

Republic of Iraq
Ministry of Higher Education and Scientific Research
Al-Nahrain University
College of Science



Implementation of a Parallel Computing Environment Using Message Passing Interface

A Thesis

*Submitted to the College of Science, Al-Nahrain University
In Partial Fulfillment of the Requirements for
The Degree of Master of Science in Computer Science*

By

Dunia Hamid Hameed
(B.Sc. 2004)

Supervisors

Dr. Lamia H. Khalid

Dr. Sawsan K. Thamer

December 2007

Dhulhejja 1428

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

(مَنْ رَزَقَهُ اللَّهُ رِزْقًا فَهُوَ حَسْبُهُ)

مَنْ رَزَقَهُ اللَّهُ رِزْقًا فَهُوَ حَسْبُهُ

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

Supervisor Certification

We certify that this thesis was prepared under our supervision at the Department of Computer Science/College of Science/Al-Nahrain University, by **Dunia Hamid Hameed** as partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Supervisors

Signature:

Name: **Lamia H. Khalid**

Title: **Assist. Prof.**

Date: 16 / 12 / 2007

Signature:

Name: **Sawsan K. Thamer**

Title: **Lecturer**

Date: 16 / 12 / **2007**

The Head of the Department Certification

In view of the available recommendations, I forward this thesis for debate by the examination committee.

Signature:

Name: **Dr. Taha S. Bashaga**

Title: **Head of the department of Computer Science, Al-Nahrain University.**

Date: 17 / 12 / **2007**

Examining Committee Certification

We certify that we have read this thesis and as an examining committee, examined the student in its content and what is related to it, and that in our opinion it meets the standard of a thesis for the degree of Master of Science in Computer Science.

Examining Committee Certification

Signature:

Name: **Dr. Abdul Monem S. Rahma**

Title: **Assistance Professor (Chairman)**

Date: 20 / 4 / 2008

Signature:

Name: **Dr. Amir S. Al-Malah**

Title: **Assistant Professor (Member)**

Date: 20 / 4 / 2008

Signature:

Name: **Dr. Jamal M. Kadhum**

Title: **Lecturer (Member)**

Date: 20 / 4 / 2008

Supervisors Certification

Signature:

Name: **Dr. Lamia H. Khalid**

Title: **Assistance Professor**

Date: 16 / 4 / 2008

Signature:

Name: **Sawsan K. Thamer**

Title: **Lecturer**

Date: 16 / 4 / 2008

The Dean of the College Certification

Approved by the Council of the College of Science

Signature:

Name: **Dr. LAITH ABDUL AZIZ AL - ANI**

Title: **The Dean of College of Science, Al-Nahrain University.**

Date: / / 2008

Dedication

To the two candles in my life, my father for his guidance and support throughout my life, to extend my gratitude from all my heart to my mother, I am eternally grateful for their continuous love, patience, and understanding, especially during this project.

To my uncle *Eng. Saad* who helped me as my father and whatever I do for him I will see it little in his right because he deserves more.

My very special thanks to my family (my sisters and my brothers) especially, my brother *Dr. Ghaith* for their continuous supports and encouragement during the period of my study.

To every one who helps me and supports me I could not do this without you, so thank you.

Dunia

Acknowledgement

First of all, my great thanks to God who helped me and gave me the ability to perform this work.

I am very grateful to *Dr. Lamia K. Khalid* for guiding me with a lot of helpful and advice. I would like to express my gratitude to *Dr. Saman Kamal* because she gave me many useful suggestions. They built the whole infrastructure of my work. And they supplied much functionality to the infrastructure in response to my requests.

I also wish to give my thanks to all the members of Computer Science Department in Al-Nahrain University for their help and encouragement, especially the Head of Department of Computer Science *Dr. Taha S. Bashaga*.

Special thanks to *Haider Mageed* for his help during the period of my study. Furthermore, this work would not have been achieved without the support and friendship of my classmate *Wurood Saad*.

I would like to thank my family for their love and patience during my project time.

Dunia

Abstract

Message Passing Interface (MPI) provides an infrastructure that enables users to build a high performance distributed computing environment from networked computers with minimum effort. It provides a common Application Programming Interface (API) for the development of parallel applications regardless of the type of multiprocessor system used.

This research implements a distributed computing system called Java Message Passing Interface Middleware which supports a Message Passing Interface Application Programming Interface (MPI API). It installs Java Message Passing Interface (JMPI) package and runs three applications (Range Addition, Matrix - Vector Multiplication and Gauss Elimination method) in two modes serial and parallel.

The system implemented on a Local Area Network (LAN) consisted of five computers. Many experiments have been performed to test the system and it found that results of parallel applications were close to the results of serial applications because the calculation times of applications were simple compared to communication times.

Table of Contents

Abstract	I
List of Abbreviations	V
List of Figures	VII
Chapter One: Introduction	
1.1 Parallel Computing	1
1.2 Message Passing Model	3
1.3 Message Passing Libraries	5
1.4 Message passing Interface (MPI) Overview	6
1.5 Literature Survey	8
1.6 Thesis's Objective	12
1.7 Thesis's Outlines	13
Chapter Two: Process Communication	
2.1 Process Definition	14
2.2 Process Communication Overview	15
2.2.1 Named Pipes	17
2.2.2 Semaphore	18
2.2.3 Shared Memory	18
2.2.4 Remote Procedure Call (RPC)	19
2.2.5 Socket	20
2.2.6 Remote Method Invocation (RMI)	22
2.3 Java Messaging Systems	24

Chapter Three: Message Passing Interface (MPI)

3.1 Introduction to Message Passing Interface (MPI)	26
3.2 MPI Advantages	28
3.3 MPI Basic Concepts	29
3.3.1 Point to Point Communications	30
3.3.1.1 Buffering	31
3.3.1.2 Parts of MPI Message	32
3.3.2 Collective Communications	33
3.3.3 Process Groups	34
3.3.4 Communication Contexts	35
3.3.5 Process Topologies	36
3.3.6 Bindings for Fortran and C	36
3.3.7 Environmental management and inquiry	36
3.3.8 Profiling Interface	37
3.4 Java Binding of MPI	38
3.5 Message Passing in JMPI	39
3.6 JMPI Architecture	40
3.6.1 MPI Application Programmer's Interface	41
3.6.2 Communication Layer	43
3.6.3 Java Virtual Machine (JVM)	46

Chapter Four: Design and Implementation of JMPI

Middleware System

4.1 Introduction	47
4.2 JMPI Middleware System	48
4.3 JMPI Middleware System Applications	54
4.3.1 Range Addition Application	56

4.3.2 Matrix-Vector Multiplication Application	58
4.3.3 Gauss Elimination Application	60
4.4 JMPI Middleware System Tests	65
Chapter Five: Conclusions and Future Work	
5.1 Conclusions	76
5.2 Future Work	77
References	78
Appendix A	

List of Abbreviations

API	Application Programming Interface
CSR	Compressed Sparse Row
DSG	Distributed Systems Group
FIFO	First In First Out
GUI	Graphical User Interface
IBM	International Business Machine
I/O	Input / Output
IPC	InterProcess Communication
JDK	Java Development Kit
JMPI	Java Message Passing Interface
JNI	Java Native Interface
JVM	Java Virtual Machine
LAM/MPI	Local Area Multicomputer / Message Passing Interface
LAN	Local Area Network
MBCF	Memory-Based Communication Facilities
MIMD	Multiple Instruction, Multiple Data
MPI	Message Passing Interface
MPIF	Message Passing Interface Forum
MPL	Message Passing Library
NAS	Numerical Aerodynamic Simulation
NIO	New Input Output
NOWs	Networks of Workstations
NPAC	Northeast Parallel Architecture Center
ORNL	Oak Ridge National Laboratory

PCG	Precondition Conjugate Gradient
PVM	Parallel Virtual Machine
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RSH	Remote SHell
Rshd	Remote shell daemon
SPCs	Scalable Parallel Computers
SPMD	Single Program, Multiple Data
SSI	System Service Interface
TCP	Transmission Control Protocol
URL	Uniform Resource Locator
WAN	Wide Area Network

List of Figures

Figure Name	page number
Figure (1-1): The message-passing programming paradigm	4
Figure (2-1): Principle of RPC between a Client and Server Program	20
Figure (2-2): Connection Request	21
Figure (2-3): Communication between Client and Server	21
Figure (3-1): Four process Distributed Parallel Application	40
Figure (3-2): JMPI Architecture	41
Figure (3-3): MPI API Class Organization	43
Figure (4-1): JMPI Middleware System	48
Figure (4-2): JMPI Process Communication in LAN	50
Figure (4-3): Flowchart of JMPI Middleware model	51
Figure (4-4): Flowchart of JMPI Middleware Installation	52
Figure (4-5): Flowchart of Running Applications Using JMPI Middleware System	53
Figure (4-6): Warning Window	65
Figure (4-7): Exit Window	65
Figure (4-8): Installed JMPI Middleware System	66
Figure (4-9): JMPI Middleware System Applications Window	66
Figure (4-10): checked Serial option Window	67
Figure (4-11): One computer option window	67
Figure (4-12): LAN option window	68
Figure (4-13): Range Addition Application window	68
Figure (4-14): Matrix-Vector Multiplication Application window	69

Figure (4-15): Gauss Elimination Application window	69
Figure (4-16): Help window	70
Figure (4-17): Time comparison in execution of Serial and Parallel (one computer) of Range Addition application	71
Figure (4-18): Time comparison in execution of Serial and Parallel (LAN) of Range Addition application	72
Figure (4-19): Time comparison in execution of Serial and Parallel (one computer) of Matrix-Vector Multiplication application	73
Figure (4-20): Time comparison in execution of Serial and Parallel (LAN) of Matrix-Vector Multiplication application	74
Figure (4-21): Time comparison in execution of Serial and Parallel (one computer) of Gauss Elimination application	75
Figure (4-22): Time comparison in execution of Serial and Parallel of Gauss Elimination application	75



Chapter One
Introduction

Chapter One

Introduction

1.1 Parallel Computing

A *parallel computer* simply comprises a number of processes that work together to solve a computational problem. There are a number of different types of computers and classifications are made on the basis of both instruction/data stream characteristics and memory architectures [Baq06].

While, *parallel computing* is the simultaneous execution of the same task (split up and specially adapted) on multiple processors in order to obtain results faster. The idea is based on the fact that the process of solving a problem usually can be divided into smaller tasks, which may be carried out simultaneously with some coordination [Wik06].

In the simplest sense, *parallel computing* is the simultaneous use of multiple computer resources working together to solve a computational problem. To run a problem using multiple processors, a problem is broken into discrete parts that can be solved concurrently, and then each part is further broken down to a series of instructions. After that the instructions from each part execute simultaneously on different processors. The primary reasons for using parallel computing are [Bar06]:

- Save time.
- Solve larger problems.

- Provide concurrency (do multiple things at the same time).
- Taking advantage of non-local resources - using available computer resources on a Wide Area Network (WAN), or even the Internet when local computer resources are scarce.
- Cost savings - using multiple "cheap" computing resources instead of paying for time on a supercomputer.
- Overcoming memory constraints - single computers have very finite memory resources. For large problems, using the memories of multiple computers may overcome this obstacle.

A parallel programming model is a set of software technologies to express parallel algorithms and match applications with the underlying parallel systems. It encloses the areas of applications, languages, compilers, libraries, communication systems, and parallel Input/Output (I/O). People have to choose a proper parallel programming model or a form of mixture of them to develop their parallel applications on a particular platform. Parallel programming models are implemented in several ways: as libraries invoked from traditional sequential languages, as language extensions, or complete new execution models [Wik06].

There are five parallel programming models in common use. They are [Bar06]:

1. Shared Memory: tasks share a common address space, which they read and write asynchronously. Various mechanisms such as locks and semaphores may be used to control access to the shared memory.
2. Threads: a single process can have multiple, concurrent execution paths. Perhaps the simplest analogy that can be used to describe threads is the concept of a single program that includes a number of subroutines.

3. Message Passing: A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine as well across an arbitrary number of machines. Tasks exchange data through communications by sending and receiving messages. Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation. This model is described in more details in the following sections because it is implemented in the proposed system.
4. Data Parallel: A set of tasks work collectively on the same data structure, however, each task works on a different partition of the same data structure.
5. Hybrid: any two or more parallel programming models are combined. Like the combination of the message passing model with either the threads model or the shared memory model. Another common example of a hybrid model is combining data parallel with message passing.

1.2 Message Passing Model

Message Passing model is one of the techniques for communicating between parallel processes. A common use of message passing is for communication in a parallel computer. A process running on one processor may send a message to a process running on the same processor or another. The actual transmission of the message is usually handled by the run-time support of the language in which the processes are written, or by the operating system [Far05].

The message passing is a model of parallel programming. Several instances of the sequential paradigm are considered together. That is, the programmer imagines several processors, each with its own memory space, and writes a program to run on each processor. But parallel programming by definition

Message passing is widely used on parallel computers with distributed memory, and on clusters of servers. The advantages of using message passing include [Hew03]:

- Portability: Message passing is implemented on most parallel platforms.
- Universality: Model makes minimal assumptions about underlying parallel hardware. Message passing libraries exist on computers linked by networks and on shared and distributed memory multiprocessors.
- Simplicity: Model supports explicit control of memory references for easier debugging.

The principle drawback of message passing is the responsibility it places on the programmer. The programmer must explicitly implement a data distribution scheme and all InterProcess Communication (IPC) techniques and synchronization. In so doing, it is the programmer's responsibility to resolve data dependencies and avoid deadlock and race conditions [Mau95].

1.3 Message Passing Libraries

The set of communication operations that are allowed by an implementation of the message passing model form the components of a message passing library. Examples of message passing libraries include public domain packages that do not target a specific machine (Parallel Virtual Machine (PVM), Message Passing Interface (MPI), etc.) as well as machine dependent vendor implementations (Message Passing Library (MPL)). Until recently, users of message passing libraries had to choose between using public domain packages for improved code portability, and vendor implementations for improved performance on a given machine. The MPI Library has been developed to meet the dual goals of portability and performance on a wide range of machines. There are three main criteria for choosing a MPL [Mau95]:

- Performance - latency and bandwidth
- Portability
- Functionality

Latency is time to transmit 0 length message, while bandwidth is the amount of data that can be communicated per unit of time. Commonly expressed as megabytes/sec. Latency and bandwidth vary greatly from machine to machine. The best performance on a specific machine is typically obtained from the native message passing library written specifically for that machine [Bar06].

1.4 Message Passing Interface (MPI) Overview

MPI is a MPL standard based on the consensus of the MPI Forum (MPIF), which has over 40 participating organizations, including vendors, researchers, software library developers, and users. The goal of the MPI is to establish a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs [Bar05].

MPIF started working on the standard in 1992. The first draft (Version 1.0), which was published in 1994, was strongly influenced by the work at the International Business Machine (IBM) Thomas J. Watson Research Center. MPIF has further enhanced the first version to develop a second version (MPI-2) in 1997. The next release of the first version (Version 1.2) is offered as an update to the previous release and is contained in the MPI-2 document. The design goal of MPI is quoted from “MPI: A Message-Passing Interface Standard (Version 1.1)” as follows [Aoy99]:

- Design an Application Programming Interface (API) (not necessarily for compilers or a system implementation library).

- Allow efficient communication: Avoid memory-to-memory copying and allow overlap of computation and communication and offload to communication co-processor, where available.
- Allow for implementations that can be used in a heterogeneous environment.
- Allow convenient C and Fortran bindings for the interface.
- Assume a reliable communication interface: The user need not cope with communication failures. Such failures are dealt with by the underlying communication subsystem.
- Define an interface that is not too different from current practice, such as PVM and provides extensions that allow greater flexibility.
- Define an interface that can be implemented on many vendors' platforms, with no significant changes in the underlying communication and system software.
- Semantics of the interface should be language independent.
- The interface should be designed to allow for thread-safety.

Although several message-passing libraries exist on different systems, MPI is popular for the following reasons [Hew03]:

- Support for full asynchronous communication: process communication can overlap process computation.
- Group membership: processes may be grouped based on context.
- Synchronization variables that protect process messaging: When sending and receiving messages, synchronization is enforced by source and destination information, message labeling, and context information.
- Portability: All implementations are based on a published standard that specifies the semantics for usage.

1.5 Literature Survey

Various efforts in the field of message passing libraries were introduced; some of these efforts are summarized below:

1. Morimoto [Mor99], “Implementing Message Passing Communication with a Shared Memory Communication Mechanism”.

This thesis describes a high-performance implementation of MPI for a message passing communication library and uses the Memory-Based Communication Facilities (MBCF). This implementation, called MPI/MBCF, combines two protocols to utilize the shared memory communication mechanism: the **write** protocol and the **eager** protocol. In the **write** protocol, Remote Write is used for communication with no buffering. In the **eager** protocol, Memory-Based First In First Out (FIFO) is used for buffering by the library.

These two protocols are switched autonomously according to the precedence of the send and receive functions. The performance of the MPI/MBCF was evaluated on a cluster of workstations. The round-trip time and the peak bandwidth were measured the Numerical Aerodynamic Simulation (NAS) Parallel Benchmarks were executed. The results show that a MPL achieves high performance by using a shared memory communication mechanism.

2. Morin [Mor00],”JMPI: Implementing the Message Passing Standard in Java”.

Java MPI (JMPI) was an experimental implementation of MPI developed at the architecture and real-time lab at the University of Massachusetts. This library implemented a large subset of MPI’s functionality. It proposed that MPI and Java are complementary technologies and develop a reference implementation written

completely in Java. This library supports the transfer of multi-dimensional arrays through the use of introspection. The library had no runtime infrastructure to support bootstrapping parallel processes on remote hosts. It used Remote Method Invocation (RMI) as the communication medium and supports the transfer of Java objects using the object serialization.

3. Mande [Man02], “Performance Prediction of Message Passing Communication in Distributed Memory Systems”.

Communication is an important component in determining the overall performance of a distributed memory parallel computing application. It therefore becomes essential to predict communication performance of applications on the underlying network hardware of a target distributed memory system. This thesis concentrates on integrating a cycle driven k-ary n-cube network simulator to the existing execution driven distributed memory simulator running message passing called CAL-SIM for evaluating communication performance of message passing applications. The design and implementation of a suitable network interface required for the integration is presented. With detailed network simulation, the accuracy of predictions made is very high.

The impact on communication performance by varying some of the network design parameters is studied. Other important aspect of this work is the access to an evaluation platform for evaluating network design tradeoffs, using real applications as workload instead of synthetic workloads.

4. Squyres [Squ04],” A Component Architecture for the MPI: The System Services Interface (SSI) of Local Area Multicomputer/ Message Passing Interface (LAM/MPI)”.

This work presents the design and implementation of component system architecture in LAM/MPI, a production quality, open source implementation of the MPI-1 and MPI-2 standards. Previous versions of LAM/MPI, as well as other MPI implementations, are based on monolithic software architectures that – regardless of how well-abstracted and logically constructed – are highly complex software packages, presenting a steep learning curve for new developers and third parties. The current version of LAM/MPI has been re-architected to utilize a component system architecture consisting of four component frameworks and a Meta framework that ties them together. Each component framework was designed from analysis of prior monolithic implementations of LAM/MPI and represents a major functional category: run-time environment startup, MPI point-to-point communication, MPI collective communication, and parallel checkpoint/restart. The result is an MPI implementation that is highly modular, has published abstraction and interface boundaries, and is significantly easier to develop, maintain, and use as a vehicle for research. Performance results are shown demonstrating that this component-based approach provides identical (if not better) performance compared to prior monolithic-based implementations. But it found that the module frameworks themselves can be used to explore their respective domains. Point-to-Point and collective message passing, for example, still have many unanswered questions.

The design of the MPI specification was specifically intended to allow multi-threaded MPI applications. Fault tolerance is also becoming increasingly important. High failure rates can render large computational resources effectively

useless, unless models can be determined that allow some form of continued operation in the presence of faults.

5. Baker, et.al [Bak04], “A Status Report: Early Experiences with the implementation of a Message Passing System using Java NIO”.

Since its release in 1996, Java has become a popular software development language. The reasons for its popularity can be attributed to the easy-to-use syntax, its portability, the extensive set libraries, and the support of object-oriented features like data hiding and polymorphism. One of the main drawbacks of Java was the blocking I/O package, but the situation has improved with the addition of the Java New I/O (NIO) package that adds scalable and non-blocking I/O to the language. The Distributed Systems Group (DSG) implemented a Java message passing system based on Java NIO package that runs on heterogeneous environment. In this report, they discussed and evaluated their reference implementation, known as MPJ.

One of the obvious advantages of such a message passing system is a portable approach to problem solving using heterogeneous operating systems and hardware without compromising the overall communication performance. MPJ follows a layered structure that allows enhancements to the existing infrastructure. This also allows the higher communication layers to swap in various device drivers to make use of specialized hardware or protocols. The mpjdev device driver implemented as part of this project has an efficient buffering API that is used to pack/unpack the data to/from the buffer. mpjdev implements three communication protocols, inter-process, eager-send and the rendezvous protocol. mpjdev provides a simple interface that provides the basic functionality for starting up the device, setting up the communication infrastructure and sending/receiving the data to/from the other peers.

6. Baqer [Baq06], “A Multiprocessing Computer System for the Finite Element Analysis”.

This work uses MPI and describes the parallel implementation of the Precondition Conjugate Gradient (PCG) iterative method for the solution of large linear equation systems resulting from the finite element method. A diagonal Jacobi preconditioner is used in order to accelerate the convergence. Parallel implementation of the Gaussian Elimination and back substitution method with different techniques, block 1-D partitioning and row wise partitioning are presented and compared with sequential implementation of the method. Different types of matrix storage schemes are implemented such as the Compressed Sparse Row (CSR) to achieve better performance. An automatic mesh generator is built using C++ programming language. The code was tested on a machine with dual processor. The same accuracy was obtained for the serial and parallel code.

The results show that the CSR format reduces computation time compared to the order format. The parallel run time was reduced to 66% from the sequential time. Good performance is achieved with MPI. The PCG method (serial or parallel) is better for the solution of large linear system (sparse matrices) than Gaussian Elimination and back substitution method.

1.6 Thesis's Objective

The objective of this project is to implement a system using one of the reference implementations of MPI systems that follows the recommended standard API, which includes a library and infrastructure that can provide some support needed by parallel applications. The proposed system supposes to implement a middleware system to support a MPI API based on a Java binding MPI which uses one of the Java IPC techniques.

1.7 Thesis's Outlines

In this section, the contents of individual chapters of this thesis are briefly reviewed:

Chapter 2: Presents definitions and types of IPC.

Chapter 3: Describes the concepts of the theoretical part of the project that provides a more in-depth background of MPI and JMPI package.

Chapter 4: Describes the implementation of the proposed system and presents test and results of the proposed system.

Chapter 5: Introduces the conclusions and suggestions for future work.



Chapter Two
Process Communication

Chapter Two

Process Communication

2.1 Process Definition

A process is a set of executable instructions (program) which runs on a processor. One or more processes may execute on a processor. In a message passing system, all processes communicate with each other by sending messages - even if they are running on the same processor. In other words, a process is an instance of a program running in a computer. It is close in meaning to task, a term used in some operating systems. In Unix and some other operating systems, a process is started when a program is initiated (either by a user entering a shell command or by another program). Like a task, a process is a running program with which a particular set of data is associated so that the process can be kept track of. An application that is being shared by multiple users will generally have one process at some stage of execution for each user. A process can initiate a subprocess, which is called a child process (and the initiating process is sometimes referred to as its parent). A child process is a replica of the parent process and shares some of its resources, but cannot exist if the parent is terminated [Bar05, Tec06].

2.2 Process Communication Overview

When processes interact with one another, two fundamental requirements must be satisfied: synchronization and IPC. Processes need to be synchronized to enforce mutual exclusion; cooperating processes may need to exchange information.

The communication of a message between two processes implies some level of synchronization between the two processes: The receiver can not receive a message until it has been sent by another process. In addition, sender and receiver need to specify what happens to a process after it issues a *send* or *receive* primitive [Sta98].

Consider the *send* primitive first. When a *send* primitive is executed in a process, there are two possibilities: Either the sending process is blocked until the message is received, or it is not. Similarly, when a process issues a *receive* primitive, there are two possibilities [Sta98]:

1. If a message has previously been sent, the message is received and execution continues.
2. If there is no waiting message, then either: (a) the process is blocked until the message arrive. Or (b) the process continues to execute, abandoning the attempt to receive.

Processes can exchange information or synchronize their operation through several methods of IPC, which is a capability supported by some operating systems that allows a process to communicate with another one. The processes can be running on the same computer or on different computers connected through a network. IPC enables one application to control another one, and for several applications to share the same data without interfering with one another.

IPC is required in all multiprocessing systems, but it is not generally supported by single-process operating systems [Jup05].

IPC is a set of programming interfaces that allows a programmer to coordinate activities among different program processes that can run concurrently in an operating system. This allows a program to handle many user requests at the same time. Since even a single user request may result in multiple processes running in the operating system on the user's behalf, the processes need to communicate with each other. The IPC interfaces make this possible. Each IPC method has its own advantages and limitations so it is not unusual for a single program to use all of the IPC methods [Tec05].

IPC should provide the following services [Web1]:

- Protocol for coordinating sending and receiving of data between processes.
- Queuing mechanism to enable data to be entered asynchronously and faster than it is processed.
- Support for many-to-one exchanges (a server dealing with many clients).
- Network support, location independence, integrated security, and recovery.
- Remote procedure support to invoke a remote application service.
- Support for complex data structures.
- Standard programming language interface.

All these services should be implemented with little code and excellent performance.

The following are some of the common IPC methods:

1. Named Pipes.
2. Semaphores.
3. Shared Memory.
4. Remote Procedure Call (RPC).
5. Socket.
6. Remote Method Invocation (RMI).

2.2.1 Named Pipes

A named pipe is a method for passing information from one computer process to other processes using a pipe or message holding place that is given a specific name. Unlike a regular pipe, a named pipe can be used by processes that do not have to share a common process origin and the message sent to the named pipe can be read by any authorized process that knows the name of the named pipe. A named pipe is sometimes called a First In-First Out (FIFO) because the first data written to the pipe is the first data that is read from it [Tec05].

named pipes support peer-to-peer processing through the provision of two-way communication between unrelated processes on the same machine or across the Local Area Network (LAN). The server creates the pipe and waits for clients to access it. A useful compatibility feature of named pipes supports standard Operating System (OS/2) file service commands for access. Multiple clients can use the same named pipe concurrently. named pipes are easy to use, compatible with the file system, and provide local and remote support. named pipes provide strong support for many-to-one IPCs [Web1].

2.2.2 Semaphore

Semaphores are a technique for coordinating or synchronizing activities in which multiple processes compete for the same operating system resources. A semaphore is a value in a designated place in operating system (or kernel) storage that each process can check and then change. Depending on the value that is found, the process can use the resource or will find that it is already in use and must wait for some period before trying again. Semaphores can be binary (0 or 1) or can have additional values. Typically, a process using semaphores checks the value and then, if it using the resource, changes the value to reflect this so that subsequent semaphore users will know to wait. Semaphores are commonly used for two purposes: to share a common memory space and to share access to files [Tec03].

2.2.3 Shared Memory

In computer programming, shared memory is a method by which program processes can exchange data more quickly than by reading and writing using the regular operating system services. For example, a client process may have data to pass to a server process that the server process is to modify and return to the client. Ordinarily, this would require the client writing to an output file (using the buffers of the operating system) and the server then read that file as an input from the buffers to its own work space. Using a designated area of shared memory, the data can be made directly accessible to both processes without having to use the system services. To put the data in shared memory, the client gets access to shared memory after checking a semaphore value, writes the data, and then resets the semaphore to signal to the server (which periodically checks shared memory for possible input) that data is waiting. In turn, the server

process writes data back to the shared memory area, using the semaphore to indicate that data is ready to be read [Tec05].

Shared memory provides IPC when the memory is allocated in a named segment. Any process that knows the named segment can share it. Each process is responsible for implementing synchronization techniques to ensure integrity of updates. Tables are typically implemented in this way to provide rapid access to information that is infrequently updated [Web1].

2.2.4 Remote Procedure Call (RPC)

RPC means a slice of code in a client application that invokes a procedure on the server application. RPC is the method that modern middleware often replaces because they require programmers to rewrite them over and over when wiring a bunch of applications together. Other middleware approaches are often more efficient as the number of applications grows [CIO05].

The RPC facility provides for the invocation and execution of requests from processors running different operating systems and using different hardware platforms from the caller's. The standardized request form provides the capability for data and format translation in and out. These standards are evolving and being adopted by the industry [Web1]. Figure (2-1) shows the communication between a Client and Server in RPC Method [Web2].

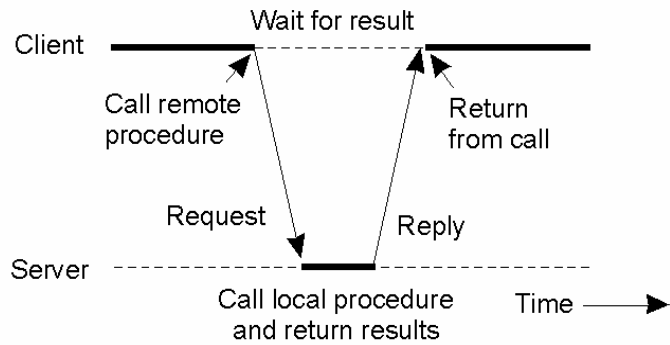


Figure (2-1): Principle of RPC between a Client and Server Program

2.2.5 Socket

A socket is a method for communicating between a client program and a server program in a network. A socket is defined as "the endpoint in a connection". Sockets are created and used with a set of programming requests or "function calls" sometimes called the sockets API. Sockets can also be used for communication between processes within the same computer [Tec01].

In other words, a socket is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the Transmission Control Protocol (TCP) layer can identify the application that data is destined to be sent. Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request. On the client-side; the client knows the hostname of the machine on which the server is running and the port number to which the server is connected. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. Figure (2-2) shows that the original port is used for accepting connection requests from other clients [Sun07, Tec01].

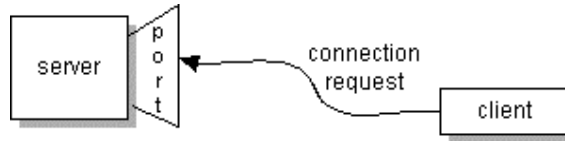


Figure (2-2): Connection Request

If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to a different port. It needs a new socket (and consequently a different port number) so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client. On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server. Note that the socket on the client side is not bound to the port number used to rendezvous with the server. Rather, the client is assigned a port number local to the machine on which the client is running. Figure (2-3) shows that the new port is used to communicate with each connected client. The client and server can now communicate by writing to or reading from their sockets [Sun07].



Figure (2-3): Communication between Client and Server

2.2.6 Remote Method Invocation (RMI)

RMI is a mechanism which invokes a method on an object that exists in another address space. The other address space could be on the same machine or on a different one. The RMI mechanism is basically an object-oriented mechanism. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a *distributed object application* [Bac98, Sun05].

There are three processes that participate in supporting RMI [Bac98]:

1. A *Client* is the process that is invoking a method on a remote object.
2. A *Server* is the process that owns the remote object. The remote object is an ordinary object in the address space of the server process.
3. An *Object Registry* is a name server that relates objects with names. Objects are registered with the Object Registry. Once an object has been registered, one can use the Object Registry to obtain access to a remote object using the name of the object.

RMI uses sockets as the underlying communication medium and is primarily meant for client server interactions rather than the distributed peer processes. RMI can save a lot of programming time and effort for the developers of a message passing system, but is not the best option because of the performance issues associated with RMI. One of the reasons for these issues includes sending the basic data-types as objects, because all the arguments to the remote methods should be serializable. Secondly, at least one RMI registry should be running to locate distributed objects. The address and port of the RMI registry needs to be known to all processes that have to query the registry to locate distributed objects [Bak04].

Distributed object applications need to [Sun05]:

- **Locate remote objects:** Applications can use one of two mechanisms to obtain references to remote objects. An application can register its remote objects with RMI's simple naming facility, the *rmiregistry*, or the application can pass and return remote object references as part of its normal operation.
- **Communicate with remote objects:** Details of communication between remote objects are handled by RMI; to the programmer, remote communication looks like a standard Java method invocation.
- **Load class bytecodes for objects that are passed around:** Because RMI allows a caller to pass objects to remote objects, RMI provides the necessary mechanisms for loading an object's code, as well as for transmitting its data.

The primary advantages of RMI are [Sun07]:

1. **Object Oriented:** RMI can pass full objects as arguments and return values, not just predefined data types.
2. **Mobile Behavior:** RMI can move behavior (class implementations) from client to server and server to client.
3. **Design Patterns:** Passing objects lets the programmer use the full power of object oriented technology in distributed computing, such as two- and three-tier systems.
4. **Safe and Secure:** RMI uses built-in Java security mechanisms that allow the programmer's system to be safe when users downloading implementations.
5. **Easy to Write/Easy to Use:** RMI makes it simple to write remote Java servers and Java clients that access those servers.

6. **Connects to Existing/Legacy Systems:** RMI interacts with existing systems through Java's Native Interface (JNI) method.
7. **Write Once, Run Anywhere:** RMI is part of Java's "Write Once, Run Anywhere" approach. Any RMI based system is 100% portable to any Java Virtual Machine (JVM).
8. **Distributed Garbage Collection:** RMI uses its distributed garbage collection feature to collect remote server objects that are no longer referenced by any client in the network.
9. **Parallel Computing:** RMI is multi-threaded, allowing the programmer's servers to exploit Java threads for better concurrent processing of client requests.
10. **The Java Distributed Computing Solution:** RMI is part of the core Java platform.

2.3 Java Messaging Systems

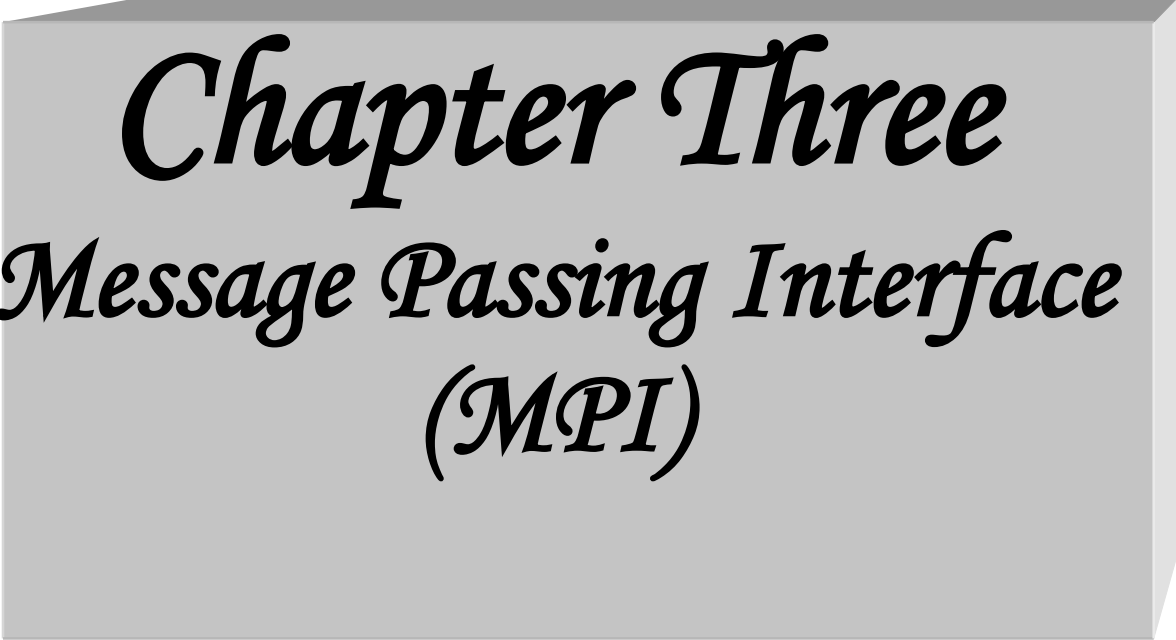
There has been a significant amount of effort in developing Java message passing systems. Most of the systems were experimental and are no longer supported or in other cases, the software is not available. The approaches used to implement a Java messaging system can be divided into three categories [Bak04]:

1. **Using JNI:** It is a Java API that allows programmers to call C routines from their applications. Often developers of message passing systems use this package to interface their Java code to an underlying native MPI implementation. This technique saves a lot of additional programming and testing efforts but does not result in a portable code, which is the primary reason for implementing a message-passing system in pure Java. JNI also introduces an additional copying of the data between the

Java and the native MPI code. Moreover, using JNI breaks the programming model of Java because there is no way to ensure code type safety. It also may lead to memory leaks because in C unlike Java, the programmer is responsible for allocating and freeing the memory.

2. **Using Sockets.**

3. **Using RMI.**



Chapter Three
Message Passing Interface
(MPI)

Chapter Three

Message Passing Interface (MPI)

3.1 Introduction to Message Passing Interface (MPI)

MPI is a widely adopted communication library for parallel and distributed computing. It provides an infrastructure for users to build high performance distributed computing environment using simple, high-level message-passing primitives. It is portable, and has been implemented on many platforms and in parallel machines. Although most of the existing MPI standard specifies language bindings for Fortran, C and C++, there are a number of implementations of MPI provided for Java [Wan01].

MPIF sought to make use of the most attractive features of a number of existing message-passing systems, rather than selecting one of them and adopting it as the standard. They identified some critical shortcomings of existing message-passing systems, in areas such as complex data layouts or support for modularity and safe communication. This led to the introduction of new features in MPI. The MPI standard defines the user interface and functionality for a wide range of message-passing capabilities. Since its completion in June of 1994, MPI has become widely accepted and used. Implementations are available on a range of machines from Scalable Parallel Computers (SPCs) with distributed memory to Networks Of Workstations (NOWs). A growing number of SPCs have an MPI supplied and supported by the vendor. Because of this, MPI has achieved one of its goals - adding

credibility to parallel computing. Third party vendors, researchers, and others have a reliable and portable way to express message-passing, parallel programs [Sni95].

MPI is a standard protocol in terms of user interface. It uses the same function names, and MPI functions are called and arguments are passed in the same way across different platforms. It hides the underlying hardware implementation of how these functions work. MPI functions are called and pass their arguments in the same way regardless of the platform. MPI is the leading message passing programming paradigm. The public domain implementation was written at Argonne National Laboratory and is currently available for virtually all major computer architectures [Lat97].

MPI supports process grouping capability and allows the programmer to [NCS02]:

- Organize tasks based upon application nature into task groups.
- Enable Collective Communications operations across a subset of related tasks.
- Provide basis for implementing virtual communication topologies.
- Ensure the communication safety and handle the application complexity.

This standard is intended for use by all those who want to write portable message-passing programs. This includes individual application programmers, developers of software designed to run on parallel machines, and creators of environments and tools. In order to be attractive to this wide audience, the standard must provide a simple, easy-to-use interface for the basic user while not semantically precluding the high-performance message-passing operations available on advanced machines [MPI94].

There have been efforts to provide MPI for Java language. Existing approaches to MPI for Java can be grouped into two types [Wan01]:

1. Native MPI bindings where some native MPI library is called by Java programs through Java wrappers:

The native MPI binding approach provides efficient MPI communication through calling native MPI methods. Conflicts could arise on the use of system resources such as signals between the MPI library and JVM.

2. Pure Java implementations:

The pure Java implementation approach on the other hand can provide a portable MPI implementation since the whole MPI library is rewritten in Java, but the MPI communication would be relatively less efficient since Java operates at a higher level.

3.2 MPI Advantages

The main advantages of MPI are [Lat97, Sni95]:

- MPI provides a high degree of portability. An MPI source code can be ported to different platforms, compiled, and run without modification as long as the MPI library is available on the system.
- MPI can run jobs across heterogeneous systems where a mixture of processors of different architecture is clustered together. The MPI library transparently does the appropriate data conversion when data are sent between different systems.
- MPI standards are flexible. They specify what the MPI functions should do, but it is left to vendors how to implement them in the most efficient way that meets the standards.

- MPI is widely supported by most vendors of parallel systems, who have developed highly optimized native implementations for systems.
- MPI offers a high degree of functionality with over 100 routines implemented in the MPI library.
- MPI was designed to encourage overlap of communication and computation, so as to take advantage of intelligent communication agents, and to hide communication latencies. This is achieved by the use of nonblocking communication calls, which separate the initiation of a communication from its completion.
- MPI allows or supports scalability through several of its design features. For example, an application can create subgroups of processes that, in turn, allow collective communication operations to limit their scope to the processes involved.
- MPI, as all good standards, is valuable in that it defines a known, minimum behavior of message-passing implementations. This relieves the programmer from having to worry about certain problems that can arise. One example is that MPI guarantees that the underlying transmission of messages is reliable. The user need not check if a message is received correctly.

3.3 MPI Basic Concepts

An MPI program consists of a set of processes and a logical communication medium connecting those processes. An MPI process cannot directly access memory in another MPI process. IPC requires calling MPI routines in both processes. MPI defines a library of routines through which MPI processes communicate [Hew03].

The MPI standard routines provide a set of functions that support the following [MPI03]:

- Point-to-point communications.
- Collective communications.
- Process groups.
- Communication contexts.
- Process topologies.
- Bindings for Fortran and C.
- Environmental management and inquiry.
- Profiling interface.

3.3.1 Point to Point Communications

Point-to-point: is the basic communication pattern in MPI. As the name implies, point-to-point communications handle data transmission between any two processes in a communicator. Only two processes are involved: one sends the data and the other receives it. Most MPI communications are built around this basic point-to-point communications [NCS01].

There are different types of send and receive routines used for different purposes. Some of them are [Bar05]:

- Synchronous send.
- Blocking send / blocking receive.
- Non-blocking send / non-blocking receive.
- Buffered send.
- Combined send/receive.
- "Ready" send.

Any type of send routine can be paired with any type of receive routine. MPI also provides several routines associated with send - receive operations, such as those used to wait for a message's arrival or probe to find out if a message has arrived.

Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode [Bar05]:

- **Blocking Communication:** A communication routine is blocked if the completion of the call is dependent on certain "events". For sends, the data must be successfully sent or safely copied to system buffer space so that the application buffer that contained the data is available for reuse. For receives, the data must be safely stored in the receive buffer so that it is ready for use.
- **Non-blocking Communication:** A communication routine is non-blocking if the call returns without waiting for any communications events to complete (such as copying of message from user memory to system memory or arrival of message). It is not safe to modify or use the application buffer after completion of a non-blocking send. It is the programmer's responsibility to insure that the application buffer is free for reuse. Non-blocking communications are primarily used to overlap computation with communication to effect performance gains.

3.3.1.1 Buffering

In a perfect world, every send operation would be perfectly synchronized with its matching receive. This is rarely the case. Somehow or other, the MPI implementation must be able to deal with storing data when the two tasks are out of synchronization. Consider the following two cases [Bar05]:

- A send operation occurs 5 seconds before the receive is ready - where is the message while the receive is pending?
- Multiple sends arrive at the same receiving task which can only accept one send at a time - what happens to the messages that are "backing up"?

The MPI implementation (not the MPI standard) decides what happens to data in these types of cases. Typically, a system buffer area is reserved to hold data in transit. System buffer space is [Bar05]:

- Opaque to the programmer and managed entirely by the MPI library
- A finite resource that can be easy to exhaust
- Often mysterious and not well documented
- Able to exist on the sending side, the receiving side, or both
- Something that may improve program performance because it allows send - receive operations to be asynchronous.

User managed address space (i.e. user's program variables) is called the application buffer. MPI also provides for a user managed send buffer.

3.3.1.2 Parts of MPI Message

The arguments passed to an MPI function distinguish two parts of an MPI message [NCS02]:

1. Message data, which describes the actual data in the message
2. Message envelope, which contains extra information to help deliver the message in a **SEND** and insure it is the right message to receive in a **RECV** function.

3.3.2 Collective Communications

As the name implies, collective communications refers to those MPI functions involving all the processes within the defined communicator group. Collective communications are mostly built around point-to-point communications. Several features distinguish collective communications from point-to-point communications, which are [NCS02]:

- A collective operation requires that all processes within the communicator group call the same collective communication function with matching arguments.
- The size of data sent must exactly match the size of data received. In point-to-point communications, a sender buffer may be smaller than the receiver buffer. In collective communications they must be the same.
- Except for synchronization routines, MPI collective communication functions are not synchronizing as set by the MPI standards.
- Collective communications exist in blocking mode only. Blocking here means that a process will block until its role in the collective communication is complete, no matter what the completion status is of the other processes participating in the communications.
- Collective operations do not use the tag field. They are matched according to the order they are executed.

There are three types of collective communication [Bar05]:

1. Synchronization: processes wait until all members of the group have reached the synchronization point.
2. Data Movement: broadcast, scatter/gather, all to all.

3. Collective Computation (reductions): one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.

There are number of considerations and restrictions that must be considered by the programmer in collective communication routines, some of those are [Bar05]:

- Collective operations are blocking.
- Collective communication routines do not take message tag arguments.
- Collective operations within subsets of processes are accomplished by first partitioning the subsets into new groups and then attaching the new groups to new communicators.
- Collective operations can only be used with MPI predefined datatypes - not with MPI Derived Data Types.

3.3.3 Process Groups

A process group is an ordered collection of processes, and each process is uniquely identified by its rank within the ordering. For a group of n processes the ranks run from 0 to $n-1$. This definition of groups closely conforms to current practice.

Process groups can be used in two important ways. First, they can be used to specify which processes are involved in a collective communication operation, such as a broadcast. Second, they can be used to introduce task parallelism into an application, so that different groups perform different tasks. If this is done by loading different executable codes into each group, then the programmers refer to this as Multiple Instruction, Multiple Data (MIMD) task

parallelism. Alternatively, if each group executes a different conditional branch within the same executable code, then this will be referred as Single Program, Multiple Data (SPMD) task parallelism (also known as control parallelism) [MPI93].

3.3.4 Communication Contexts

Communication contexts were initially proposed to allow the creation of distinct, separable message streams between processes, with each stream having a unique context. A common use of contexts is to ensure that messages sent in one phase of an application are not incorrectly intercepted by another phase. The point here is that the two phases may actually be calls to two different third-party library routines, and the application developer has no way of knowing if the message tag, group, and rank completely disambiguate the message traffic of the different libraries from one another and from the rest of the application. Context provides an additional criterion for message selection, and hence permits the construction of independent message tag spaces [MPI93].

A communicator encompasses a group of processes that may communicate with each other. All MPI messages must specify a communicator. In the simplest sense, the communicator is an extra "tag" that must be included with MPI calls. Like groups, communicators are represented within system memory as objects and are accessible to the programmer only by "handles" [Bar05].

Within a communicator, every process has its own unique, integer identifier assigned by the system when the process initializes. A rank is sometimes also called a "process ID". Ranks are contiguous and begin at zero and they are used by the programmer to specify the source and destination of messages. Often

used conditionally by the application to control program execution (if rank=0 do this / if rank=1 do that) [Bar05].

3.3.5 Process topologies

A topology is an extra, optional attribute that one can give to an intra-communicator; topologies cannot be added to inter-communicators. A topology can provide a convenient naming mechanism for the processes of a group (within a communicator), and additionally, may assist the runtime system in mapping the processes onto hardware [Sni95].

MPI provides a means for ordering the processes of a group into topologies. MPI topologies are virtual - there may be no relation between the physical structure of the parallel machine and the process topology. Virtual topologies allow application programmer to attach numerical names other than group names to processes, so that the domain decomposition or other application – relevant mapping of data to processes is intuitive and convenient. MPI provides such mapping of convenience of the application programmer; both Cartesian grids and general graph mappings are supported [Bar05, Skj94].

3.3.6 Bindings for Fortran and C

Defines the rules for MPI language bindings in general and defines for Fortran and C in particular [MPI03].

3.3.7 Environmental management and inquiry

Environmental management and inquiry discusses routines for getting and, where appropriate, setting various parameters that relate to the MPI implementation and the execution environment (such as error handling). It also

describes the procedures for entering and leaving the MPI execution environment [MPI03].

3.3.8 Profiling interface.

The objective of the MPI profiling interface is to ensure that it is relatively easy for authors of profiling (and other similar) tools to interface their codes to MPI implementations on different machines [MPI03].

To satisfy the requirements of the MPI profiling interface, an implementation of the MPI functions must [Sni95]:

1. Provide a mechanism through which all of the MPI defined functions may be accessed with a name shift.
2. Ensure that those MPI functions which are not replaced may still be linked into an executable image without causing name clashes.
3. Document the implementation of different language bindings of the MPI interface if they are layered on top of each other, so that the profiler developer knows whether the profile interface must be implemented for each binding, or whether it needs to be implemented only for the lowest level routines.
4. Ensure that where the implementation of different language bindings is done through a layered approach, these wrapper functions are separable from the rest of the library. This is necessary to allow a separate profiling library to be correctly implemented, since the profiling library must contain these wrapper functions if it is to perform as expected. This requirement allows the person who builds the profiling library to extract

these functions from the original MPI library and add them into the profiling library without bringing along any other unnecessary code.

3.4 Java Binding of MPI

Although MPI has Fortran, C and C++ language bindings, a Java binding for MPI are designed for the Java implementation of MPI. Note that the Java language is different from Fortran, C or C++ in many ways. For example, in Java, The programmers program with classes and objects, and they use objects by reference, not by pointer. C is not an object-oriented language. Even C++ is not a pure object-oriented programming language, due to the requirement of being compatible with C. Therefore, they have to design a Java binding for MPI. In the Java binding, MPI functions are mapped to classes, methods and objects [Pen99].

One of the Java Binding for MPI is the Java Message Passing Interface (JMPI). It is implemented according to the Java binding for MPI.

Although no formal Java bindings exist yet for the MPI standard, the bindings as proposed by the Northeast Parallel Architectures Center (NPAC) was used in JMPI. First of all, the proposed Java bindings are derived from and closely follow the official C++ bindings developed by the MPI Forum. Secondly, it was important to maintain compatibility between the JMPI implementation of MPI and other implementations based on the proposed bindings. As a result, programmers don't need to modify their code for it to work with different Java implementations of MPI [Mor00].

3.5 Message Passing in JMPI

JMPI is an implementation of the Message Passing Interface for distributed memory multi-processing in Java. JMPI is completely written in Java and runs on any host that the JVM is supported on [Kor01].

JMPI implements Message Passing with Java's RMI. RMI is a native construct in Java that allows a method running on the local host to invoke a method on a remote host. One benefit of RMI is that a call to a remote method has the same semantics as a call to a local method. As a result, Serializable objects can be passed as parameters and returned as results of remote method invocations. Object Serialization is the process of encoding a Java Object into a stream of bytes for transmission across the network. On the remote host, the object is reconstructed by deserializing the object from the byte array stream. If the object contains references to other objects, the complete graph is serialized [Kor01].

For example, assume that Host B in Figure (3-1) wishes to send a message to Host C. At the time the distributed machine is formed, Host C registers a local object available for remote method invocation with a Java registry. The purpose of the registry is to function as a naming service. The registry allows a local method, such as the one running on Host B, to turn the object's Uniform Resource Locator (URL) into a reference to the remote method. Once the reference is bound, the process on Host B invokes the remote method with the Message as a single parameter. The method running on Host C inserts the message into a local FIFO queue, and notifies all blocked threads that a new message has arrived. The remote method returns a Boolean value of true to indicate the successful transmission of a message, otherwise false is returned [Mor00].

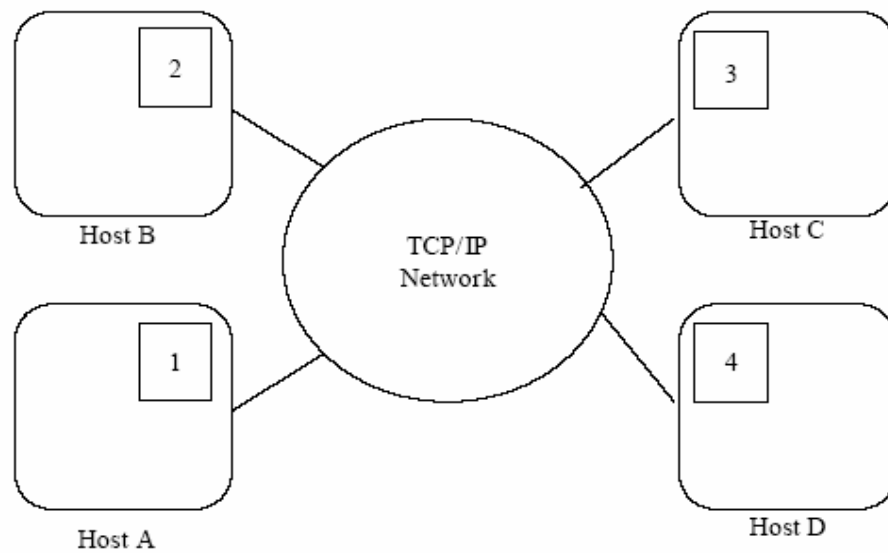


Figure (3-1): Four process Distributed Parallel Application

3.6 JMPI Architecture

JMPI has three distinct layers as shown in Figure (3-2): the MPI API, the Communications Layer and the JVM. The MPI API, which is based on the proposed set of Java bindings by the NPAC at Syracuse University, provides the core set of MPI functions that MPI applications are written to. The Communications Layer contains a core set of communication primitives used to implement the MPI API. Finally, the JVM that compiles and executes Java Bytecode [Kor01].

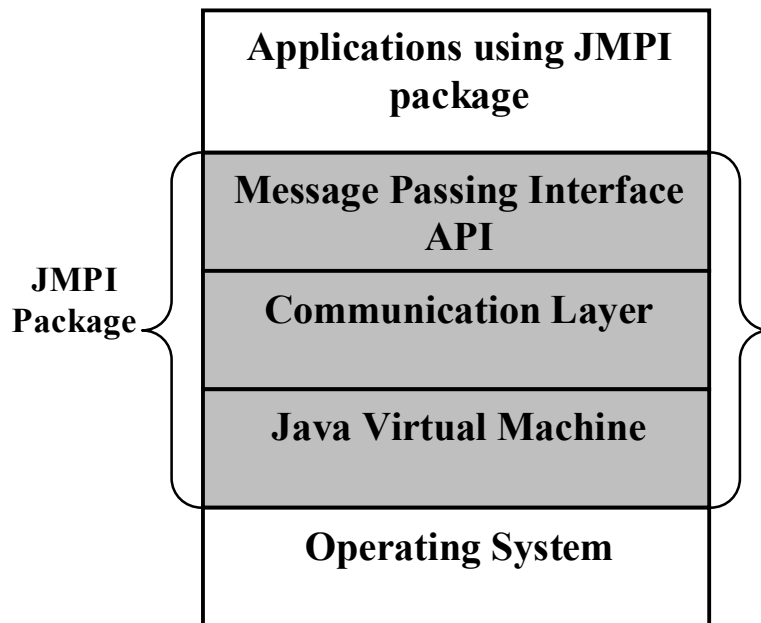


Figure (3-2): JMPI Architecture

3.6.1 MPI Application Programming Interface

MPI API is divided into six main classes: MPI, Group, Comm, Datatype, Status and Request. They are all placed in a package called mpi. These classes include [Mor00]:

- MPI: A class that includes constants in MPI and functions for initialization and finalization. The constants, variables, and methods in the MPI class are all declared as class constants, class variables, and class methods, respectively.
- Group: The class of the group objects. This class includes all group-related methods, which are used to create groups.
- Comm: The class of the communicators that includes all point-to-point communication and attribute-caching methods. All kinds of communicators are subclasses of the Comm class. The Comm has two

subclasses: the IntraComm class and the InterComm class. IntraComm class inherits the Comm class and includes all functions of point-to-point intra-communication. The functions of collective communication are added to the IntraComm class. This class is the superclass of various intra-communicators, while the InterComm class inherits the Comm class and includes all functions of point-to-point inter-communication. Furthermore, IntraComm class is split into two subclasses: CartComm and GraphComm. CartComm inherits from the IntraComm class. A CartComm object is an intra-communicator with the cartesian topological attributes. The CartComm class includes all the methods used for the cartesian topology management, while GraphComm class inherits from the IntraComm class. A GraphComm object is an intra-communicator with the graph topological attributes. The GraphComm class includes all the methods used for the graph topology management.

- Datatype: This class includes several methods for constructing derived data types like Contiguous, Vector, Indexed and Struct.
- Status: The class of the return-status objects in communications. A Status object contains the information about the source, tag, count, and data type of the message in a communication.
- Request: The class of the request objects. The request objects are used in non-blocking or permanent communications.

Figure (3-3) shows the organization of the MPI API [Mor00].

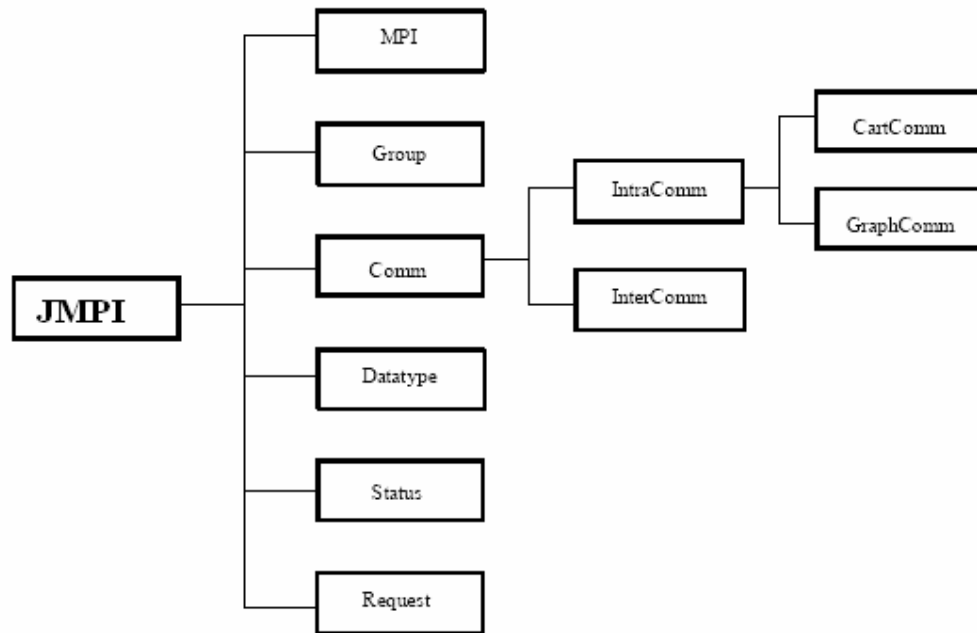


Figure (3-3): MPI API Class Organization

3.6.2 Communications Layer

The Communications Layer has three primary responsibilities: virtual machine initialization, routing messages between processes, and providing a core set of communications primitives to the MPI API. The Communications Layer is multi-threaded and runs in a separate thread than the MPI process. This allows for the implementation of non-blocking and synchronous communication primitives in the MPI API. Messages are transmitted directly to their destination via RMI [Kor01].

The Communications Layer performs three tasks during virtual machine initialization: start an instance of the Java registry, register an instance of the Communication Layer's server skeleton and client stub with the registry, and perform a barrier with all other processes which will participate in the virtual machine. The Registry serves as an object manager and naming service. Each

process within the virtual machine registers an instance of the Communications Layer with the local registry. The registered instance of the Communication Layer is addressable with a URL of the form [Mor00, Kor01]:

```
rmi://hostname.domainname.com:portno/Commx
```

Where portno represents the port number that the registry accepts connections on, and x represents the rank of the local process, which ranges from 0 to n-1, where n is the total number of processes in the virtual machine. One of the command line parameters passed to each process during initialization is the URL of the Rank 0 Communications Layer. After the Communications Layer has started its local registry and registered its instance of the local Communications Layer, the last step during initialization is to perform a barrier with all other processes. There are two advantages of barrier operation: ensure that all processes have initialized properly, and receive a table containing the URLs of all the remote processes Communications Layers within the virtual machine. The barrier is formed when all processes have invoked the barrier method on the rank 0 process. For processes 1 to n-1, this means a remote method invocation on the rank 0 process. Before the remote method can be invoked, the process must bind a reference to the remote object through its uniform resource locator. Once the reference is bound, the remote method is invoked in the same manner as a local method would be. The barrier on the rank 0 process uses Java's notify and wait methods to implement the barrier[Mor00].

Messages between processes are passed as a parameter to a remote method invocation on the destination process. The Message is a serializable object that consists of fields to represent the source rank, the destination rank, and message tag. In addition, the Message incorporates a Datatype object to indicate the type of data being sent in the message, as well as the actual data. The remote method

inserts the message object into the local processes FIFO Message Queue, and then notifies all locally blocked processes that a new message has arrived through Java's synchronization routines. For synchronous mode sends, where completion of the call indicates the receiver has progressed to the matching receive, the Communication Layer blocks and waits for notification from the local process that the matching receives has started. Java's wait and notify methods are used to implement synchronous mode sends. For all other communication modes, the remote method returns after the message has been inserted into the local message queue [Kor01].

The local process receives messages by invoking a local method in the Communications Layer. This method checks the incoming message queue for a message that matches the source rank and message tag passed as parameters. If a matching message is found, the receive call returns with this message. Otherwise, the call blocks and waits for a new message to come in. The message queue is implemented with a Java Vector Class for two reasons: synchronized access allows more than one thread to insert or remove messages at a time, and Vectors perform significantly faster than a hand-coded linked list implementation. In addition, Vectors also support out-of-order removal of messages, which is required to implement the MPI receive calls [Mor00, Kor01].

The Communications Layer provides communication primitives from which all other MPI bindings are implemented. The blocking point-to-point primitives have been discussed in the previous paragraphs. The non-blocking versions of the point-to-point primitives are implemented by creating a separate thread of execution, which simply calls the respective blocking version of the communication primitive. Collective communication primitives, such as a

broadcast to all processes are built on top of the point-to-point primitives. A separate message queue is used to buffer incoming collective operations to satisfy the MPI specification document. Two probe functions allow the message queue to be searched for matching messages without retrieving them. Finally, a barrier primitive allows all processes within an intracommunicator to synchronize their execution [Mor00, Kor01].

3.6.3 Java Virtual Machine (JVM)

Java source code is compiled to produce object code, known as *bytecode*. So far, this is just like any other language. The critical difference is that bytecode is not the binary code of any existing computer. It is an architecture-neutral machine code that can quickly be interpreted to run on any specific computer. A Java program is executed by running another program called the JVM. The JVM is typically invoked by running the program called *Java*. The JVM reads the bytecode program and interprets or translates it into the native instruction set.

Running bytecode on a JVM is a highly significant feature and this makes the Java software is “Write Once, Run Everywhere”. A Java executable is a binary file that runs on every processor. A Java program is compiled on any computer and runs on any computer.

The JVM – a fancy name for interpreter – needs to be implemented once for each computer system; then all bytecode will run on that system. There are several alternative JVMs available for the personal computers, and they differ in speed, cost and quality [Lin99].

Chapter Four
Implementation of JMPI
Middleware System

Chapter Four

Implementation of JMPI Middleware System

4.1 Introduction

This chapter concerned with the implementation of the JMPI Middleware system, which is a system for installing and running JMPI package, and run applications using this package. Three applications are taken as examples to be run on this system. They can be run in two ways: Serial and Parallel.

JMPI is completely written in Java, and it uses RMI technique. It requires a Java Development Kit (JDK), which is a Sun Microsystems product aimed at Java developers. Since the introduction of Java, it has been by far the most widely used Java Software Development Kit. The primary components of the JDK are a selection of programming tools, including: loader, compiler, debugger, etc. JDK Version 1.3 is preferred over Version 1.2 because of the performance enhancements made to RMI and Object Serialization.

JMPI Middleware System implemented on a LAN consists from five computers. Each computer is an IBM Pentium 4 has CPU 3.0 GHz Hyperthreading, 512 cache and RAM is 512 MB. The LAN connected by a switch which sends a signal to specific computers. The topology of LAN is star topology.

4.2 JMPI Middleware System

JMPI Middleware system is a software which performs the installation and simplifies JMPI package usage by the application programmer. The process of downloading and installing JMPI software and configuring the Virtual Machine are described in more details in appendix A. JMPI Middleware system must check that JMPI package is installed on the client computer or not. If JMPI is not installed, then JMPI Middleware system adds a key to the Windows registry and create a String value in it called JMPI_Reg. The proposed system knows that JMPI package is not installed if the key is not available in the registry. During the Installation, JMPI Middleware System does the following operations:

- Setup RSH service.
 - Copy JMPI package from Compact Disk to the Hard Disk.
 - Change value of Path system variable.
 - Create CLASSPATH system variable and put path of mpi.jar file.
- Create Machine File.

Figure (4-1) depicts the architecture of JMPI Middleware system.

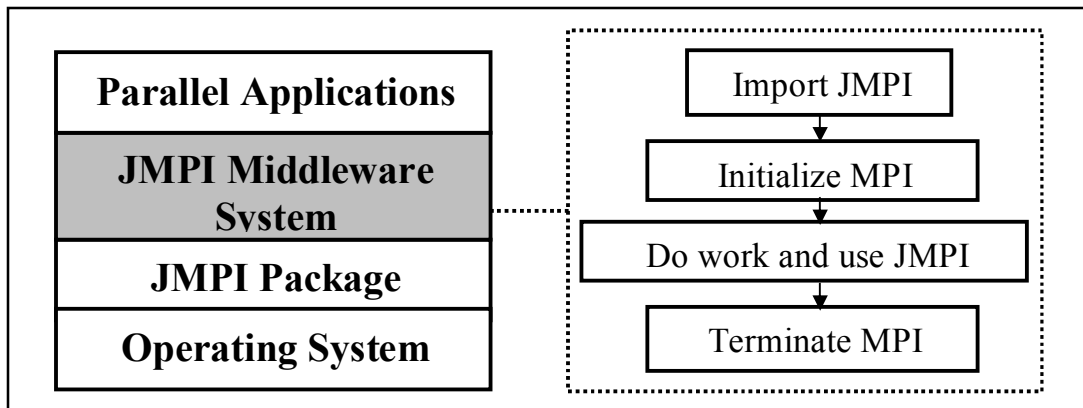


Figure (4-1): JMPI Middleware System

To execute any application of JMPI Middleware system, the package should be imported and there are initialization and finalization for MPI

execution environment. The initialization is in the form of `MPI.Init(args)` call. This function must be called in every MPI program. It must be called before any other MPI functions, and must be called only once in an MPI program. It gives an indication to the OS that "this is an MPI program" and allows the OS to do any necessary initialization. The finalization is in the form of `MPI.Finalize()` call, which indicates OS that "clean up" with respect to MPI can commence. This function should be the last MPI routine called in every MPI program and no other MPI routines may be called after it.

JMPI Middleware system can be executed on one computer (stand-alone) or several computers (LAN). The Proposed Middleware system runs multiple processes as separate threads within one JVM on the stand-alone client computer because communication layer in JMPI package is multithreaded. The Middleware system runs one process per computer when it is executed on a LAN for reasons of efficiency, because message passing systems generally associate only one process per processor. RMI technique is used to exchange messages between the processes.

JMPI Middleware partitions the user application program into a number of processes to make all processors busy and none of them remains idle, so many processes working toward one solution. The basic premise is that multiple parallel processes work concurrently towards a common target using "messages" as their means of communicating with each other. Figure (4-2) shows communication between processes in a LAN by using JMPI Middleware system.

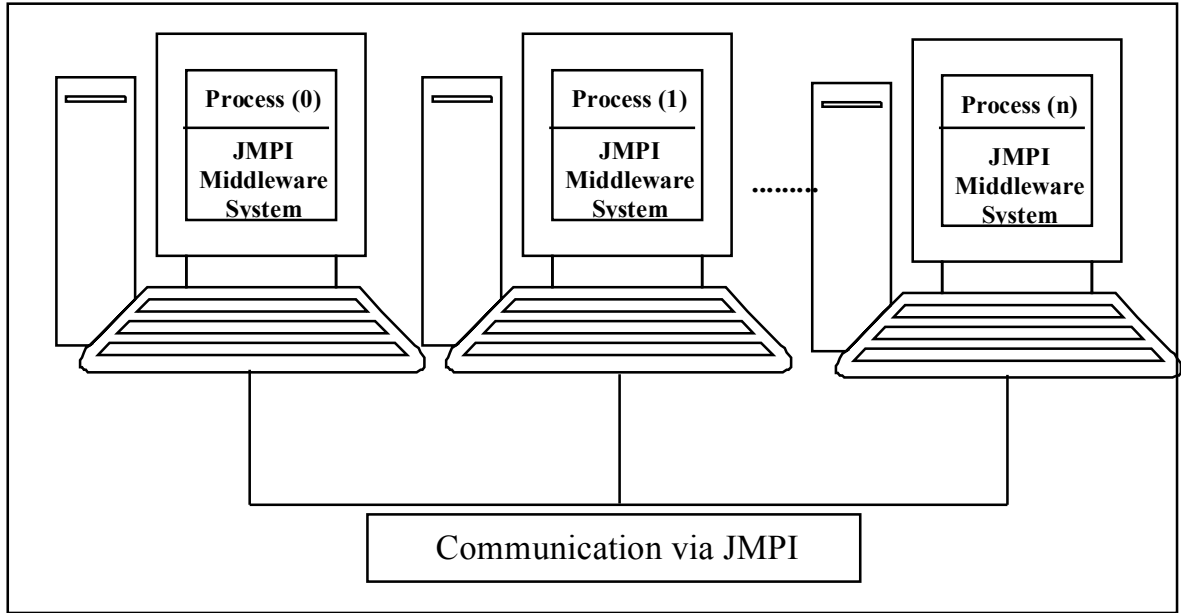


Figure (4-2): JMPI Process Communication in LAN

Figure (4-3) shows Flowchart of JMPI Middleware system model, it consists of two sub models (JMPI Middleware Installation) and (Use JMPI Middleware Package), which are appeared in more details in Figures (4-4 and 4-5).

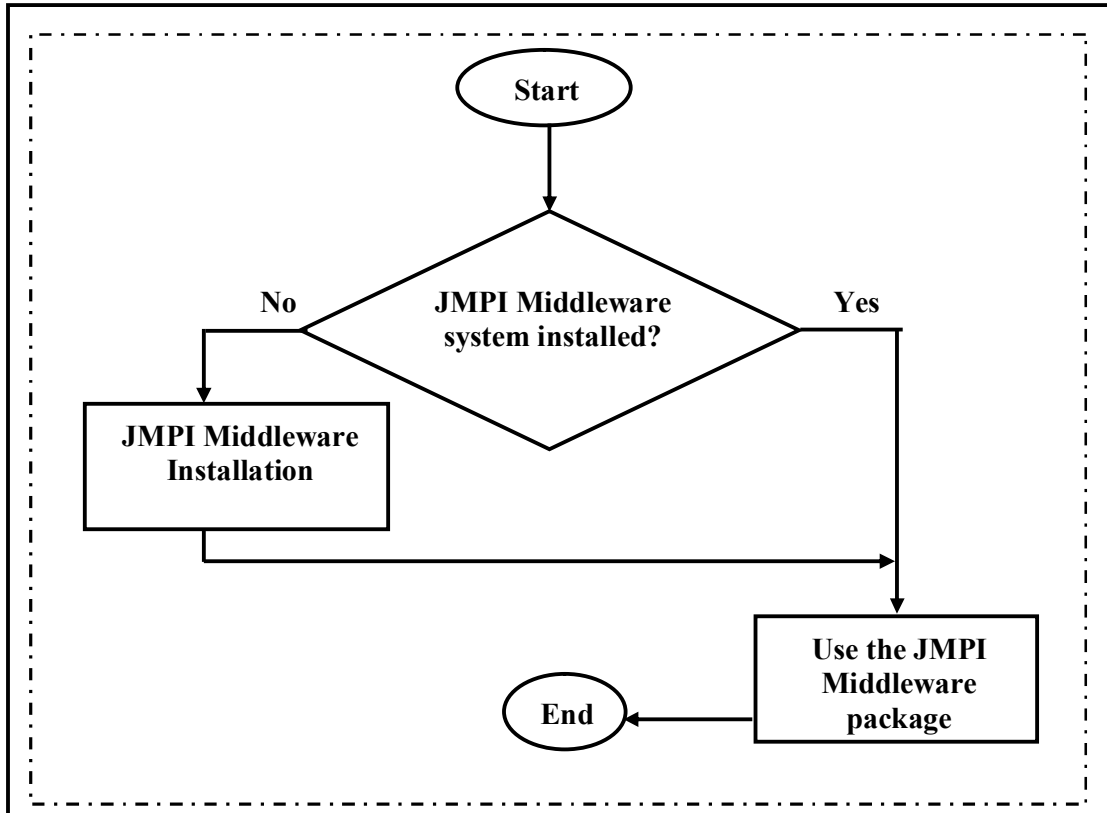


Figure (4-3): Flowchart of JMPI Middleware model

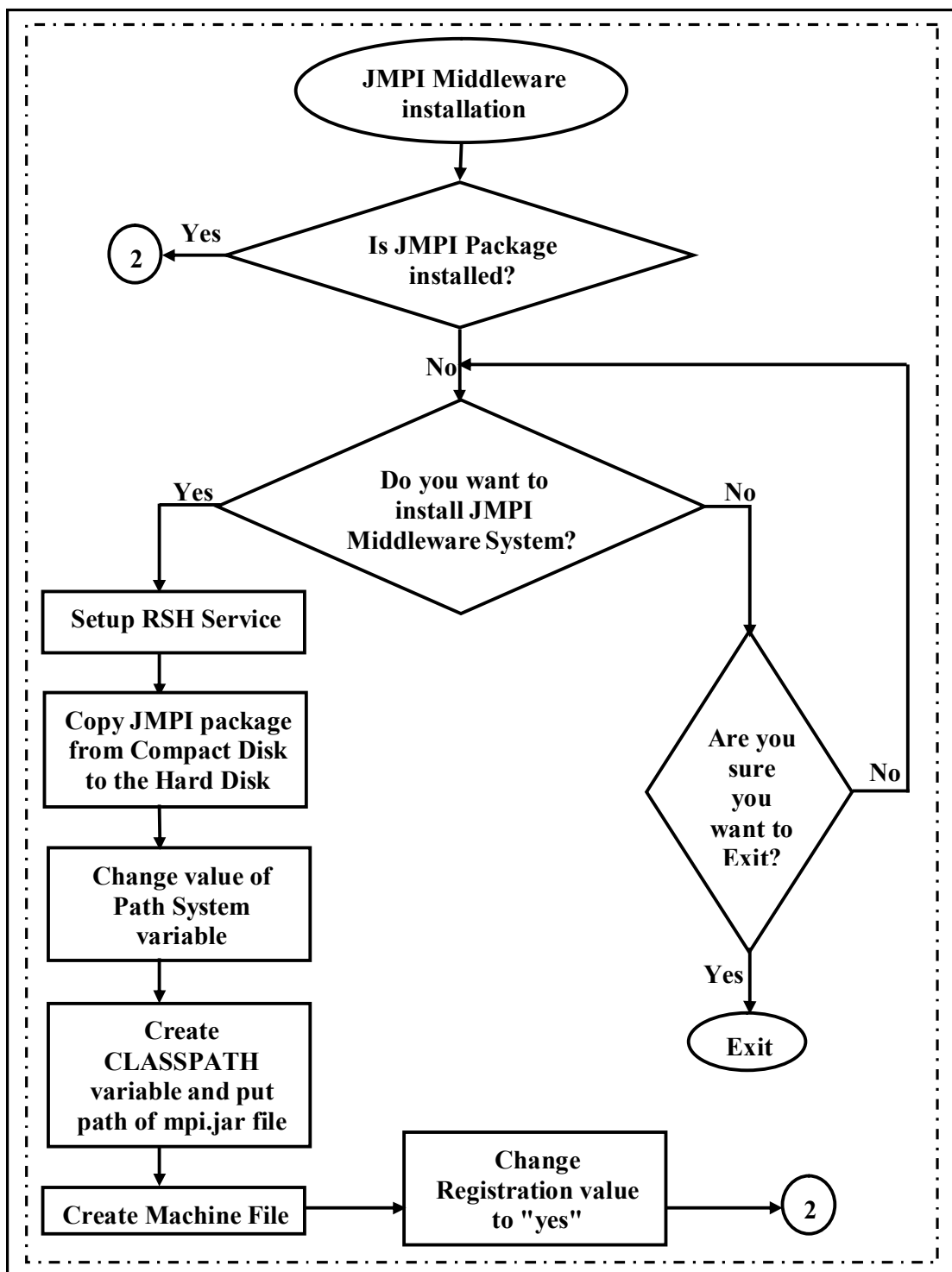


Figure (4-4): Flowchart of JMPI Middleware Installation

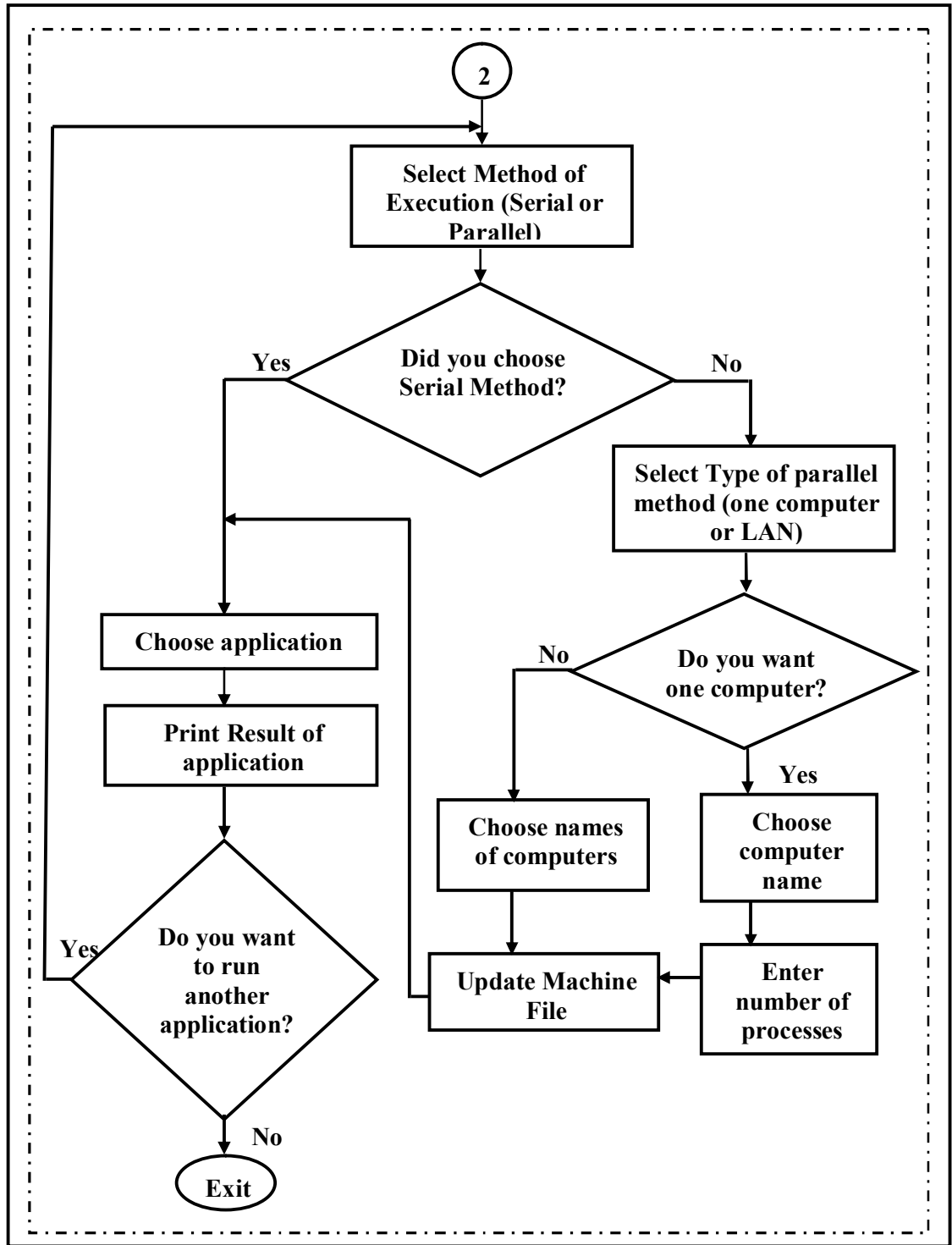


Figure (4-5): Flowchart of Running Applications Using JMPI Middleware System

4.3 JMPI Middleware System Applications

The applications which are implemented in this system have the facility to be executed in two ways serial and parallel. These applications are Range Addition, Matrix-Vector Multiplication and Gauss Elimination.

In parallel algorithm the following functions of JMPI package will be called:

- `MPI.COMM_WORLD.Size()`: Determines the number of processes in the group associated with a communicator. Generally used within the communicator `MPI_COMM_WORLD` to determine the number of processes being used by user's application.
- `MPI.COMM_WORLD.Rank()`: Determines the rank of the calling process within the communicator. Initially, each process will be assigned a unique integer rank between 0 and number of processors - 1 within the communicator `MPI_COMM_WORLD`. This rank is often referred to as a task ID. If a process becomes associated with other communicators, it will have a unique rank within each of these as well.
- `MPI.COMM_WORLD.Send(buf, offset, count, datatype, dest, tag)` : Basic blocking send operation. The function returns only after the application buffer in the sending task is free for reuse. This function may be implemented differently on different systems. The MPI standard permits the use of a system buffer but does not require it. Some implementations may actually use a synchronous send to implement the

basic blocking send. The arguments are:

buf: buffer of data.

offset: .starting address of data.

count: number of elements in the send buffer.

datatype: data type of the elements in the send buffer

dest: process rank to send the data.

tag: Arbitrary non-negative integer assigned by the programmer to uniquely identify a message.

- MPI.COMM_WORLD.Recv(buf, offset, count, datatype, source, tag):
Receives a message and blocks until the requested data is available in the application buffer in the receiving task.

The arguments are:

buf: buffer of data.

offset: .starting address of data.

count: number of elements in the send buffer.

datatype: data type of the elements in the send buffer

source: process rank indicates the originating process of the message.

tag: Arbitrary non-negative integer assigned by the programmer to uniquely identify a message.

- MPI.COMM_WORLD.Allreduce(sendbuf, sendoffset, recvbuf, recvoffset, count, datatype, op): is collective communication function. One member of the group collects data from the other members and perform an operation (such as add) on that data. This is equivalent to an MPI.COMM_WORLD.Reduce followed by MPI.COMM_WORLD.Bcast.
- MPI.COMM_WORLD.Barrier():Creates a barrier synchronization in a group. Each task, when reaching the MPI.COMM_WORLD.Barrier()

call, blocks until all tasks in the group reach the same MPI.COMM_WORLD.Barrier() call.

4.3.1 Range Addition Application

This Application is used to find the summation of a set of numbers within a specific range using two methods: serial and parallel. Algorithms (4.1) and (4.2) illustrate the serial method and the parallel method respectively.

Algorithm (4.1): Range Addition Using Serial Method

Goal: Find Sum of Numbers between Lower and Upper bounds.

Input: Lower and Upper Bounds.

Output: Result of Summation and program Time.

Store starting clock time.

Read Lower_bound and Upper_bound of range.

If (Lower Bound > Upper Bound)

Print "Range is incorrect".

Else Do I=Lower_bound,...Upper_bound

sum=sum+I

Print sum.

Store time after finishing calculations.

Compute and print time of execution.

Algorithm (4.2): Range Addition Using Parallel Method

Goal: Find Sum of Numbers between Lower and Upper bounds.

Input: Lower and Upper Bounds.

Output: Result of Summation and program Time.

Initialize MPI Environment using MPI.Init(args) function.

Compute number of processes (size) using MPI.COMM_WORLD.Size() function.

Find Rank of Process using MPI.COMM_WORLD.Rank() function.

Store starting clock time.

Read Lower_bound and Upper_bound of range.

If (Lower Bound > Upper Bound) Print Range is incorrect.

Else {

start_value = Upper_bound*rank/size+Lower_bound

end_value = Upper_bound*(Rank+Lower_bound)/size

Do I=start_value,...end_value

sum= sum+ I;

If (Rank!=0)

Call MPI.COMM_WORLD.Send(sum,0,1,MPI.INT,0,1) function

Else

Do J=1,..size {

Call MPI.COMM_WORLD.Recv(accum,0,1,MPI.INT,j,1) function.

sum= sum + accum.

}

Print Sum.

Store time after finishing calculations.

Compute and print time of execution.

4.3.2 Matrix-Vector Multiplication Application

This Application is used to find the product of a Matrix and a Vector using two methods: Serial and Parallel. Algorithms (4.3) and (4.4) illustrate the serial method and the parallel method respectively.

Algorithm (4.3): Matrix-Vector Multiplication Using Serial Method

Goal: Multiplying a Matrix with a Vector and store the result in a Vector Result.

Input: Matrix of size $N*N$ and Vector of size N .

Output: Vector of size N .

Store starting clock time.

Read Matrix and Vector.

Result = 0.

Do $I=0, \dots, N$.

{

sum1=0

Do $J=0, \dots, N$

{

Multiply Matrix $[I][J]$ with Vector $[J]$ and store the result in sum2

Add sum2 to sum1

}

Result $[I]=$ sum1

}

Print Matrix, Vector and Result.

Store time after finishing calculations.

Compute and print time of execution.

Algorithm (4.4): Matrix-Vector Multiplication Using Parallel Method

Goal: Multiplying a Matrix with a Vector and store the result in a Vector Result.

Input: Matrix of size $N*N$ and Vector of size N .

Output: Vector of size N .

Initialize MPI Environment using `MPI.Init(args)` function.

Compute number of processes using `MPI.COMM_WORLD.Size()` function.

Find Rank of Process using `MPI.COMM_WORLD.Rank()`

Store starting clock time

Read Matrix and Vector.

Result = 0.

Do $I=0, \dots, N$.

Do $J=0, \dots, N$

$Result[I]=Result[I]+Matrix[I][J]*Vector[J]$.

 Call `MPI.COMM_WORLD.Allreduce(Result,0,Matrix,0,N,MPI.DOUBLE, MPI.SUM)` function.

If (Rank=0) Print values of Matrix, Vector and Result.

 Store time after finishing calculations.

 Compute and print time of execution.

 Finalize MPI Environment using `MPI.Finalize()`.

4.3.3 Gauss Elimination Application

This Application is used to find the values of a Vector X from the equation $(A*X=B)$ using two methods: serial and parallel. Algorithm (4.5) and (4.8) illustrate the serial method and the parallel method respectively.

Algorithm (4.5): Gauss Elimination Using Serial Method

Goal: Find the values of Vector X in equation $AX=B$.

Input: N, Matrix A of size $N*N$, Vector B of size N.

Output: Vector X of size N.

Store starting clock time.

Read A and B.

X=0.

// k = current row

for (k=0; k<n; k++)

{

// in division step

for(j=k+1; j<n; j++)

{

if($A[k*n+k] \neq 0$)

$A[k*n+j] = A[k*n+j] / A[k*n+k]$

else

$A[k*n+j] = 0$

}

// calculates new value

if ($A[k*n+k] \neq 0$)

```

// for equation solution
X[k] = b[k] / A [k*n+k]
else
X[k] = 0.0
// sets Upper Triangular Matrix diagonal value
A[k*n+k] = 1.0
// Guassian elimination occurs in all remaining rows
for( i=k+1; i<n; i++ ) {
for( j=k+1; j<n; j++ )
A[i*n+j] -= A[i*n+k] * A[k*n+j]
b[i] -= A[i*n+k] * y[k]
A[i*n+k] = 0.0
}
}
Print A,B and X.
Store time after finishing calculations.
Compute and print time of execution.

```

Algorithm (4.6): Matrix Distribution

Goal: Distribute Matrix among Processes.

Input: Matrix M of size x by y, Matrix M of size x by y, x and y.

Output: None

```
if( rank == 0 )
{
    for( p=size-1; p>=0; p-- ) {
        for( i=p; i<y; i=i+size )
            for( j=0; j<x; j++ ) {
                LM[ (i/size)*x+j ] = M[ i*x+j ] }
        if ( p != 0 )
            MPI.COMM_WORLD.Send(LM,0,(y/size*x),MPI.DOUBLE,p,10) }
    else MPI.COMM_WORLD.Recv(LM,0,(y/size)*x,MPI.DOUBLE, 0,10)
}
```

Algorithm (4.7): Matrix Gathering

Goal: Gather Matrix from Processes.

Input: Matrix LM of size x by y, Matrix M of size x by y, x and y.

Output: None

```
if( rank == 0 ) {
    for( p=0; p<size; p++ ) {
        if( p != 0 )
            MPI.COMM_WORLD.Recv(LM,0,(y/size*x),MPI.DOUBLE,p,10)
        for( i=p; i<y; i+=size )
            for( j=0; j<x; j++ )
                M[i*x+j]=LM[(i/size)x+j] } }
    else MPI.COMM_WORLD.Send(LM,0,(y/size)*x,MPI.DOUBLE, 0,10)
```


Algorithm (4.8): Gauss Elimination Method Using Parallel Method

Goal: Find the values of Vector X in equation $AX=B$.

Input: N, Matrix A of size $n*n$, Vector B of size n.

Output: Vector X of size n.

Store starting clock time.

Read A,B.

$X = 0$.

Initialize MPI Environment using `MPI.Init(args)` function.

Compute number of processes using `MPI.COMM_WORLD.Size()` function.

Find Rank of Process using `MPI.COMM_WORLD.Rank()` function

$pred = (size+(rank-1)) \% size$

$succ = (rank+1) \% size$

// Distribute Matrix

Distribute_matrix(A,LA,n,n).

Distribute_matrix(B,LB.1,n).

for(k=0; k<n; k++) {

if((k%size) == rank) {

$ksn = (k/size)*n$

// Perform division step

for(j=k+1; j<n; j++)

$LA[ksn+j] = LA[ksn+j] / LA[ksn+k]$

$Ly[k/size] = Lb[k/size] / LA[ksn+k]$

$LA[ksn+k] = 1.0$

for(j=0; j<n; j++)

$cur_row[j] = LA[(k/size)*n+j]$

$cur_row[n] = Ly[k/size]$

 // Send row to successor using `MPI.COMM_WORLD.Send` function

```

MPI.COMM_WORLD.Send(cur_row,0, n+1, MPI.DOUBLE , succ, 20 )
}
else
{
// Receive row from predecessor using MPI.COMM_WORLD.Recv function
MPI.COMM_WORLD.Recv( cur_row,0, n+1, MPI.DOUBLE, pred, 20 )
    if( succ != k%size ) {
// Forward row to successor using MPI.COMM_WORLD.Send function
MPI.COMM_WORLD.Send(cur_row,0, n+1, MPI.DOUBLE, succ, 20 )
    }
}
// compute starting point
start = (rank <= k%size) ? (int) k/size+1 : (int) k/size
// Perform elimination step
    for( i=start; i<np; i++ ) {
        in = i*n; ink = in+k;
    for( j=k+1; j<n; j++ ) {
        LA[ in+j ] -= LA[ ink ] * cur_row[j];
    }
    Lb[i] -= LA[ ink ] * cur_row[n];
    LA[ ink ] = 0.0;    }    }
    MPI.COMM_WORLD.Barrier();
Print A,B and X.
Store time after finishing calculations.
Compute and print time of execution

```

4.4 JMPI Middleware System Tests

JMPI Middleware system designed with two main frames to enable the user to install and run applications. These frames are: JMPI Middleware System Startup and JMPI Middleware System Applications.

When JMPI Middleware system executed, it will check JMPI_Reg key in Windows Registry. If the key is not found then Warning Message will appear as shown in Figure(4-6) to inform the user that, JMPI package is not installed and ask him to install it or not. If the user wants to start the installation, then he must choose **Ok** button. After that the installation begins with RSH service setup and continues as mentioned in the appendix A.

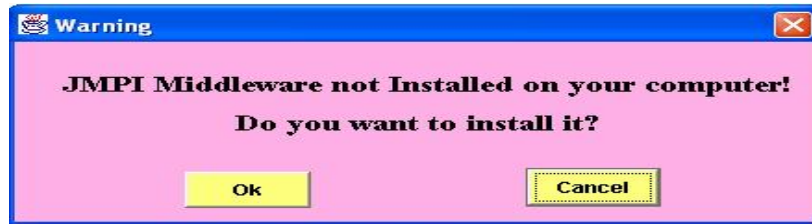


Figure (4-6): Warning Window

If the user chooses **cancel** then Exit window will be displayed as shown in Figure (4-7).



Figure (4-7): Exit Window

If the JMPI Middleware system installed properly, then the window in Figure (4-8) will be displayed and **Next** button will be enabled.



Figure (4-8): Installed JMPI Middleware System

When the user selects **Next** button, the JMPI Middleware System Applications window will be displayed as shown in Figure (4-9). The user can not choose any application unless he chooses an Execution Method.



Figure (4-9): JMPI Middleware System Applications Window

Execution Method contains two options: **Serial** and **Parallel**. **Parallel** involves two choices: **One computer** and **LAN**. If one of these options selected by the user, then **application** menu is enabled.

If the user selects **Serial**, then his working computer name will be displayed and **application** menu will be enabled as shown in Figure (4-10).



Figure (4-10): checked Serial option Window

If the user selects **Parallel** option, and then selects **One computer**, the window in Figure (4-11) will be appeared which contains computer name and the default value of number of processes which can be changed by the user.



Figure (4-11): One computer option window

If the user selects **Parallel** option, and then selects **LAN**, the window in Figure (4-12) will be appeared which contains list of Available Installed Computers and list of chosen computers which are selected by the user.



Figure (4-12): LAN option window

Then the user can choose any application. If he selects **Range Addition** the Figure (4-13) will be appeared and he must choose one of the Ranges that given in the list. After that the Result of addition and time of execution program will be printed.



Figure (4-13): Range Addition Application window

If the user selects **Matrix-Vector Multiplication**, Figure (4-14) will be appeared and he must choose one of the matrix sizes that have been given in the list. After that the values of resulted vector and time of execution program will be printed.

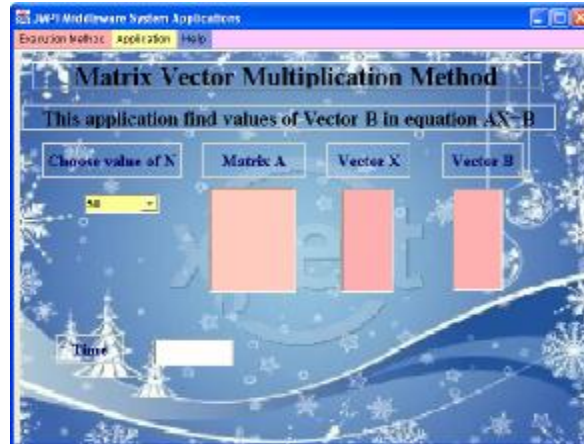


Figure (4-14): Matrix-Vector Multiplication Application window

If the user selects **Gauss Elimination**, the Figure (4-15) will be appeared and he must choose one of the matrix sizes that have been given in the list. After that the values of resulted vector and time of execution program will be printed.



Figure (4-15): Gauss Elimination Application window

Help menu is an option to display a message box contains short information about the JMPI Middleware System, as shown in Figure (4.16).



Figure (4-16): Help window

JMPI Middleware System tested on three applications with different Runs in two modes: Serial and Parallel on different sizes of data. In general the tested serial execution times are less than parallel execution times. Also the running time of using the LAN is less than using one computer. These differences in execution times are caused by the following reasons:

1. The speed of the used LAN (Fast Ethernet 100Mb/sec) is slow and also there is an overhead time resulted from using RMI technique. These two points increased the communication time which is added to the total execution time.
2. The execution times of one computer were increased in general when the number of processes increased. This was resulted from the time required by the OS to perform process and thread management.

The obtained results are in Tables (4-1), (4-2) and (4-3). Each value has been taken from the average of ten runs, and it is measured in millisecond (ms). When running the Gauss Elimination Application, the number of processes must be chosen such that the division of the data size over the number of processes is an integer number (i.e. data size mod number of processes = 0). Figures from (4-

17) to (4-22) show Time comparison in execution of Serial and Parallel (one computer) and Time comparison in execution of Serial and Parallel (LAN) of the three applications.

Table (4-1): Results of Range Addition Application

Application Name	Range	Time In										
		Serial (ms)	Parallel (ms)									
			One Computer (number of processes)					LAN (number of computers)				
			1	2	3	4	5	2	3	4	5	
Range Addition Application	1..1000	0.015	0.051	0.163	0.212	0.328	0.254	0.078	0.079	0.104	0.089	
	1..50000	0.029	0.054	0.176	0.226	0.304	0.397	0.081	0.081	0.104	0.107	
	1..100000	0.017	0.046	0.162	0.209	0.279	0.378	0.078	0.079	0.102	0.097	
	1..150000	0.025	0.073	0.152	0.211	0.266	0.367	0.083	0.084	0.092	0.104	
	1..250000	0.028	0.043	0.149	0.206	0.249	0.37	0.079	0.084	0.098	0.099	

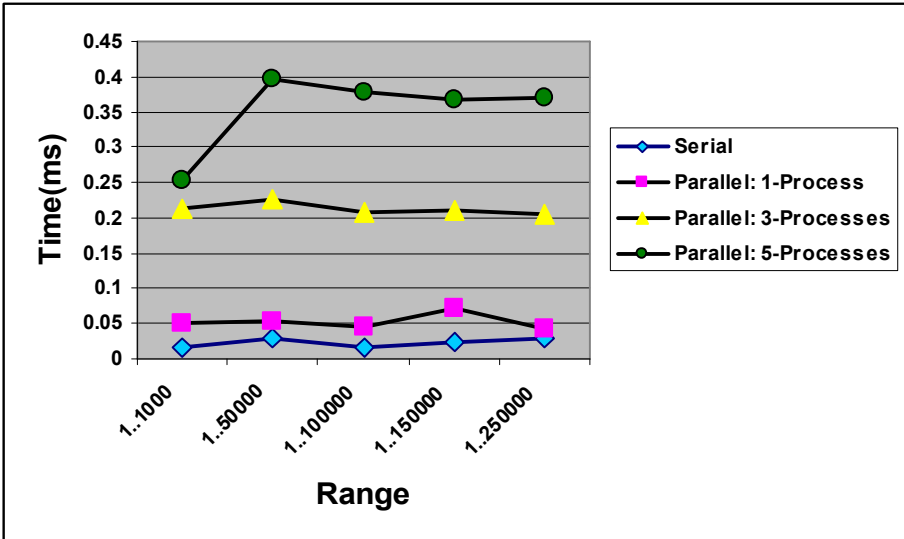


Figure (4-17): Time comparison in execution of Serial and Parallel (one computer) of Range Addition application

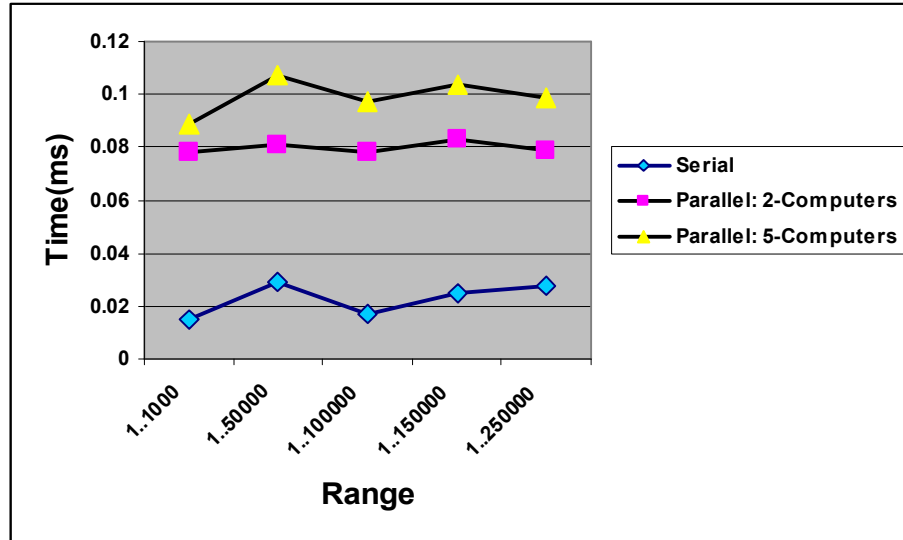


Figure (4-18): Time comparison in execution of Serial and Parallel (LAN) of Range Addition application

Table (4-2): Results of Matrix-Vector Multiplication Application

Application Name	Size (N)	Time In									
		Serial (ms)	Parallel (ms)								
			One Computer (number of processes)					LAN (number of computers)			
			1	2	3	4	5	2	3	4	5
Matrix-Vector Multiplication	50	0.09	0.155	0.205	0.155	0.158	0.157	0.154	0.253	0.157	0.15
	100	0.181	0.241	0.409	0.487	0.615	0.656	0.239	0.253	0.251	0.243
	150	0.182	0.246	0.415	0.526	0.769	1.03	0.253	0.245	0.249	0.403
	200	0.2	0.262	0.441	0.559	0.704	0.974	0.263	0.262	0.263	0.628
	250	0.184	0.25	0.410	0.576	0.788	0.884	0.251	0.251	0.249	0.936
	300	0.185	0.25	0.414	0.549	0.723	1.002	0.257	0.251	0.254	1.267

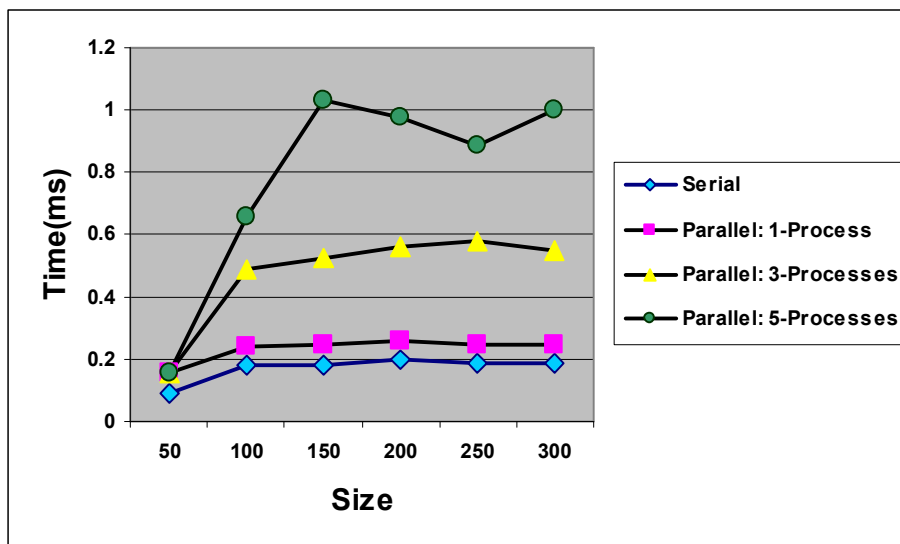


Figure (4-19): Time comparison in execution of Serial and Parallel (one computer) of Matrix-Vector Multiplication application

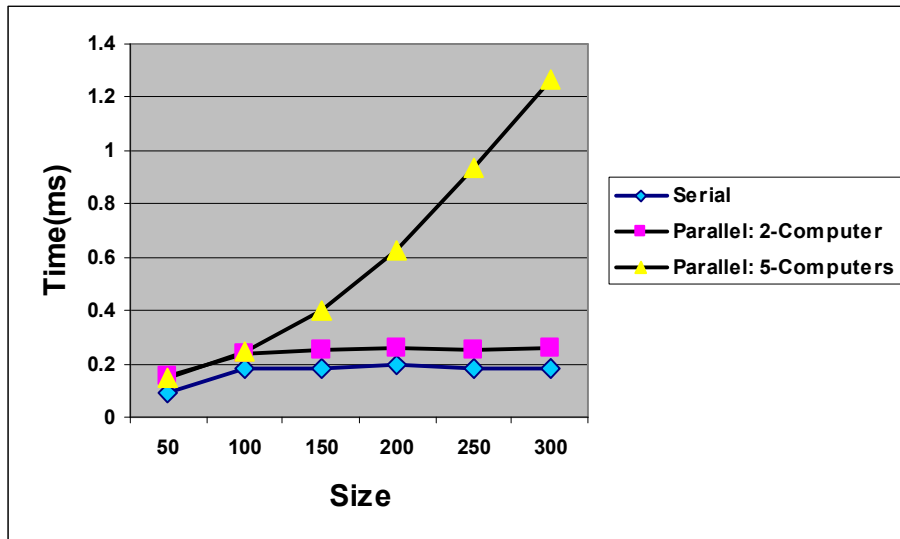


Figure (4-20): Time comparison in execution of Serial and Parallel (LAN) of Matrix-Vector Multiplication application

Table (4-3): Results of Gauss Elimination Application

Application Name	Size (N)	Time In								
		Serial (ms)	Parallel (ms)							
			One Computer (number of processes)					LAN (number of processes)		
			1	2	3	4	5	2	3	4
Gauss Elimination Application	50	0.108	0.905	1.156	-	-	2.062	0.969	-	-
	100	0.187	1.609	1.735	-	4.594	5.922	1.547	-	3.094
	150	0.359	1.219	1.25	2.485	-	4.266	2.109	4.204	-
	200	0.594	0.969	3.093	-	4.172	5.172	2.812	-	5.922
	250	0.907	3.733	3.781	-	-	6.093	3.468	-	-

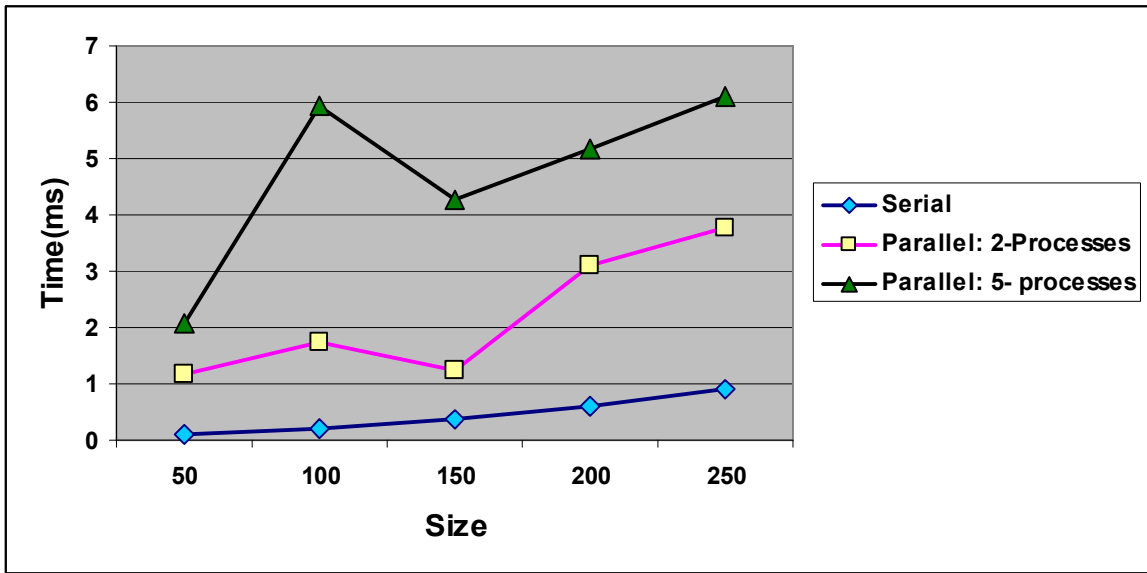


Figure (4-21): Time comparison in execution of Serial and Parallel (one computer) of Gauss Elimination application

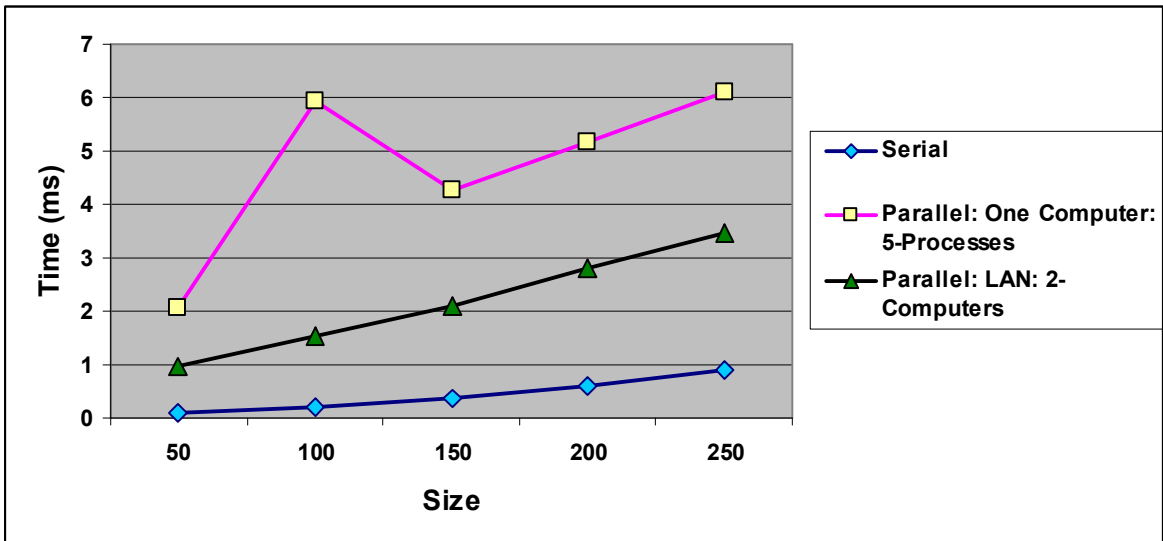


Figure (4-22): Time comparison in execution of Serial and Parallel of Gauss Elimination application



Chapter Five
Conclusions and
Future Work

Chapter Five

Conclusions and Future Work

5.1 Conclusions

From this research, several things were noticed and concluded. The most important ones are:

1. There is a difficulty in deciding which work can be accomplished concurrently and which in parallel. Also when the communication between these processes are necessary.
2. In the LAN each computer can run more than one process in parallel. For reasons of efficiency, only one process was run on each computer in this project.
3. The execution times that have gotten from running parallel applications were similar to the execution times of running serial applications because the calculations time of applications were simple compared to communications time.
4. The running times of using one computer are larger than LAN because of the time required by the OS to perform process and thread management.
5. The speed of LAN is slow which adds time to the communication time that makes execution time of using LAN became more than execution times of using one computer.

5.2 Future Work

There are number of suggestions appeared during the implementation of this work, some of these suggestions will be described below:

1. Modifying JMPI Middleware System to make it able to discover the installed and logged on computers in LAN automatically and make the system check the case of
2. Add a new facility to the JMPI Middleware System which gives the user the ability to add new applications which need distributed environment.
3. Implement another parallel computing system using another MPI package or another message passing model and make comparison with JMPI Middleware System.
4. RMI adds time overhead to each message passed. To reduce reliance of RMI for IPC, replace it with package applies Socket as a communication technique.
5. In JMPI Middleware System, the input data of the applications are fixed, add the facility which makes an online input for any data which are required by the applications.



References

References

- **[Aoy99]**
Y. Aoyama, J. Nakan, "RS/6000 SP: Practical MPI Programming", First Edition, 1999.
Website: <http://www.redbooks.ibm.com/redbooks/pdfs/sg245380.pdf>.
- **[Bac98]**
K. Baclawski, "Java RMI Tutorial", College of Computer Science, Northeastern University, 1998.
Website: http://www.ccs.neu.edu/home/kenb/com3337/rmi_tut.html.
- **[Bak04]**
M. A. Baker, H. Ong, A. Shafi, "A Status Report: Early Experiences with the implementation of a Message Passing System using Java NIO", 2004.
Website: http://dsg.port.ac.uk/%7Eshafia/res/papers/DSG_2.pdf.
- **[Baq06]**
Z. T. Baqer, "A Multiprocessing Computer System for the Finite Element Analysis", PhD thesis, Department of computer Engineering, Baghdad University, 2006.
- **[Bar05]**
B. Barney, "Message Passing Interface (MPI) ", 2005.
Website: <http://www.llnl.gov/computing/tutorials/mpi/>.
- **[Bar06]**
B. Barney, "Introduction to Parallel Computing", 2006.
Website: http://www.llnl.gov/computing/tutorials/parallel_comp/.

- **[CIO05]**
CIO magazine, "Middleware Demystified", 2005.
Website: <http://www.cio.com/archive/051500/middle.html>.
- **[Far05]**
Farlex, The Free Dictionary, 2005.
Website: <http://computingdictionary.thefreedictionary.com/Message+Passing+Interface>.
- **[Hew03]**
Hewlett-Packard Development Company, L.P., "HP MPI User's Guide", Eighth Edition, 2003.
Website: <http://docs.hp.com/en/B6060-96013/B6060-96013.pdf>.
- **[Jup05]**
Jupitermedia Corporation, Small business computing channel, "Interprocess Communication (IPC)", 2005.
Website: http://www.webopedia.com/TERM/I/interprocess_communication_IPC.html.
- **[Kor01]**
I. Koren, C. Mani Krishna, S. Morin, "JMPI: Implementing the Message Passing Standard in Java", Department of Electrical and Computer Engineering, University of Massachusetts, 2001.
- **[Lat97]**
LaTeX , "MPI: A Message-Passing Interface Standard", 1997.
Website: <http://www.cilea.it/~bottoni/mpi/nodi/mpi-report.htm>.
- **[Lin99]**
P. V. D. Linden, "Just Java", Sun Microsystems, Fourth edition, 1999.

- **[Mac95]**
 N. MacDonald, E. Minty, M. Antonioletti, J. Malard, T. Harding, S. Brown,
 "Writing Message-Passing Parallel Programs with MPI", Edinburgh Parallel
 Computing Centre, The University of Edinburgh, Epic Version, 1995.
 Website: http://www.epcc.ed.ac.uk/epic/mpi/notes/mpi-course_epic.book_1.html.
- **[Man02]**
 Mande, "Performance Prediction of Message Passing Communication in
 Distributed Memory Systems", Master Thesis, the Department of Electrical
 and Computer Engineering, University of Houston, 2002.
 Website: <http://www.bu.edu/caadlab/AparnaMande.pdf>.
- **[Mau95]**
 Maui High Performance Computing Center, "SP Parallel Programming
 Workshop Message Passing Overview ", 1995.
 Website: http://www.hku.hk/cc/sp2/workshop/html/message_passing/message_passing.html#message1.
- **[MPI93]**
 The MPI Forum, "MPI: A Message Passing Interface", 1993.
 Website: <http://www-fp.mcs.anl.gov/~lusk/papers/mpi-worksho/paper.html>.
- **[MPI94]**
 Message Passing Interface Forum, "MPI: A Message-Passing Interface
 Standard", 1994.
 Website: <http://www.cilea.it/~bottoni/mpi/nodi/mpi-report.htm>.
- **[MPI03]**
 Message Passing Interface Forum, " MPI: A Message-Passing Interface
 Standard", 2003.
 Website: <http://www.mpi-forum.org/docs/mpi1-report.pdf>.

- **[Mor99]**
 K. Morimoto, "Implementing Message Passing Communication with a Shared Memory Communication Mechanism ", Master Thesis, the University of Tokyo, 1999.
 Website:<http://www.sssc.org/ssspc/paper/master99-morimoto.pdf> March 1999.
- **[Mor00]**
 S. R. Morin, "JMPI: Implementing the Message Passing Interface in Java", Master Thesis, Department of Electrical and computer Engineering, University of Massachusetts, 2000.
 Website:<http://www.ecs.umass.edu/ece/realtime/publications/steve-thesis.pdf>.
- **[NCS01]**
 The National Center for Supercomputing Applications (NCSA), "Point-to-Point Communication", 2001.
 Website:<http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/CommonDoc/MessPass/>.
- **[NCS02]**
 The National Center for Supercomputing Applications (NCSA), "Message Passing Interface (MPI)", 2002.
 Website:<http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/CommonDoc/MessPass/MPI.html>.
- **[Pen99]**
 Y._Peng and W. Yang," Java Message Passing Package - A Design and Implementation of MPI in Java", Department of Computer and Information Science, National Chiao_Tung University, 1999.

Website:<http://citeseer.ist.psu.edu/cache/papers/cs/13901/http:zSzzSzwww.cis.nctu.edu.twzSz~wuuyangzSzpaperszSzjmpp.pdf/java-message-passing-package.pdf>.

- **[Skj94]**

A. Skjellum, E. Lusk and W. Gropp, "Early Application in Message-Passing Interface (MPI)", 1994.

Website:http://citeseer.ist.psu.edu/cache/papers/cs/5069/http:zSzzSzwww.cs.msstate.eduzSzstaffzSztonyzSzpublic_htmlzSzdocumentszSzApplicationszSzearly_apps_mpi.pdf/skjellum95early.pdf.

- **[Sni95]**

M. Snir, S. Otto, S. H. Lederman, D. Walker, J. Dongarra, "MPI: The Complete Reference", 1995.

Website: <http://www.netlib.org/utk/papers/mpi-book/book.html>.

- **[Squ04]**

J. M. Squyres, "A Component Architecture for the Message Passing Interface (MPI): The System Services Interface (SSI) of LAM\MPI" , PhD Thesis, University of Notre Dame, 2004.

Website:http://www.osl.iu.edu/publications/prints/2004/squyres04:_compon_archit_messag_passin_inter_mpi.pdf.

- **[Sta98]**

W. Stallings, "Operating Systems internals and Design principles", Prentice-hall.inc, 1998.

- **[Sun05]**

Sun Microsystems, "An Overview of RMI Applications", 2005.

Website:<http://java.sun.com/docs/books/tutorial/rmi/overview.html#generic>.

- **[Sun07]**
Sun Microsystems, 2007.
Website:<http://java.sun.com>.
- **[Tec01]**
TechTarget network of industry-specific IT encyclopedia, searchSMB.com
Definitions - powered by whatis.com, "Sockets", 2001.
Website:http://searchsmb.techtarget.com/sDefinition/0,,sid44_gci213021,00.html.
- **[Tec03]**
TechTarget network of industry-specific IT encyclopedia, SearchSMB.com
Definitions - powered by whatis.com, "semaphore", 2003.
Website:http://searchopensource.techtarget.com/sDefinition/0,,sid39_gci212959,00.html.
- **[Tec05]**
TechTarget network of industry-specific IT encyclopedia, searchSMB.com
Definitions - powered by whatis.com, "interprocess communication",
Copyright © 2004-2005.
Website:<http://searchsmb.techtarget.com>.
- **[Tec06]**
TechTarget network of industry-specific IT encyclopedia, "process", 2006.
Website:http://whatis.techtarget.com/definition/0,,sid9_gci212832,00.html.
- **[Wan01]**
C. L. Wang, R. K. K. Ma, and F. C. M. Lau, "M-JavaMPI: A Java-MPI
Binding with Process Migration Support", Department of Computer Science
and Information Systems, The University of Hong Kong, 2001.
Website:<http://www.cs.hku.hk/~clwang/papers/CCGrid2002-2nd-revision-0201.pdf>.

- **[Wik06]**

From Wikipedia, the free encyclopedia, 2006.

Website: http://en.wikipedia.org/wiki/Parallel_computing

Websites

- **[Web1]**

"Components of Client/Server Applications – Connectivity".

Website: <http://www.vanwijk.com/-%3D%20Bookz%20%3D-/Client%20Server%20Computing%20Second%20Edition/csc05.htm>.

- **[Web2]**

"Inter-process Communication and Coordination".

Website: www.nctu.edu.tw/~claven/course/aos/Chapter04.ppt.



Appendix A

Appendix A

A.1 JMPI Package Installation

This appendix describes the process of downloading and installing the JMPI software and configuring the Virtual Machine using JMPI Middleware system. The latest version of the JMPI source code, pre-compiled class files, and documentation are available on the web at:

<http://euler.ecs.umass.edu/jmpi>

JMPI is distributed as a compressed tar archive for Unix platforms, and as a WinZip file for Microsoft Windows platforms.

A.1.1 Downloading and installing RSH Service

Users of JMPI on the Microsoft Windows platform need to install and configure a **Remote SHell (RSH)** service on their machines. **RSH** allows the execution of non-interactive programs on another computer. In some systems, this command sometimes is called *remsh* or *rcmd*. It executes the command on the remote computer and returns the program's standard output and standard error output. The remote computer must run a **Remote shell daemon (Rshd)** to handle the incoming *rsh* commands.

The **RSH** service is available for download from the following website:

<http://www.bookcase.com/library/software/win3x.winsock.daemon.rsh.html>

The Middleware begins with the installation of **RSH** service by executing set up icon of service and the window shown in Figure (A-1) will be displayed.

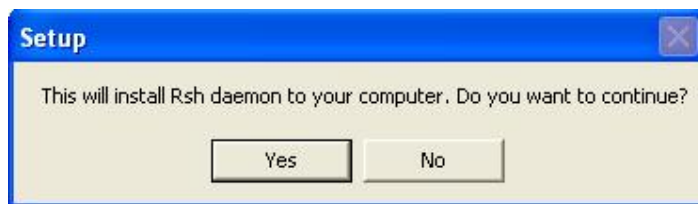


Figure (A-1): RSH Setup Window

Select **yes** to continue, then License agreement window will be displayed as shown in Figure (A-2).



Figure (A-2): License Agreement Window

By pressing on **I agree**, another window will be displayed to choose a folder to install the service in it, as shown in Figure (A-3).

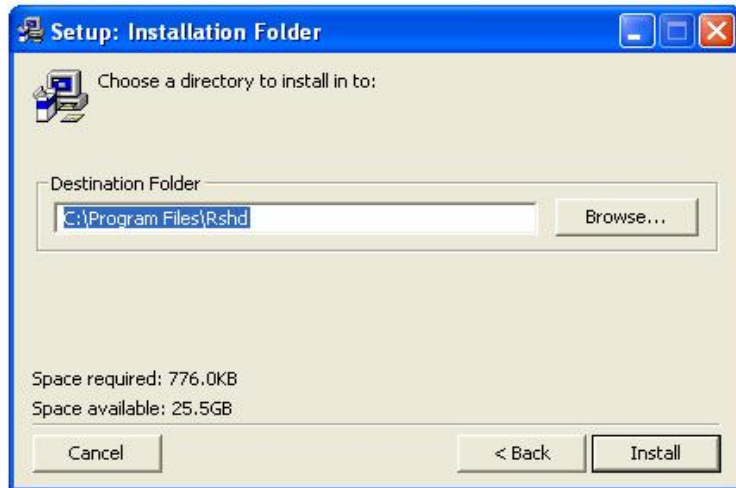


Figure (A-3): selection of Installation Folder Window

After choosing installation folder, press **Install** button, then the window in Figure (A-4) appears on the screen.

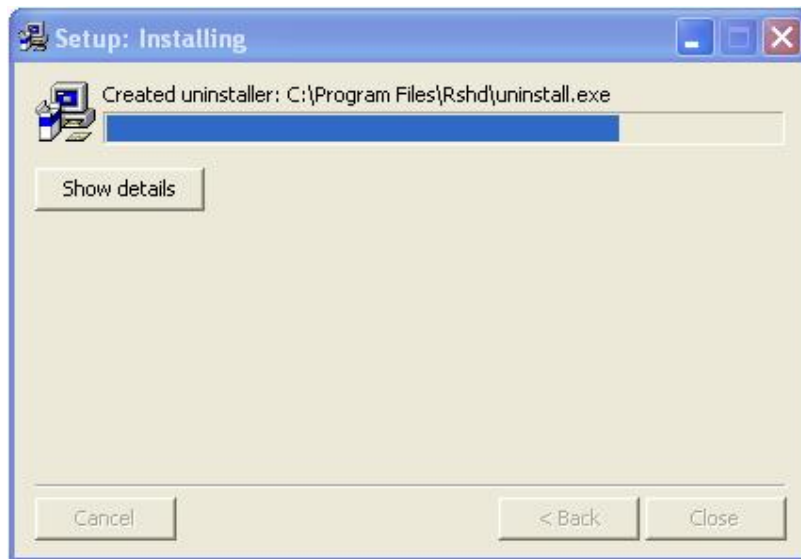


Figure (A-4): Installation Window

During the installation process, the window in Figure (A-5) will be displayed and the user must select **yes** option to complete installation.



Figure (A-5): Setup Window

By accomplishing the previous steps, the installation of RSH service will be done. To show the details of setup operation, the user can press **show details** button as shown in Figure (A-4). The window in Figure (A-6) will be displayed.

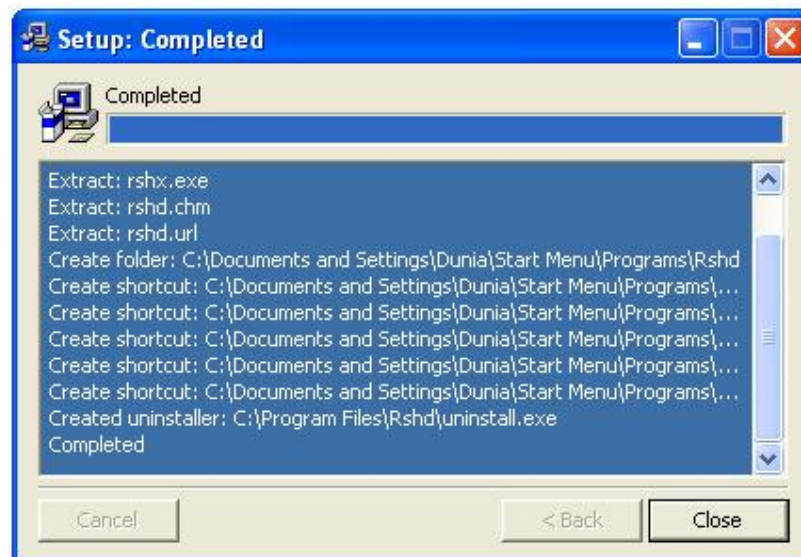


Figure (A-6): Setup Termination Window

When the user runs the Graphical User Interface (GUI) mode, the screen shown in Figure (A-7) will be displayed. This screen is used for basic service settings. Here the user can change the settings of service. For example: Installing, uninstalling, starting and stopping service. If he needs to set or change other properties (like startup type: change from automatic to manual start,...) of installed service, Windows standard dialog for service setting can be used. To open this dialog, open Control Panel, select Administrative Tools Icon,

choose Services Icon and choose the service name to change status of this service (start, stop and pause).



Figure (A-7): Basic Service Settings

The Second screen of the *Rshd* GUI is used for the daemon settings. It manages incoming connections and some other functions which are shown in Figure (A-8). The user can see which type of request he wants to accept, access logging and address lookup verification (it means that *Rshd* tries to find out IP address for remote computer name and translate it back, original and translated name have to be the same). *Rshd* gives the user the ability to kill some or all connections.

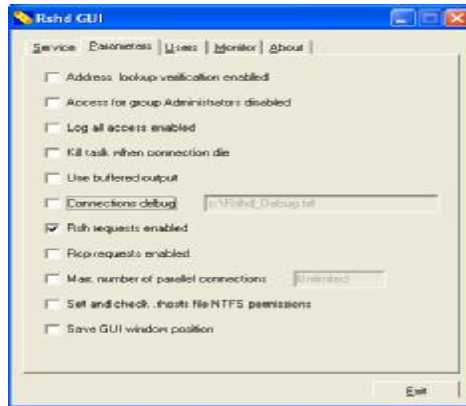


Figure (A-8): Daemon Settings

Figure (A-9) shows the table which holds the users names for remote connections and corresponding passwords (passwords are certainly ciphered). Every user on Windows has home directory. If this directory is on shared disk the user can optionally put the connection entry. These entries will be used for mounting shared disk. If the user name is not local but the user will use the account in the domain, he can put the whole name included the domain name.



Figure (A-9): User Names Table

Rshd monitoring can be enabled or disabled by implementing simple telnet server as shown in Figure (A-10). It provides the following:

- The connected user name and its address.
- The running process and its priority.

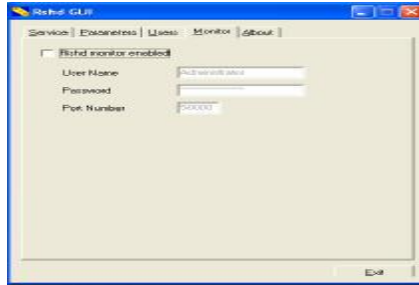


Figure (A-10): Daemon Monitoring

A.1.2 Configuring Virtual Machine

The set of hosts that MPI application will execute on, forms a Virtual Machine. Before an MPI process is compiled or executed on the Virtual Machine, two environmental variables need to be set or modified. In addition, each host in the Virtual Machine needs to have an entry in the *.rhosts* file of the host that launches the MPI application with *mpirun*. The *.rhosts* file is a text file in which each line is an entry. An entry consists of the local computer name, the local user name, and any comments about the entry. Each entry is separated by a tab or space, and comments begin with a pound sign (#).

The *.rhosts* file typically permits network access. The *.rhosts* file lists computer names and associated logon names that have access to remote computers. When the user runs *rsh* command remotely with a correctly configured *.rhosts* file, he do not need to provide logon and password information for the remote computer.

The JDK uses the CLASSPATH environment variable to locate Java classes. The filename of the *jmp.jar* archive needs to be included in the CLASSPATH environment variable. These environment variables can be changed from Windows by doing the following steps: For example, if the *jmp.jar* is located in `D:\jmp\mpi`, then to set the CLASSPATH environment

variable open **Control Panel**, select **System** Icon, go to **advanced** and choose **Environments Variables** as shown in Figure (A-11):

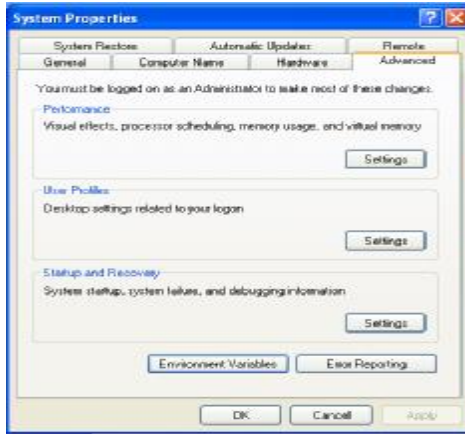


Figure (A-11): System Properties window

Select the **CLASSPATH** variable from the Environment Variables window as shown in Figure (A-12).

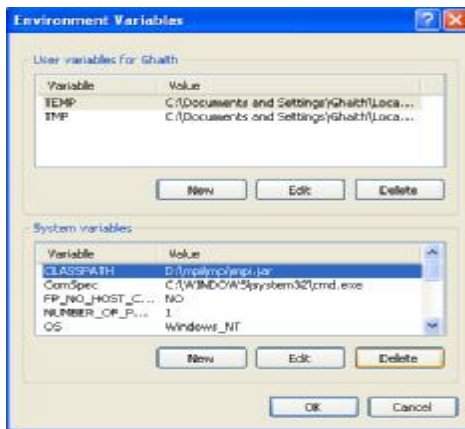


Figure (A-12): Environment Variables window

Press **Edit** button, and then print the path of Java classes in the variable value field as shown in Figure (A-13).



Figure (A-13): Edit System Variable Window of CLASSPATH variable

The PATH system variable needs to include the PATH where the JDK binaries are installed. For example, if the JDK is installed in D:\JBuilder7\jdk1.3.1, include D:\JBuilder7\jdk1.3.1\bin in PATH environment variable. To include this path, open **Control Panel**, choose **System** icon, go to **Advanced** tab, press **Environment Variables** button and select **PATH** system Variable, then press **Edit** button and print the path of JDK binaries in it as shown in Figure (A-14).

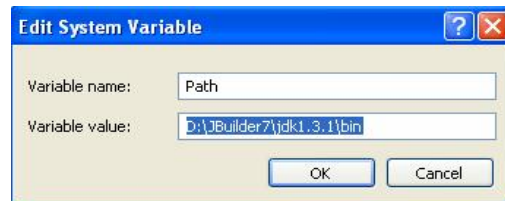


Figure (A-14): Edit System Variable Window of PATH variable

To perform the changing of CLASSPATH environment variable by using JMPI Middleware system, copy JMPI package to specific path (For example D:\) by creating batch file contains:

```
D:  
cd\  
@echo on  
%windir%\system32\xcopy G:\*.* D:\ /E  
pause
```

JMPI Middleware system changes CLASSPATH and PATH environment variables by creating batch files and using the following instructions:

```
set PATH = "path of bin directory of used JDK".  
set CLASSPATH = "path of mpi.jar file".
```

By using previous paths that used in the first method, the batch file will contain:

```
set PATH = D:\JBuilder7\jdk1.3.1\bin  
set CLASSPATH = D:\jmp\mpi\mpi.jar
```

The last step of installation is creating a file named *machinefile*, which contains an entry for each host which will participate in the virtual machine. Each host is placed on a single line, optionally followed by the number of processes to run on that host. If omitted, one process is started per machine. Additionally, the user may optionally specify the path and filename of the Java class on the remote machine.

After completing Installation process and changing the String value of JMPI_Reg to "yes", then JMPI package is installed successfully.

الخلاصة

واجهة عبور الرسالة تُزود بناءً تحتِي يُمكن المستخدمين لبناء بيئة حاسبات موزعة عالية الأداء باستخدام شبكة حاسوبية بأدنى جهد و يسهل إستعمال هذه البيئة بصورة بسيطة نسبياً باستخدام نموذج عبور الرسالة و هو يجهز واجهة تطبيق برمجية شائعة لبناء و تطوير التطبيقات المتوازية بغض النظر عن نوع النظام المتعدد المعالجات المستخدم.

يستخدم البحث المقترح لغة الجافا لتطبيق نظام إحتسابي موزع يسمى (Java Message Passing Interface Middleware System) يُستخدم لتنصيب رزمة Java Message Passing Interface (JMPI) وتنفيذ ثلاثة تطبيقات (Range Addition) و (Matrix - Vector Multiplication) و (Gauss Elimination method) بنمطين هما المتسلسل و المتوازي ثم حساب النتائج على النظام المُقترح.

نُفذ النظام على شبكة حاسوبية محلية مؤلفة من خمسة حاسبات و طُبِّقت العديد من التجارب لإختبار النظام و قد وجدت ان نتائج تنفيذ التطبيقات المتوازية مُقاربة لنتائج التطبيقات المُتسلسلة والسبب هو ان وقت الحسابات الخاص بالتطبيقات بسيط مُقارنةً بوقت الاتصالات.



جمهورية العراق
وزارة التعليم العالي و البحث العلمي
جامعة النهريين
كلية العلوم

بناء بيئة حسابية متوازنة باستخدام واجهة عبور الرسالة

رسالة مقدمة الى كلية العلوم, جامعة النهريين كجزء من متطلبات نيل شهادة
الماجستير في علوم الحاسوب

من قبل
دنيا حامد حميد

بكالوريوس
2004

المشرفون

د. سوسن كمال ثامر

د. لمياء حافظ خالد

ذو الحجة 1428

كانون الأول 2007