applying heuristics to say that the communicating processes are more likely to send and receive messages of same size more than once. Hence instead of creating a static object pool we can maintain a dynamic pool of fixed size and try to pool and reuse the recently used Request object until we receive message of same size.

2. Create a static pool of Request objects with predetermined size. If the data message size is greater than the Request object size then use multiple Request objects.

Each of the above solutions can be experimented with MPLJava in order to reduce the latency of inter-node communication.

5.1.3 Multiplexing I/O

The shared connector model used in MPLJava uses a single thread to perform I/O operations of all the tasks. Hence I/O operations can only be performed sequentially. If the underlying network adaptor is capable of supporting multiple channels concurrently we need a mechanism to utilize this capability of the network adaptor. We propose a solution where once a thread, running a task, completes its computation and waits for
I/O operation to complete; it can potentially be used to perform I/O operation concurrently with the main I/O thread.

The multiplexed I/O functionality can be implemented using a custom version of `FixedThreadPoolExecutor` which is available from JDK 1.5 as part of Executor Framework. The size of the `FixedThreadPoolExecutor` should be equal to the number of processes in the node plus one for the main I/O thread. Whenever the select loop selects a SocketChannel for read/write operation the main I/O thread can query the `ThreadPoolExecutor` to find if there is any idle thread. If we find an idle thread we can use it to perform I/O concurrently with the main I/O thread. However, the mechanism to find an idle thread is not available in `FixedThreadPoolExecutor` but is available as part of `CachedThreadPoolExecutor`. But we cannot use a `CachedThreadPoolExecutor` for this problem because its size is unbounded. Hence we need to write a custom version of `ThreadPoolExecutor` which is a blend of both `FixedThreadPoolExecutor` and `CachedThreadPoolExecutor`.

### 5.1.4 Other Optimizations

Apart from the above three extensions we can also experiment and try various advanced concurrency options available in Java with the existing MPLJava implementation. We already saw that use of ReentrantLock in MatchMaker has tremendously increased the performance. Similarly we can try to use `ConcurrentMap` instead of `SynchronizedMap` to perform concurrent operations in a map without synchronization. We can use `ReadWriteLock` instead of intrinsic Java synchronization at places where multiple threads read while a single thread write. Some of these techniques have already been implemented in MPLJava but the library still has potential synchronization blocks which can be experimented.

### 5.2 More Client APIs

The current version of MPLJava provides client APIs only for point to point communication like `send`, `recv`, `isend` and `irecv`. The library can be extended in future to provide more APIs by implementing the bulk of other message passing concepts such as collective communications, communicators, process topologies etc.
Chapter 6

Conclusions

In this research, we implemented a new Message Passing Library for Java called MPLJava. The primary focus of this research was to implement the point to point communication for MPLJava, and compare its performance with standard MPI implementation and some of the existing java message passing libraries like mpiJava. We studied the message passing model and the potential java techniques that can be used to implement the message passing model. We also studied the existing message passing libraries in java, their pitfalls and the need for a new library.

MPLJava provides a generic extensible architecture to integrate different communication medium. The library is written in pure Java. The library provides a single-JVM common to all tasks running in a node, enabling fast intra-node communication via shared address space. The library implemented a shared memory based intra-node connector and a java.nio based inter-node connector for communication via IP. The library provides seamless integration with existing job submission tools with minimal configuration.

The performance of MPLJava was measured in terms of latency and bandwidth of point to point communication. Performance comparison was done in HPCx server to test both shared memory connector and NIO connector. Overall, MPLJava proved to be performing better than standard MPI and mpiJava for intra-node communication. This was accredited to the use of single JVM per node in MPLJava. However, the performance of MPLJava for inter-node communication was not impressive. The overall transmission time of message was dominated by poor latency. The reason for poor latency was due to usage of thread-safe mechanisms and possible additional copying done between Java heap and C heap. However, we proposed some solution, in the future work chapter, to improve the latency by using direct buffer, to eliminate the additional copying, and by using object pooling. The poor latency in turn affected the bandwidth of MPLJava for smaller message size. But for larger message size the bandwidth of MPLJava proved to be performing better than standard MPI and mpiJava for a single-pair inter-node communication. However, the bandwidth of MPLJava for multi-pair inter-node communication was lesser than standard MPI implementation. This is because of the sequential I/O operation performed by a single threaded shared connector model of MPLJava. But a solution has been proposed in the future work chapter to improve the bandwidth, by multiplexing I/O operation using the idle threads created for running the task.
All the above comparison holds good provided the libraries are using the same underlying communication mechanism like TCP/IP. However, we found that overall performance of IBM MPI in HPCx is much better than MPLJava. This is because the standard MPI libraries are fine tuned to exploit the local hardware. In HPCx, MPI uses a low latency communication mechanism called user-space. MPLJava being a generic library doesn’t use the user-space for communication.

MPLJava has implemented only point to point communication, shared memory connector and java.nio based inter-node connector. It can further be extended to provide inter-node connectors specific to the hardware architecture like in HPCx. It still has to implement a bulk of message passing concepts such as collective communications, communicators, process topologies etc. Thus, it is premature to conclude whether MPLJava is performing better or not. However, this research has emphasised the need for a generic and extensible message passing library for java and has initiated the implementation of such a library called MPLJava.

To conclude, MPLJava is a just born library; it has a long way to mature in order to compete with professional MPI implementations.
Appendix A

Compiling and Running MPLJava

Compiling

1. Checkout the library from training.epcc.ed.ac.uk:/home/mpjava/cvsroot
2. Make sure you have at least JDK 1.5 and Apache Ant 1.7.
3. Modify the Config.properties, if you wanted to change the default port for IP connection or if you wanted to change the default eager message size or if you want to change the name of any environment variables
4. Run ant build. This will create an MPLJava.jar in the parent directory.
5. Enter the MPLJava.jar in the classpath.

Creating MPI like Task

1. Create a task by implementing uk.ac.ed.epcc.mplj.common.Task interface.
2. Use client API’s found in Javadocs to create MPI like task.

```java
Comm comm = MPI.getMpiCommWorld();
int rank = MPI.getRank(comm);
int size = MPI.getSize(comm);
MPI.send(bytearray,destinationRank,comm);
MPI.recv(bytearray,sourceRank,comm);
```

3. Compile the Task as a normal Java application.
4. Specify the location of the task class in the classpath.

Running MPLJava

1. Execute the Startup.class in MPLJava
2. Specify the fully qualified name of the task as a command line argument to Startup.class. Any argument to the task can also be specified in the command line argument following the fully qualified name of the task.
Job Submission in HPCx

Figure-27 shows a sample job submission script for HPCx, and the various parameters to be specified for running MPLJava are highlighted in block.

```bash
#@ shell = /bin/ksh
#@ job_name = PingPong
#@ job_type = parallel
#@ cpus = 2
#@ node_usage = not_shared
#@ wall_clock_limit = 00:5:00
#@ account_no = zzz
#@ requirements = ( Feature == "SMT" )
#@ tasks_per_node = 1
#@ output = $(job_name).$(schedd_host).$(jobid).out
#@ error = $(job_name).$(schedd_host).$(jobid).err
#@ notification = never
#@ queue
export JAVA_HOME=/usr/java5_64
export PATH=/usr/java5_64/bin:$PATH
export MPLJAVA_TOTAL_PROCESSES=32
export MPLJAVA_EAGER_SIZE=65535
export MPLJAVA_MACHINE_NAME=HPCX

poe java -cp /hpcx/home/z004/z004/s0895271/MPLJava.jar uk.ac.ed.epcc.mplj.Startup PingPong
```

Figure 27 - sample job submission script for HPCx

1. Specify the total number of processes in MPLJAVA_TOTAL_PROCESSES environment variable.
2. Specify the eager message size in MPLJAVA_EAGER_SIZE environment variable (if it is different from the default value of 65535 bytes).
3. Specify the machine name in MPLJAVA_MACHINE_NAME environment variable (if you wanted to modify the default hostname).
4. Specify the TASKS_PER_NODE to be 1.
5. Specify the node usage to be NON_SHARED.
6. Specify the total number of nodes to be used in the CPUS field. Ideally total number of nodes for HPCx = MPLJAVA_TOTAL_PROCESSES / 16.
Appendix B

Development Process

We followed an incremental and iterative development process in this project. The work plan was divided into multiple phases. Each phase builds incrementally on top of the previous. Within each phase we followed iterative model to design, implement and iteratively improve the solution based on performance. Fig shows the initial work plan created during the project preparation stage. It shows the various phases, their milestones, timeline and target date of completion.

![Initial Work Plan](image)

**Figure 28 - initial work plan - phase, milestones, timeline and completing date**

Based on the initial work plan, initial risk assessment was done and various risks were identified and documented as shown in table-3.
Table 3 - Initial Risk Assessment

<table>
<thead>
<tr>
<th>Risk</th>
<th>Level (L/M/H)</th>
<th>Likelihood</th>
<th>Mitigation Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semester Examination</td>
<td>H</td>
<td>Certainty</td>
<td>Phase-1 and Phase-2 timelines are comprehensively allocated considering the semester examinations.</td>
</tr>
<tr>
<td>HPC server unavailable</td>
<td>M</td>
<td>Unlikely</td>
<td>A simulation environment will be created in my personal computer to avoid any delay due to testing in HPC server</td>
</tr>
<tr>
<td>source code control</td>
<td>H</td>
<td>Certainty</td>
<td>Sometimes the source code might be deleted by mistake or it becomes unavailable due to system failure. A CVS repository will be created in <code>training.epcc.ed.ac.uk</code> and a local copy will be checked-out to my personal computer.</td>
</tr>
<tr>
<td>Report and Presentation</td>
<td>H</td>
<td>Somewhat likely</td>
<td>Timeline for each phase is comprehensively allocated so as to do the report simultaneously</td>
</tr>
<tr>
<td>Timeline deviation due to above risks or due to unrealistic estimates</td>
<td>H</td>
<td>Somewhat likely</td>
<td>Timeline reviewed at the end of each phase to prevent undetected deviation. But if timeline is found to be deviating beyond two weeks Phase-5 will be dropped and carried out as a future development.</td>
</tr>
</tbody>
</table>

Once the project started we made some changes to the initial design and restructured some of the milestones. First, we decided to replace the bootstrap program with a task manager. According to the initial design, the bootstrap program was determined to be started in each and every node and the tasks were supposed to be submitted to the bootstrap program. This solution was trying to propose a new mechanism for job submission instead of trying to leverage the existing job submission tools. Hence in order to develop a generic solution we redesigned the library to view the library along with its task as a single entity. The modified design is described in the design chapter.

The other change to the development process was writing the dissertation report. The initial work plan comprehensively allocated time to write the report at each phase. Beyond a certain stage this practically became impossible because the iterative improvements repeatedly modified the report. Hence we planned to document all the changes and write the dissertation report at the end after all the implementation is complete.

The above changes affected the timeline of the initial phases. But effectively the first three phases were completed one week in advance of the initial target deadline. Phase-4 took considerable more time than expected because of the complexity in testing NIO connector. We tried to do the initial testing in a local environment and we succeeded in setting up a local test environment in the development machine to simulate multi-node scenario. However, the local test environment was not able to capture most of the concurrency issues associated with the inter-node communication because it was executing on a single processor machine. Hence HPCx was used for further testing of NIO connector. The process of building the library, submitting the job to HPCx and obtaining the results took considerably more time. This utilized the buffer time gained from previous phases. Since we decided to write the report at the end after completing all
the implementations, there was not enough time to implement and test collective communications, hence Phase-5 was dropped from this dissertation and added to the future work.

Regular weekly project meetings were organized with the supervisor except a few weeks. The meetings discussed the technical details of the project such as design and performance, the status of the project and changes to the work plan.
References


