PARMI: A PUBLISH/SUBSCRIBE BASED ASYNCHRONOUS RMI FRAMEWORK

By

HEE JIN SON

Bachelor of Arts
Kyung Hee University
Seoul, Korea
1998

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2007

PARMI: A PUBLISH/SUBSCRIBE BASED

ASYNCHRONOUS RMI FRAMEWORK

Thesis Approved:

Xiaolin Li

Thesis Adviser

Nohpill Park

Venkatesh Sarangan

A. Gordon Emslie

Dean of the Graduate College

TABLE OF CONTENTS

Chapter Page

CHAPTER 1 Introduction.......... 1

1.1 Motivation.......... 1
1.2 Research Overview.......... 3
1.3 Contributions.......... 4
1.4 Outlines of the Thesis.......... 5

CHAPTER 2 Programming Models and Related Work.......... 6

2.1 Remote Method Invocation.......... 6
2.2 Asynchronous RMI.......... 12
2.3 Publish/Subscribe Communication.......... 15
2.4 Generics in the Java Programming Language.......... 16
2.5 Scientific Computing and Grid Computations.......... 17

CHAPTER 3 System Architecture.......... 33

3.1 Asynchronous RMI with a Future Object.......... 34
3.2 PARMI Framework Architecture.......... 42
3.3 Application of PARMI with Jacobi Iteration.......... 49

CHAPTER 4 Evaluation.......... 55

4.1 System Setup.......... 55
4.2 Results.......... 58
4.3 Conclusion.......... 66

CHAPTER 5 Conclusion and Future Work.......... 69

5.1 Conclusion.......... 69
5.2 Future Work.......... 70

REFERENCES.......... 72

LIST OF TABLES

Table Page

Table 2.1 General syntax for RMI 9

Table 2.2 Program of π calculation........ 20

Table 2.3 Sequential Jacobi iteration........ 26

Table 2.4 Functional inlining for enhanced performance........ 27

Table 2.5 Parallel Jacobi iteration........ 28

Table 3.1 RMI codes in the server side........ 36

Table 3.2 RMI codes in the client side 38

Table 3.3 Sync/async RMI over local calls. 40

Table 3.4 Sync/async RMI over remote calls. 40

Table 4.1 Shell script controlling SSH shells 60

Table 4.2 Comparisons of speedup using different environment. 68

LIST OF FIGURES

Figure Page

Figure 2.1 The RMI runtime architecture, courtesy: A. Wollrath 7

Figure 2.2 The stage of RMI call 12

Figure 2.3 Asynchronous invocation vs. publish/subscribe, courtesy: P.T. Eugster 14

Figure 2.4 π calculation 19

Figure 2.5 Computation of two- dimensional finite difference 21

Figure 2.6 Exchange the edges and boundaries between two processes 29

Figure 3.1 Operation of the asynchronous RMI with a future object 39

Figure 3.2 Sync/async RMI over local calls. 41

Figure 3.3 Sync/async RMI over remote calls. 42

Figure 3.4 PARMI organization 46

Figure 3.5 Flow chart for a publisher 47

Figure 3.6 Flow chart for a subscriber 48

Figure 3.7 The Jacobi iteration processes between a master and workers: part 1 50

Figure 3.8 The Jacobi iteration processes between a master and workers: part 2 51

Figure 3.9 Implementations of Jacobi application using PARMI framework 52

Figure 3.10 Time analysis for computation and communication 53

Figure 4.1 Execution cost for sync/async versions 60

Figure 4.2 The number of iteration for sync/async versions 61

Figure 4.3 Average communication cost for sync/async versions 62

Figure 4.4 Average computation cost for sync/async versions 63

Figure 4.5 The number of processors: Linux versus Windows 64

Figure 4.6 Execution cost for sync/async versions... 64

Figure 4.7 The number of iteration for sync/async versions 65

Figure 4.8 Average communication cost for sync/async versions.................................... 65

Figure 4.9 Average computation cost for sync/async versions... 66

CHAPTER 1

Introduction

1.1 Motivation

The widespread proliferation of the internet, as well as small organizational intranets, has provided seamless access to information that is distributed in remote locations across the network. There are many ways that application software components of different machines can communicate with one another over a network.

The first generation of the distributed communication mechanism uses the call interface of the network layer directly, such as *socket* mechanism. While flexible and sufficient for general communication, the use of sockets requires the client and server using this medium to engage in some application-level protocol to encode and decode messages for exchange. Design of such protocols is cumbersome and can be error-prone[40].

An already classic alternative to sockets is *Remote Procedure Call* (RPC). RPC allows a program running on one computer to cause a subroutine on another computer to be executed without explicitly coding[11, 22]. Arguments and return values are automatically packaged and sent between the local and remote procedures. However, while RPC is well suited for the procedural programming paradigm, it does not translate well into distributed object systems that have gained much popularity in recent years.

In order to match the semantics of object invocation, distributed object systems require *remote method invocation* (RMI). RMI is similar to RPC, but integrates the distributed object model into the Java language in a natural way[33]. While RMI has many similarities with RPC, it supports Java's security mechanism, multi-threaded nature, and object-oriented characters such as inheritance, polymorphism and dynamic binding. RMI has emerged as a powerful and easy to use language. It provides integrated and reliable communication between objects in a distributed system via remote method calls. It allows programmers to develop distributed Java programs with the same syntax and semantics used for non-distributed programs. For many people who are used to Java, RMI support powerful tools without understanding special new concept for the distributed object system.

However, although RMI is attractive to the dynamic distributed systems, it is not desirable in many applications due to its *synchronous nature*. The client is blocked while the call is processed by the server. The trend in distributed systems goes towards asynchronous and reactive systems that cannot wait indefinitely for a synchronous call to terminate. The synchronous nature of RMI leads to lower performance. Therefore, a number of projects have investigated fast implementations supporting asynchronous communication. Apparently the bottleneck caused by synchronous remote method invocation effects Java technologies and challenges to look for a suitable solution.

1.1.1 Problem Statement

The main shortcoming of RMI is that it causes the execution of the requesting object to be suspended until the computation is carried out in a remote address space. This delay is incurred regardless of whether the invoking objects require the return value or not. Therefore, the invoker always waits for a reply before continuing the remaining processes.

Another limitation of RMI is that it only supports point-to-point communication. Remote invocations can only be forwarded to a single destination. Although point-to-point communication is perfectly suited for expressing communication in client-server applications, many parallel applications are difficult to implement efficiently using this limited model and require broadcast communication.

1.2 Research Overview

The overall goal of this study is to design, implement and evaluate an asynchronous RMI system that is suitable for use in parallel and distributed system for grid computing. The detailed objectives are as follows.

- Explore the description of the RMI model and analyze the performance of an existing RMI implementation.
- Study the related programming models for designing asynchronous RMI structure.
- Introduce the structure and concepts for a new asynchronous way of communication in RMI accepting the publish/subscribe paradigm, which provides a conceptual simplicity and has the benefit to decouple objects in space and time.
- Provide the framework with concrete interfaces and classes.
- Evaluate the performance of an asynchronous way of RMI communication on the local/remote and homogeneous/heterogeneous environments.

1.3 Contributions

A publish/subscribe based asynchronous RMI framework (PARMI) has been implemented and experimentally evaluated in large-scale systems with up to a 51 machine cluster system. These experiments demonstrate that PARMI successfully improves the overall performance and stability. The contributions of PARMI are elaborated as follows:

- Optimizing performance with PARMI support, applications linked with RMI will be improved. In presence, middleware technologies are primarily based on some modification of the RPC/RMI mechanism [35]. In the first instance, RMI is one of supporting technologies on the well-known middleware Jini. Jini is an infrastructure that runs on top of Java and RMI to create a federation of devices and software components to implement services. In combination with object-based technologies, Jini allows the creation of large distributed programs without greater problems. In the second instance, as a standard component of the enterprise level, the basic concepts of Enterprise Java Bean (EJB) started from RMI. One of protocol uses in the internal of EJB is JRMP (as a transport protocol of RMI, EJB support IIOP) and the class design in EJB is similar to RMI except home interface.
- PARMI can be used as a communication element in diverse scientific applications, such as mathematics, physics, chemistry, large-scale image processing, and distributed stochastic simulations.

1.4 Outlines of the Thesis

The remainder of the thesis is organized as follows. Chapter 2 studies theories and related work in the areas of PARMI. Chapter 3 describes the architecture and overall design of framework including important features provided. Chapter 4 presents the evaluation of the framework and reports the performance and the nature of losses and overheads involved. Finally, Chapter 5 draws conclusions and also discusses the opportunities for improvement and future work.

CHAPTER 2

Programming Models and Related Work

With the increasing adoption of Java for parallel and distributed computing, RMI is one of the most popular communication paradigms in distributed computing. There are many considerable works for enhancing the high performance RMI implementations. This chapter presents the background and approaches for PARMI implementations and gives an overview of the proposed strategies.

Section 2.1 describes the features provided by the Java RMI. Section 2.2 reviews various research done on mechanisms and complete systems which are related to asynchronous RMI. Section 2.3 discusses Generics provided on Java languages. Section 2.4 introduces the highlighted communication paradigm in the diversified distributed computing, the publish/subscribe communication. Finally, Section 2.5 studies scientific and grid computing in use for the application model for PARMI.

2.1 Remote Method Invocation

In this section, an overall RMI mechanism is described. The RMI system allows an object running in one Java virtual machine (JVM) to invoke methods on an object running in another JVM. A user can utilize a remote reference in the same manner as a local reference. This feature is regarded as an object-oriented version of a Remote Procedure Call (RPC).

2.1.1 RMI System Architecture

Starting with an overview of the underlying RMI architecture, there are three layers that comprise the basic remote-object communication facilities in RMI: *the stub/skeleton layer, the remote reference layer*, and *the transport layer*[40]. As Figure 2.1 shows, each layer is independent of the next and can be replaced by an alternate implementation without affecting the other layers in the system.

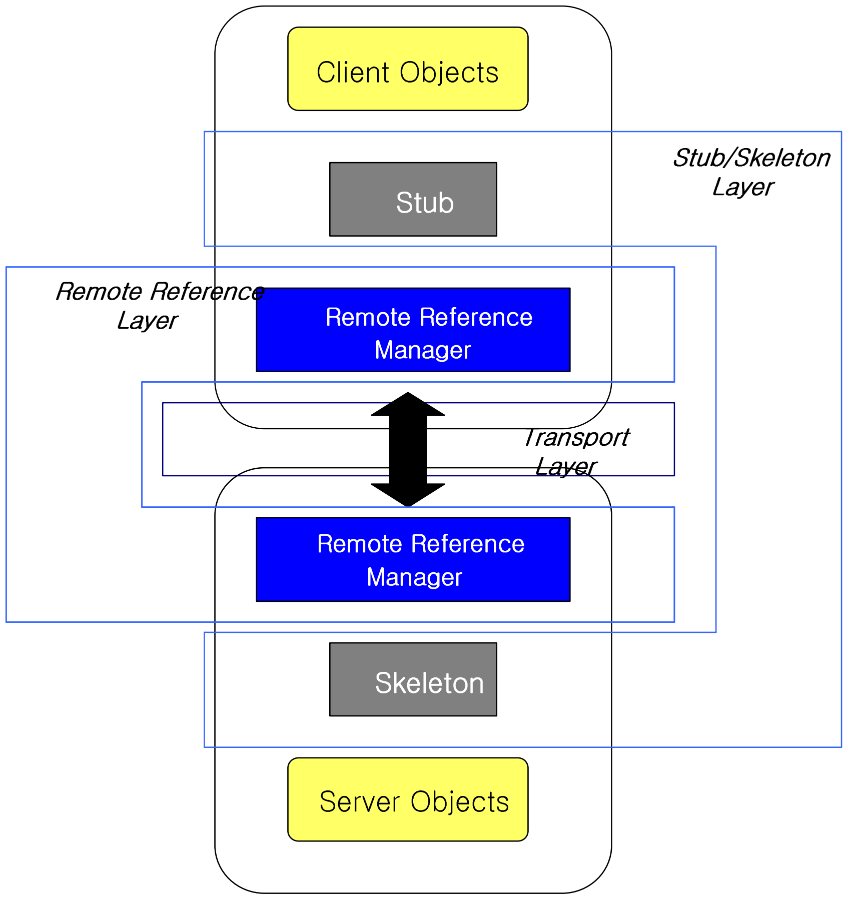


Figure 2.1 The RMI runtime architecture, courtesy: A. Wollrath

The stub/skeleton layer provides the interface between the application layer and the rest of the RMI system. In RMI, a *stub* for a remote object acts as a client's local representative or proxy for the remote object. A stub implements the same set of remote interfaces that a remote object uses. If a caller invokes a method on the local stub, then the stub carries out the method call on the remote object. When a stub's method is invoked, it initiates a connection with the remote JVM containing the remote object, marshals (writes and transmits) the parameters, receives them from the RRL, unmarshals (reads) the return value, and finally transmits the value to the client. A *skeleton* dispatches the call to the actual remote object implementation. When a skeleton receives an incoming method invocation, it unmarshals (reads) the parameters for the remote method, invokes the method on the actual remote object implementation, and marshals (writes and transmits) the result to the client[34]. However, the skeleton is no longer required in JDK 1.2 or newer versions because additional stub protocol was introduced.

The remote reference layer (RRL) is the middleware between the stub/skeleton layer and the underlying transport protocol. This layer handles the life of remote object references, controls the communication between client/server and virtual machines, and performs threading and garbage collecting for remote objects. Furthermore, it is responsible for the semantics of the invocation. For example, the RRL determines whether the server is a single object or is a replicated object requiring communications with multiple locations. Each remote object implementation chooses its own remote reference semantics – whether the server is a single object or is a replicated object requiring communications with multiple locations. Also the RRL handles the reference semantics for the server. The RRL abstracts the different ways of referring to objects that are implemented in (a) servers that are always running on some machine, and (b) servers that are run only when some method invocation is made on them. At the layers above the RRL, these differences are not seen[20].

The transport layer is the binary data protocol that sends remote object requests over the wire. It is different from OSI protocols transport layer. It is responsible for connection set-up, connection management; it keeps track of and dispatches to remote objects, targets of remote calls, in the transport's address space.

2.1.2 The RMI Syntax

To create an RMI application, the programmer has to satisfy certain requirements[28].

```
interface Hello extends java.rmi.Remote {
    void sayHello(String name) throws java.rmi.RemoteException;
}

class HelloImpl extends java.rmi.server.UnicastRemoteObject
                 implements Hello {
    public void sayHello(String name) throws java.rmi.RemoteException {
        System.out.println("Hello" + name);
    }
}

class Server {
    public static void main(String args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }

        Naming.rebind("myHost", new HelloImpl());
    }
}

class Client {
    public static void main (String args[]) {
        try{
            Hello hello = (Hello) Naming.lookup("myHost");
            Hello.sayHello("Jane!");
        }catch(java.rmi.RemoteException e)
    }
}
```

Table 2.1 General syntax for RMI

As Table 2.1 shows, remote objects should be made by following steps[26, 27].

- Define an *interface* for the remote class that extends the interface *java.rmi.Remote*, which must include every method available for remote invocation. While java.rmi.Remote does not define any methods, it serves as the marker interface that allows a RMI compiler and a runtime system to recognize remote interfaces. Also, all remotely accessible methods must be declared to throw a *java.rmi.RemoteException* that is used to report communication problems and to forward application exceptions from one JVM to another.
- Create a class that implements the interfaces. The HelloImpl class in Table 2.1 illustrates how a remote object is defined. By implementing the Hello interface, the HelloImpl is suitable for receiving remote invocations. It also extends one of RMI's classes *UnicastRemoteObject*, which provides methods for remote objects.
- Create a server program that creates an instance of the HelloImpl and binds the service with the RMI registry. It also assigns a security manager to the JVM, to prevent any untrusted clients using the service.
- Create a client program that accesses the remote interface. Compile the client program. The client finds the "myHost" object on the server and create a remote reference to this object by using *Naming.lookup*. This remote reference can be used to do RMI calls on the HelloImpl object.
- Compile all classes including interfaces using the javac, the Java compiler. Then compile the server/client programs using the rmic, the Java special stub compiler, which produces two extra layers of code. These are a skeleton which runs on server side and a stub on client side.

- Start the rmiregistry. A remote object registry is a bootstrap naming service that is used by RMI servers on the same host to bind remote objects to names. Clients on local and remote hosts can then look up remote objects and make remote method invocations[32].
- Start the server application followed by the client application.

2.1.3 The Process of RMI Communication

A client application is able to invoke a method of a remote object with two methods. First, the client gets the reference of remote object from registry, a bootstrap-naming service in server machine. Secondly, it gets the reference of remote object using parameters or a return value.

The RMI client invokes a method of remote object through the remote object reference. To forward a method invocation to another JVM, the RMI runtime system must be able to transfer method parameters and a result value from one JVM to another. This is done using serialization to preserve the object type. Serialization is object to provide the object-oriented polymorphism.

As Figure 2.2 shows, when a client calls a remote method, method parameters are encoded (marshalled) simultaneously and transmitted from the network layer to remote reference. A server carries out the processes successively: decodes (unmarshals) the parameters, executes the method, encodes (marshals) the result again and finally transmits the encoded results to the client. Undoubtedly, the client decodes and uses the return value from the stub.

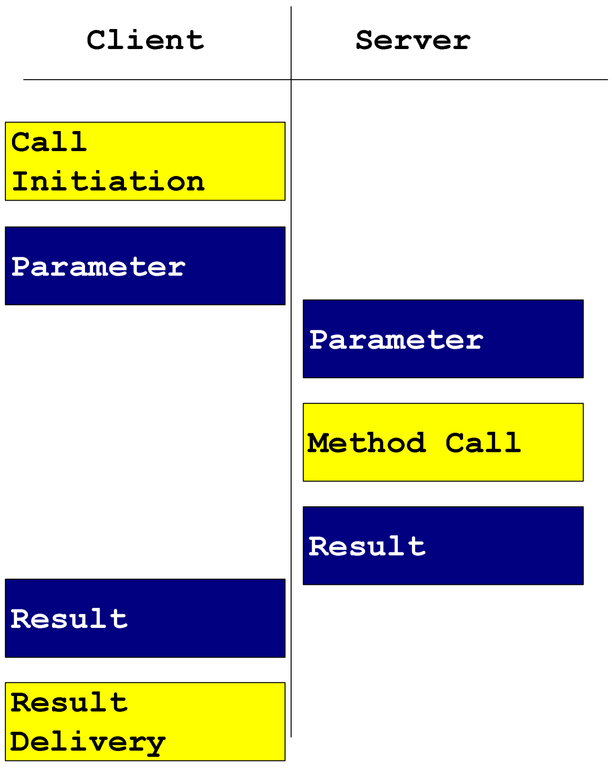


Figure 2.2 The stage of RMI call

2.2 Asynchronous RMI

As mentioned in the introduction, RMI, an object-oriented alternative of Remote Procedure Calls, is one of the most popular communication paradigms currently used in the mainstream of distributed computing, both in the industrial and scientific domains[31].

However, the synchronous nature of RMI leads network latency and effects to a limited performance. Thus, several implementations have been developed that support extended protocols for RMI. These include JavaParty, Manta, and NinjaRMI by changing the underlying protocols such as the serialization protocols. This section presents the considerable work completed in the field of RMI performance enhancement based on the previous articles[35].

2.2.1 Optimized Sequential RMI

There have been implementations, e.g. Ninja RMI and KaRMI, for optimizing the sequential RMI by the more effective marshalling and simplified channel initialization. These implementations decrease overhead of marshalling but there is no enhancement for Round-Trip Time (RTT).

2.2.2 Asynchronous RMI using Thread

When clients communicate with multiple objects/servers, a sequential communication pattern causes a bottleneck effect. There has been research using multithreading, which is natively supported in Java.

But it also has limitations. Ironically, one of the most important characterizations of RMI, such as thread management and garbage collection, cause the pause when the system enhances the performance. Also, each thread requires using separate system resources and TCP connections. As a result, the limited number of concurrent open TCP connections allows covering 100 invocations.

2.2.3 Asynchronous RMI with a Future Object.

In 1997, Object System provided a communication system called Voyager, providing several communication modes allowing for synchronous invocation, asynchronous invocations with no reply (one way), and asynchronous invocation with a reply (future/promise). Thereafter, several papers has implemented asynchronous RMI using this concept[19, 37, 24, 30, 25].

This way does not require an excessive number of threads and TCP connections. The records of all 1000 invocations executed stably. This model overcomes synchronization but it is still tightly coupled for space and time[14].

2.2.1 The Combination of Object-Orientation and Publish/Subscribe Communication

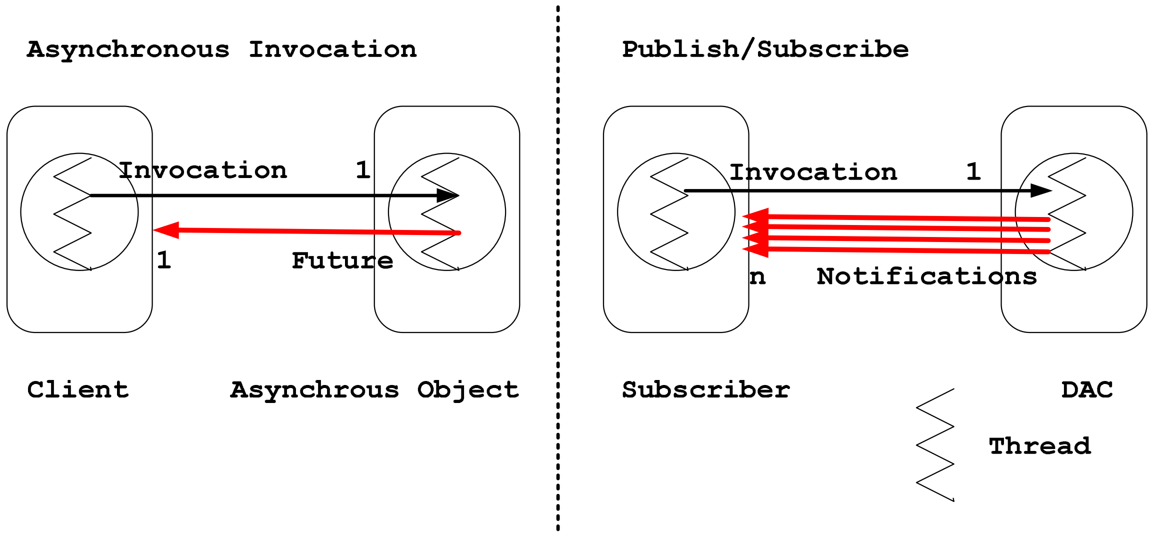


Figure 2.3 Asynchronous invocation vs. publish/subscribe, courtesy: P.T. Eugster

The classical categories of publish/subscribe communications are topic-based and content-based systems. However, the classical approaches have a limitation on designing the object oriented system. It claims that the combination of object-orientation and the publish/subscribe communication, might be applied to the current commercial practices in distributed object-oriented computing[16, 18], which are mainly based on the derivatives of the remote procedure call such as DCOM, Java RMI, and CORBA[15].

As Figure 2.3 shows, asynchronous invocation with a future object has the similar character with a publish/subscribe model, which will be a powerful tool for devising distributed applications. A *type-based* publish/subscribe system enables an object-oriented design and asynchronous communication. The type-based system characterizes to regroup events not only based on contents but also structure. This also facilitates a closer integration of the language and the middleware and guarantees type-safety.

2.3 Publish/Subscribe Communication

Starting with *the information bus* architecture, the publish/subscribe model comes in for the solution for "24 by 7" commercial environment, in which a distributed system must remain operational twenty-four hours a day, seven days a week. The information bus requires operating constantly and tolerating for dynamic system evolution and legacy system. *Providers* publish data to an information bus and *consumers* subscribe data they want to receive[29]. Providers and consumers are independent and need not even know of their existence. In general, the provider is called *the publisher*, the consumer is called *the subscriber*, and the information bus is called *middleware* or *broker*. With systems based on the publish/subscribe interaction scheme, subscribers register their interest in an event, or pattern of events, and are subsequently asynchronously notified of events generated by publishers[14].

As distributed systems on wide-area networks grow, the demand of flexible, efficient, and dynamic communication mechanisms are needed. The publish/subscribe communication paradigm provides a many-to-many data dissemination. It is an asynchronous messaging paradigm that allows for better scalable and a more dynamic network topology. The publish/subscribe interaction is an asynchronous messaging paradigm, characterized by the strong decoupling of participants in both time and space[38].

There are two most widely used approaches to publish/subscribe models. One is a topic-based system[29] and the other is a content-based one[7]. In the *topic-based system*, messages are published to *topics*, or named logical channels, which are hosted by a broker. Subscribers obtain all messages published to the topics to which they subscribe and all subscribers to the topic will receive the same messages. Each topic is grouped by keywords. In the *content-based system*, messages are only delivered to a subscriber if the attributes or content of those messages match constraints defined by one or more of the subscriber's subscriptions. This method is based on tuple-based system. Subscribers can get a selective event using filter in form of name-value pairs of properties and basic comparison operators (=, >, ≥, ≤, <). However, these approaches consider the different models for the middleware and the programming language. Consequently, the object-oriented and message-oriented worlds are often claimed to be incompatible[16]. Therefore, Patrick T. Eugster presents a type-based publish-subscribe which provides type safety and encapsulation[17].

2.4 Generics in the Java Programming Language

JDK 1.5 introduces several extensions to the Java programming language. One of these is the introduction of *Generics*. Generics is designed to increase the flexibility of object-oriented type systems with parameterized classes and polymorphic methods. A similar mechanism has recently been described for C#, and is likely to become part of a future version of that language[36].

Generics provides a way to communicate the type of collection to a compiler, so that it can be checked. Once a compiler knows an object's type, then the compiler can check that the object user has used the object consistently and can insert the correct casts on values being taken out of the object.

Generic methods allow type parameters to be used to express dependencies among types of one or more arguments to a method and/or its return type. If there isn't such a dependency, a generic method should not be used[12]. Using wild cards is clearer and more concise then declaring explicit type parameters, and should therefore be preferred whenever possible.

2.5 Scientific Computing and Grid Computations

This study focuses on developing a model for a large-scaled scientific computing and grid computation.

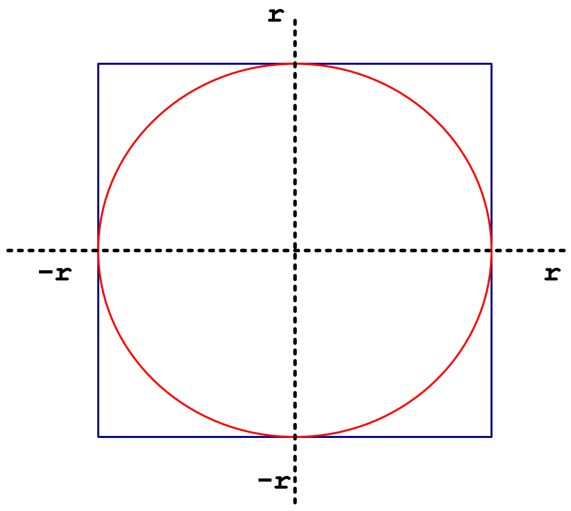
Scientific computing constructs mathematical models and numerical solution techniques using computers to analyze and solve scientific and engineering problems[39]. In practical use, it is typically an application of computer simulation and other forms of computation to problems in various scientific disciplines.

Grid computations start as a numerical solution to partial differential equations (PDEs). PDEs are used as a model because many phenomena in nature are mathematically described by PDEs, such as whether, airflow over a wing, turbulence in fluids, and so on. There are two methods to solve PDEs: a direct method and an iterative method. Simple PDEs can be solved directly, but in general, it is necessary to estimate the result at a finite number of points using iterative numerical methods.

Before this study moves on an iterative method, we will deal with the Monte-Carlo method to introduce parallelism. And we introduce Laplace's equation as a fundamental PDEs[10, 2] and Jacobi iteration as an iterative method to solve PDEs correspondingly.

2.5.1 π Calculation

There are a number of ways to calculate the value of π. Monte Carlo methods can be thought of as statistical simulation methods that utilize a sequences of random numbers to perform the simulation[21]. Given the possibility that an event will occur in certain conditions, a computer can be used to generate those conditions repeatedly. The number of times the event occurs divided by the number of times the conditions are generated should be approximately equal to the possibility[1].



$$A_S = (2r)^2 = 4r^2$$
$$A_C = \pi r^2$$
$$A_S : A_C = 4r^2 : \pi r^2$$
$$\pi = 4 \times A_C / A_S$$

Figure 2.4 π calculation

Figure 2.4 illustrates how to get π approximation. A_S is the area of a square and A_C is the area of a circle. The π value is derived from both A_S and A_C. Assuming that r is 0.5, i.e. the range of dots is between -0.5 and +0.5, we randomly select points in the square and count how many of them lie inside of the circle. Then we can approximately compute π according to the formula in Figure 2.4. For example, if 785 points are inside of the circle out of 1000 points, then π=4*785/1000=3.14. Statistically, the more points generated, the better the approximation gets. However, we set an accurate π value as 3.14159265 and a threshold as 0.0000001, and calculate an approximate π value within the threshold. Table 2.2 shows the details for a program to calculate π value.

```
public double calculate(int threadNo){
    THRESHOLD = 0.0000001;
    THRESHOLD = THRESHOLD * threadNo;

    double error = 1.0;

    while(error > THRESHOLD){
        double x = Math.random();
        double y = Math.random();
        double magX = .5 - x;
        double magY = .5 - y;

        boolean inUnitCircle=Math.sqrt(magX*magX + magY*magY) <= .5;
        if(inUnitCircle)    pointsInCircle++;
        pointsInSquare++;

        approxiPi = (double) pointsInCircle * 4 / pointsInSquare;
        error = Math.abs(approxiPi - ACCURATE_PI);
    }//while

    return approxiPi;
}//calculate

public int getPointsInCircle(){
    return pointsInCircle;
}

public int getPointsInSquare(){
    return pointsInSquare;
}
```

Table 2.2 Program of π calculation

2.5.2 Laplace's Equation

Laplace's equation is an example of an elliptic PDE. The equation for two dimensions is as follows:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0 \tag{2.1}$$

Assume that there is a two-dimensional space having coordinates *x* and *y*. Given a spatial region and values for points on the boundaries of the region, the goal is to approximate the steady-state solution for points in the interior[2].

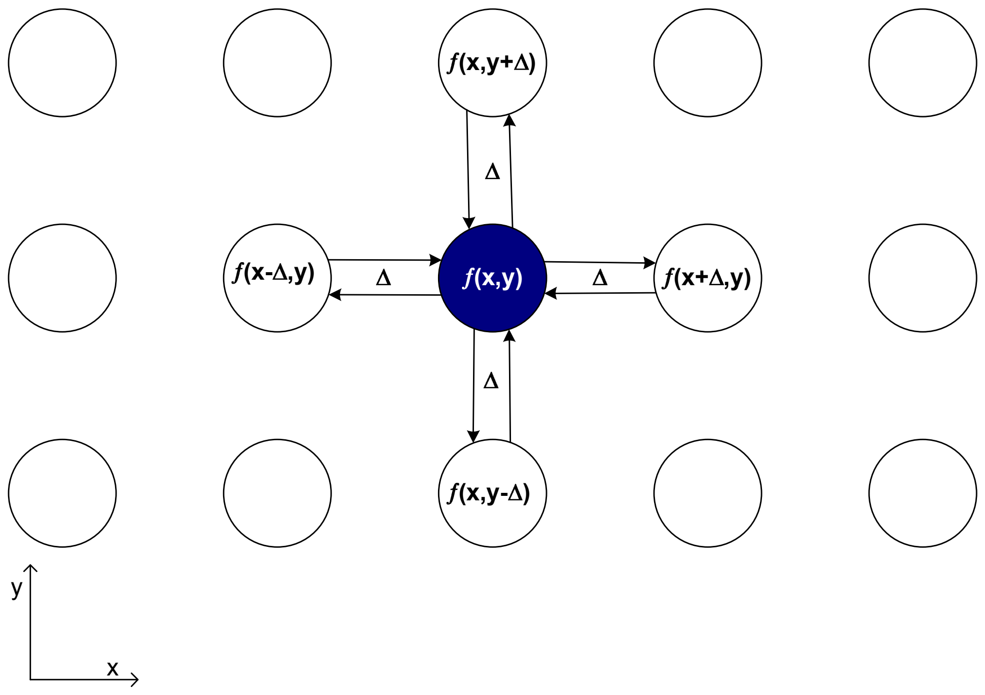


Figure 2.5 Computation of two- dimensional finite difference

As shown in Figure 2.5, we can evenly space a grid of points, the region and the interior points, and they are calculated by repeated iterations. The new value of a point is computed by the values of four neighboring points. The computation terminates after a given number of iterations or when the difference of each interior point, between a new and an old value, is less than a given value. If the distance Δ between *x* and *y* is small enough, the PDE of *f* for *x* and *y* can be described as first derivatives respectively:

$$\frac{\partial f(x,y)}{\partial x} = \lim_{\Delta x \to 0} \frac{f(x+\Delta,y) - f(x,y)}{\Delta} \tag{2.2}$$

$$\frac{\partial f(x,y)}{\partial x} = \lim_{\Delta x \to 0} \frac{f(x,y) - f(x-\Delta, y)}{\Delta} \tag{2.3}$$

And the second derivative for *x* is mentioned in equation (2.4):

$$\frac{\partial^2 f}{\partial x^2} = \frac{\frac{f(x+\Delta, y)}{\Delta} - \frac{f(x,y)}{\Delta}}{\Delta} \tag{2.4}$$

From the second derivative (2.4), we are led to formula (2.5) after we plug in the first derivative (2.2) and (2.3).

$$\frac{\partial^2 f}{\partial x^2} = \frac{\frac{f(x+\Delta, y) - f(x,y)}{\Delta} - \frac{f(x,y) - f(x-\Delta, y)}{\Delta}}{\Delta} \tag{2.5}$$

By rearranging, we have (2.6) and (2.7).

$$\frac{\partial^2 f}{\partial x^2} \approx \frac{1}{\Delta^2}[f(x+\Delta, y) - 2f(x,y) + f(x-\Delta, y)] \tag{2.6}$$

$$\frac{\partial^2 f}{\partial y^2} \approx \frac{1}{\Delta^2}[f(x, y+\Delta) - 2f(x,y) + f(x, y-\Delta)] \tag{2.7}$$

By substituting in Laplace's equation, we have

$$\frac{1}{\Delta^2}[f(x+\Delta, y) + f(x-\Delta, y) + f(x, y+\Delta) + f(x, y-\Delta) - 4f(x,y)] = 0 \tag{2.8}$$

By rearranging, we have

$$f(x,y) = \frac{1}{4}[f(x+\Delta, y) + f(x-\Delta, y) + f(x, y+\Delta) + f(x, y-\Delta)] \tag{2.9}$$

The formula can be written as an iterative formula:

$$f^k(x,y) = \frac{1}{4}[f^{k-1}(x+\Delta, y) + f^{k-1}(x-\Delta, y) + f^{k-1}(x, y+\Delta) + f^{k-1}(x, y-\Delta) \quad (2.10)$$

Where $f^k(x,y)$ is the value obtained from *k*th iteration, and $f^{k-1}(x,y)$ is the value obtained from (*k* -1)th iteration. By repeated application of the formula, we can converge on the solution.

We have several iterative methods for solving Laplace's equation including Jacobi and Gauss-Seidel algorithms. Gauss-Seidel algorithms may converge faster than Jacobi ones[3]. However, this study demonstrates Jacobi iteration due to its simplicity and readiness to be parallelized[2].

2.5.3 Jacobi Iteration

Jacobi's method is the simplest approach to designing an iterative method for solving $Ax = b$. This uses the first equation and the current values of $x_2^{(k)}, x_3^{(k)}, ..., x_n^{(k)}$ to find a new value $x_1^{(k+1)}$, where the superscript indicates the iteration. And then similarly we use the i^{th} equation and the old values of the other variables to find a new value $x_i^{(k+1)}$. Given current values $x_1^{(k)}, x_2^{(k)}, ..., x_n^{(k)}$, we find new values by solving for $x_1^{(k)}, x_2^{(k)}, ..., x_n^{(k)}$ in

$$\begin{array}{ccccccccc} a_{11}x_1^{(k+1)} & + & a_{12}x_2^{(k)} & + & ... & + & a_{1n}x_n^{(k)} & = & b_1 \\ a_{21}x_2^{(k)} & + & a_{22}x_2^{(k+1)} & + & ... & + & a_{2n}x_n^{(k)} & = & b_2 \\ ... & & ... & & ... & & ... & & ... \\ a_{n1}x_1^{(k)} & + & a_{n2}x_2^{(k)} & + & ... & + & a_{nn}x_n^{(k+1)} & = & b_n \end{array} \quad (2.11)$$

This can also be written as

$$\begin{bmatrix} a_{11} & 0 & \dots & 0 \\ \dots & a_{22} & \dots & \dots \\ 0 & \dots & \dots & 0 \\ 0 & \dots & 0 & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix}^{(k+1)} + \begin{bmatrix} 0 & a_{12} & \dots & a_{1n} \\ a_{21} & 0 & \dots & \dots \\ \dots & \dots & \dots & a_{n-1}a_n \\ a_{n1} & \dots & a_{nn-1} & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix}^{(k)} = \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{bmatrix} \qquad (2.12)$$

D, *L*, and *U* are the diagonal, the lower triangular of *A*, and the upper triangular of *A* respectively:

$$D = \begin{bmatrix} a_{11} & 0 & \dots & 0 \\ \dots & a_{22} & \dots & \dots \\ 0 & \dots & \dots & 0 \\ 0 & \dots & 0 & a_{nn} \end{bmatrix}, L = \begin{bmatrix} 0 & 0 & \dots & 0 \\ a_{21} & 0 & \dots & \dots \\ \dots & \dots & \dots & 0 \\ a_{n1} & \dots & a_{nn-1} & 0 \end{bmatrix} and\ U = \begin{bmatrix} 0 & a_{12} & \dots & a_{1n} \\ 0 & 0 & \dots & \dots \\ \dots & \dots & \dots & a_{n-1}a_n \\ 0 & \dots & 0 & 0 \end{bmatrix}, \qquad (2.13)$$

Then Jacobi's Method can be written more concisely in matrix-vector notation as

$$Dx^{(k+1)} + (L+U)x^{(k)} = b \qquad (2.13)$$

This formula is simply (*k* +1)th equation rearranged to have the (*k* +1)th unknown on the left side.

$$x^{(k+1)} = D^{-1}\left[(-L-U)x^{(k)} + b\right] \qquad (2.14)$$

That is, for elements $x_i^{(k+1)}$,

$$\begin{aligned} x_i^{(k+1)} &= \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i}^{n} a_{ij} x_j^{(k)} \right) \\ x_i^{(k+1)} &= \frac{1}{a_{ii}}\left(b_i - \sum_{j\neq i} a_{ij} x_j^{(k)} \right) \end{aligned} \qquad (2.15)$$

It is proven that the Jacobi method will converge if the diagonal values have absolute values greater than the sum of the absolute values of the other *a*'s on the row. This condition is called that the array of *a* is *diagonally dominant*. Therefore, the convergence is guaranteed if

$$\sum_{j \neq i} | a_{i,j} | < | a_{i,i} | \tag{2.16}$$

This condition is sufficient but not necessary. The method may converge even if the array is not diagonally dominant. However, the iteration formula will not work if any of the diagonal elements are zero because it requires dividing by zero[10]. Because iterative methods may not always converge, we terminates the computation in the *i*th iteration when all values are within a given error tolerance, i.e. $e^k > | x_i^{(k)} - x_i^{(k-1)} |$. Also, iterations should stop the process when a maximum number of iterations have been reached. Since the parallel formulation requires all iterations to use the previous iteration's values, the calculations have to be synchronized globally.

- Sequential Jacobi Iteration

In Jacobi iteration, the new value for each grid point is set to the average of the old values of the four neighboring points left, right, above, and below it. This process is repeated until the computation terminates. Table 2.3 shows suede codes for a sequential Jacobi program. This program works with two copies of matrix, oldMtx and newMtx. One copy represents the grid and its boundary and another copy represents the set of new values, i.e., oldMtx performs only read operations, and newMtx is for write only.

The boundaries of both matrices are initialized to the appropriate boundary conditions, and the interior points are initialized to zero, a starting value. Assume that the program terminates the computation when every new value on iteration is within EPSILON of its prior value. Then the main computational loop for Jacobi iteration is as follows:

```
double oldMtx[0:n+1, 0:n+1], newMtx [0:n+1, 0:n+1]
double maxdiff;
int iters;

while(true){
    //1.compute new values for all interior points
    for(i=1 to n, j=1 to n)
        newMtx[i,j] = (oldMtx[i-1,j]+ oldMtx[i+1,j]+
                        oldMtx[i,j-1]+oldMtx[i,j+1])/4;
    iters++;

    //2.compute the maximum difference
    maxdiff=0.0;
    for(i=1 to n, j=1 to n)
        maxdiff = max(maxdiff, abs(newMtx [i,j]- oldMtx[i,j]));

    //3.check for termination
    if(maxdiff < EPSILON)
        break;

    //4. copy newMtx to oldMtx to prepare for next updates
    for(i=1 to n, j=1 to n)
        oldMtx[i,j] = newMtx[i,j];
}
```

Table 2.3 Sequential Jacobi iteration

This code assumes that arrays are stored in row-major order as in C or Java, which loops iterate over *i* then *j*. However, loops iterate over *j* then *i* if arrays are stored in column-major order as in Fortran.

This code is correct but not efficient. So we can improve its performance with some changes. In the first loop, the division by 4 can be replaced by the multiplication by 0.25 because it takes fewer machine cycles to execute a multiplication than a division. This optimization is called *strength reduction*, which replaces a strong and expensive operation with a weaker one. In fact, it could be replaced with an even weaker operation, shift right by 2.

And when we compute maximum difference, we can get rid of the overhead of two function calls, *abs* and *max* using *function inlining* as follows: Function inlining optimize compiler by expanding the body of the function inline instead of calling and returning from a function.

```
double temp = oldMtx[i, j] - newMtx[i, j];
if(temp <0) temp = - temp;
if(temp > maxdiff) maxdiff = temp;
```

Table 2.4 Functional inlining for enhanced performance

- Parallel Jacobi Iteration

Suppose that we have *PR* processors and the dimensionality of the grid, *n*, is much larger than *PR*. We can divide the grid either into *PR* rectangular blocks or into *PR* rectangular strips. This thesis uses strips because that is easier to implement and more efficient. Long strips have better data locality than shorter blocks, and this leads to better use of data cache[2].

Assuming that *n* is a multiple of *PR* and arrays are stored in memory in row-major order, each process is assigned to a horizontal strip of size $n/PR \times n$. Each process updates its strip of points. Also we need to use *barrier synchronization*, between after every process has completed one update phase and before any process begins the next one. The processes share the points of the edges of the strips. Table 2.5 contains a parallel program for the Jacobi iteration and Figure 2.5 shows the detailed communications for exchanging the edges and boundaries between two processes.

```
int n;
int PR;
int height = n/PR;

double oldMtx[0:height+1, 0:n+1], newMtx [0:height+1, 0:n+1]

//1. initialize old and new matrix, including boundaries;
for (i=0 to height+1){
    oldMtx[i,0] = 1;
    oldMtx[i,n+1] = 1;
    newMtx[i,0] = 1;
    newMtx[i,n+1] = 1;
}

for (i=0 to n+1){
    oldMtx[0,i] = 1;
    oldMtx[height+1,i] = 1;
    newMtx[0,i] = 1;
    newMtx[height+1,i] = 1;
}

//2. Compute new matrix values
for(i=1 to height)
    for(int j=1; j <= n; j++)
      newMtx[i,j] = (oldMtx[i-1,j]+oldMtx[i+1,j]
                    +oldMtx[i,j-1]+oldMtx[i,j+1])*0.25;

//3. Send edges of new to neighbors
if(id >0)
    send newMtx[1,*] to the newMtx[height+1,*] of (id-1) process

if(id <nThread-1)
    send newMtx[height,*] to the newMtx[0,*] of (id+1) process

//4. Receive the value from neighbors to boundaries of new
if(id <nThread-1)
    receive the value to newMtx[height+1,*]
if(id >0)
    receive the value to newMtx[0,*]

//5. Compute old matrix values for interior of my strip
for(i=1 to height, j=1 to n)
    oldMtx[i,j] = (newMtx[i-1,j]+newMtx[i+1,j]
                  +newMtx[i,j-1]+newMtx[i,j+1])*0.25;

//6. Compute maximum difference for my strip
for(i=1 to height, j=1 to n){
    double diff = oldMtx[i,j] - newMtx[i,j];
    if(diff <0)   diff = -diff;
    if(diff > mydiff)   mydiff = diff;
}
```

Table 2.5 Parallel Jacobi iteration

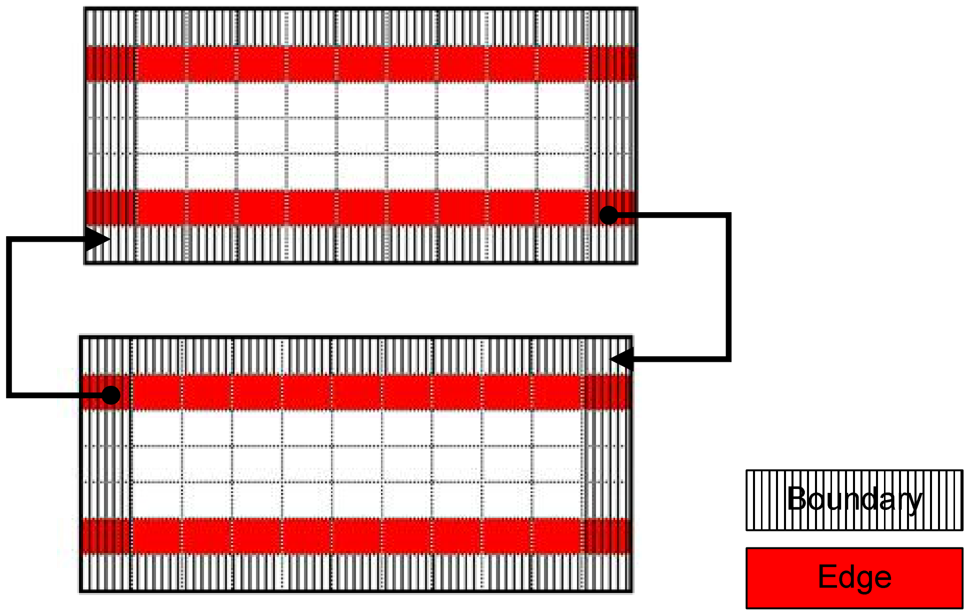


Figure 2.6 Exchange the edges and boundaries between two processes

Each process executes the same code but operates on different parts of the data. For instance, if the size of matrix is 100*100 and the number of process is 5, then each process will have 100*20 matrixes. In the initial condition, all the cells are set to 0 and the boundary to 1. The boundary is a ghost layer communicated between processes. As Figure 2.6 shows, each process communicates the ends of each row (red one) after it calculates the value of the matrix. Neighboring workers exchange edges twice per iteration of the main computational loop. All workers but the first send the top row of their strip to the neighboring one above, and all workers but the last send the bottom row of their strip to the neighbor below. Each worker then receives edges from its neighbors and these become the boundaries of each worker's strip. The second exchange is identical, except that oldMtx is used instead of newMtx.

After the appropriate number of iterations, each worker computes the maximum difference for its strip, and the first worker collects these values.

This program is optimized for better performance. First, it is not necessary to exchange edges after every update phase. We could exchange edges after every other update. Secondly, we can reprogram the remaining exchange to do local computation between the sends and receives. In particular, we can have each worker (1) send its edges to neighbors, (2) update the interior points of its strip, (3) receive edges from neighbors, and (4) update the edges of its strip. This will greatly increase the possibility that neighboring edges will have arrived before they are needed, and hence, receive statements will not be delayed.

2.5.4 Asynchronous Iterative Algorithms

We will start off with a mathematical model of *synchronous* and *asynchronous iterations* considering an iteration of the form $x := f(x)$, where f is the iteration mapping defining the algorithm[9]. There are *p* processors with the *i*th processor assigned the responsibility of updating the *i*th component x_i according to the rule $x_i := f_i(x_1,...,x_p)$. We say that an execution of iteration is *synchronous* if it can be described mathematically by the formula, $x(k+1) = f(x(k))$ where *k* is an integer-valued variable used to index different iterations, not necessarily representing real time. On the contrary, in *asynchronous iteration*, processors do not necessarily have to wait in anticipation of collecting all messages generated during the previous iteration. Each processor keeps updating its own part at its own speed. When the current value to be updated by other processor is not available, an out-of-date value is used instead. Asynchronous convergence has been proved by several authors starting with the work of Chazan and Miranker[13], under the name of *chaotic relaxation*.

Now, we present communication categories with *Synchronous* and *Asynchronous communications*. *Synchronous communications* require handshaking between tasks that are sharing data. This can be explicitly ordered in code by a programmer, or it may be occur at a low level unknown to the programmer. This is often referred to as *blocking communications* since other work must wait until the communications have completed. *Asynchronous communications* allow tasks to transfer data independently from one to another. For example, task *A* sends a message to task *B*, and then immediately begins doing other work whenever or not task *B* receives the data. So asynchronous communications are often referred as *non-blocking* communications since other works can be done while the communications are taking place. Interleaving computation with communication is the greatest benefit to using asynchronous communications [8].

Bahi, Couturier and Vuillemin seperates the classification of parallel iterative algorithms into three main parts, *synchronous iterations-synchronous communications* (*SISC*), *synchronous iterations-asynchronous communications* (*SIAC*), and *asynchronous iterations-asynchronous communications* (*AIAC*) algorithms[5, 4, 3]. They renamed AIAC algorithms to Asynchronous iterative algorithms. In *SISC* algorithms, all the processors initiate the same iteration at the same time because data exchanges are performed at the end of each iteration by synchronous global communications. In *SIAC* ones, all the processors also wait for the receptions of needed data updated at the previous iteration before they can begin the next one. In *AIAC* algorithms, all the nodes perform their iterations without considering the progression of the others because local algorithms do not need to wait for required data. In general, AIAC algorithms have a greater number of iteration times to be converged. But the execution time could be significantly reduced. Finally, they present a way to implement synchronous and asynchronous versions. In *synchronous algorithms*, each processor reports its local error every iteration time to the master node. And stops its activity and waits until it receives the convergence response from the master. Therefore, this period effects the idle times. When the master has received all the local errors, it computes the global error of the whole system and decides whether the global convergence is achieved or not. Then the master node sends the convergence responses to the others, which keep on computing. In *asynchronous algorithms*, the nodes only send a message to the master when their local convergence state changes and stays constant during several iterations. Hence, there are no idle times between two iterations. The master decides that global convergence is reached when all the nodes are in a local convergence state at a given time. Then it orders the other processors to stop computing. Finally, in both versions, as long as the convergence is not reached, the algorithm computes its iteration in order to get more accurate results.

CHAPTER 3

System Architecture

The goal of this study is to design a communication framework (PARMI) for complex large-scaled scientific applications to minimize their overall execution time. The challenging issues are follows: 1) how to overcome a synchronous and point-to-point communication nature of RMI, 2) how to provide a scalable framework for dynamic applications, and 3) how to maintain a strong decoupling of participants in both time and space. This research adopts a publish/subscribe communication paradigm on the object-oriented language framework, Java. Publish/subscribe provides a point-to-point communication and a strong decoupling of participants in both time and space. With a future object in Java, we overcome the synchronous nature of communication. Also Generics in Java makes it possible to apply for any dynamic situation. Java's object-oriented character helps to build a scalable system.

This chapter presents a design to implement the PARMI. Section 3.1 presents initial-phase implementations with the future object, to reach the final destination for providing a publish/subscribe communication. And Section 3.2 presents the PARMI with an adapter module for providing a publish/subscribe communication.

3.1 Asynchronous RMI with a Future Object

As studied in Chapter 2, RMI design with a future object is the latest and most suitable design for providing asynchronous communication between a client and a server. For a concrete and qualified test, this research also implements the RMI application with a future object and will be the criterion when we compare with our final goal, an asynchronous RMI providing a publish/subscribe communication.

3.1.1 Conceptual Architecture

As mentioned in Section 2.2.3, a future object helps a client not to be suspended while computation is carried out. Thanks to built-in classes and interfaces for the future object which holds a result of an asynchronous call, we don't need to spend time to implement the future object after Java version 1.5 or later. The previous asynchronous RMI studies have manipulated the stub class, which is generated automatically by an rmic compiler. It produces many maintenance difficulties. For example, if a method which is invoked remotely by an object is changed, then the corresponding classes and interfaces should be changed. After a stub class is generated by an rmic compiler, we must change the stub class manually. Therefore, this study does not attempt to change the stub class and add some codes in client side to use the FutureTask.

The following programs calculate the π value using the Monte-Carlo method, which is defined in Section 2.5.1. Several client-side threads send signals to invoke server-side methods to calculate the value of π.

The *CalculatePi* interface used by the server is also used by the client. On the server side, there are *CalculateServer* and *CalculatePiImpl* classes. CalculateServer class creates the registry and binds CalculatePiImpl object to the registry. CalculatePiImpl class implements CalculatePi interface, i.e. CalculatePiImpl class has all business logics for methods of CalculatePi. The threshold is equally divided by the number of clients because the whole threshold is fixed. Therefore, the client submits the number of clients as an input parameter when it invokes a server-side method. Table 3.1 shows detailed server-side implementations for asynchronous RMI codes with the future object.

```
public class CalculateServer {

    public CalculateServer(){
        try{
            // Create Remote Object
            CalculatePiImpl piRef =
                       new CalculatePiImpl("CalculatePi");

               // Create the registry
            // and bind the Server class to the registry
            LocateRegistry.createRegistry(1099);
            Registry r= LocateRegistry.getRegistry();
            r.bind(piRef.getName(), piRef);

        }catch(Exception e){}
    }

    public static void main(String[] args){
        // Create and install a security manager
        System.setSecurityManager(new RMISecurityManager());
        new CalculateServer();
    }
}


public interface CalculatePi extends Remote{
    public double calculate() throws RemoteException;
}

public class CalculatePiImpl extends UnicastRemoteObject
                         implements CalculatePi{
    private String name;
    private static final double ACCURATE_PI = 3.14159265;
    private static double THRESHOLD;
    private int pointsInCircle =0;
    private int pointsInSquare =0;
    private double approxiPi =0;
    static final boolean debug =false;
    private static long startTime, endTime;
    private static NumberFormat df =NumberFormat.getInstance();

    public CalculatePiImpl(String name) throws RemoteException{
        super();
        this.name= name;
        df.setMaximumFractionDigits(10);
    }

    public String getName(){
        return name;
    }

    public double calculate(int threadNo){
        THRESHOLD = 0.0000001;
        THRESHOLD = THRESHOLD * threadNo;

        double error = 1.0;

        while(error > THRESHOLD){
            double x = Math.random();
            double y = Math.random();
            double magX = .5 - x;
            double magY = .5 - y;
            boolean inUnitCircle=
                    Math.sqrt(magX*magX + magY*magY) <= .5;
            if(inUnitCircle)    pointsInCircle++;
            pointsInSquare++;

            approxiPi =(double) pointsInCircle * 4 / pointsInSquare;
            error = Math.abs(approxiPi - ACCURATE_PI);
        }//while

        return approxiPi;
    }//calculate

    public int getPointsInCircle(){
        return pointsInCircle;
    }

    public int getPointsInSquare(){
        return pointsInSquare;
    }
}
```

Table 3.1 RMI codes in the server side

The client-side needs *CalculatePi* interface, *Main* class, and *CalculateClient* class. Main class creates several CalculateClients and invokes calculate() method using the FutureTask object. By calling run() method in FutureTask, call() method in

CalculateClient is invoked. All FutureTasks are stored in HashTable and the results can retrieved from CalculateClient when a computation has completed. CalculateClient class implements Callable interface to provide a future object. Table 3.2 has the detailed implementations for client-side asynchronous RMI codes with a future object.

```
public class Main{
    static final boolean debug =false;
    static long startTime, endTime;
    static DecimalFormat df = new DecimalFormat("###,###,###.###");

    public static String url;
    public static int NO_PROC;
    static private Hashtable<Integer,FutureTask> future;
    static double pi;

    public Main() throws InterruptedException{
        future = new Hashtable<Integer,FutureTask>();

        // start invoke the tasks
        for(int i=0; i< NO_PROC; i++){
            FutureTask<Double> client=
            new FutureTask<Double>(new CalculateClient(i+""));
            client.run();
            future.put(i, client);
        }

        // get the future result
        for(int i=0; i< NO_PROC; i++){
            FutureTask<Double> client = future.get(i);

            try{
                pi +=client.get();
                if(debug)
                    System.out.println("pi +: "+pi);

            }catch(InterruptedException e){ e.printStackTrace();
            }catch(ExecutionException e){ e.printStackTrace();}

        }

        pi = pi/NO_PROC;
        System.out.println("pi: "+pi);

        endTime = System.currentTimeMillis();
        System.out.println("time spend: "+
            df.format(endTime - startTime));
    }
    public static void main(String[] args){
        startTime = System.currentTimeMillis();

        // Print the program usage
        if(args.length != 2) {
            System.out.println("Usage : hostName NO_PROC");
            return;
        }

        // Create and install a security manager
        if (System.getSecurityManager() == null)
            System.setSecurityManager(new RMISecurityManager());

        // Create the url string
        url = args;
        NO_PROC = Integer.parseInt(args[23]);

        try{
            new Main();
        }catch(InterruptedException e){}
        }
}

public interface CalculatePi extends Remote{
    public double calculate() throws RemoteException;
}


public class CalculateClient implements Callable<Double>{
    static final boolean debug =false;

    String id = null;
    CalculatePi cp = null;

    public CalculateClient(String id){
        this.id = id;
    }

    public Double call() throws Exception{
        //Get a remote reference by calling Naming.lookup()
        Registry r= LocateRegistry.getRegistry(Main.url);
        cp= (CalculatePi)r.lookup("CalculatePi");

        // Now use the reference cp to call remote methods
        return cp.calculate(Main.NO_PROC);
    }
}
```

Table 3.2 RMI codes in the client side

3.1.2 Operation of the Asynchronous RMI with a Future Object

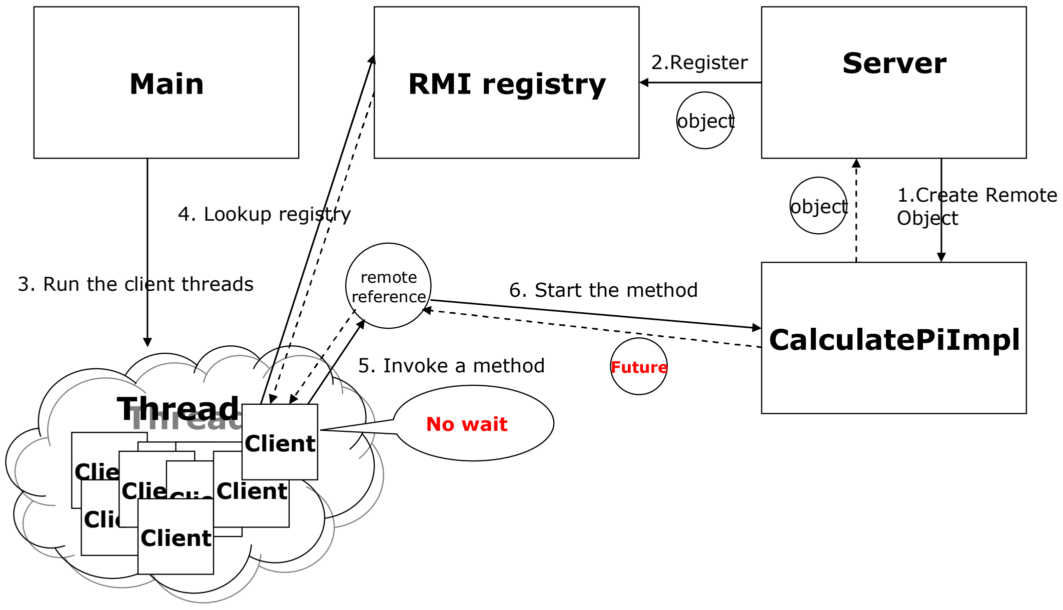


Figure 3.1 Operation of the asynchronous RMI with a future object

Figure 3.1 shows the operations of each class. After Server registers CalculatePiImpl object to RMI registry, a client is able to get a remote reference from the RMI registry. Once a client invokes a method, it proceeds with the remaining works without waiting because a future object is returned instantly when it is called.

3.1.3 Experimental Evaluation

The tests were performed to explore the different costs of standard RMI synchronous methods and our extended asynchronous method with future objects. These tests were performed on a local machine with a different directory with 2.8GHz processor running JDK 1.5.0_08 and two remote Windows machines with 2.8GHz and 1.86GHz processors connected by 10MB Ethernet running JDK 1.5.0_08 and 1.5.0_09. Performance results were carried out in groups of 5 to 10,000 threads. Measurements were performed using the *System.currentTimeMillis()* method of Java. π value was calculated on the conditions of ACCURATE_PI = 3.14159265 and THRESHOLD = 0.0000001.

Threads	**Sync(A)** **(milliseconds)**	**Future(B)** **(milliseconds)**	**A-B((A-B)/A)** **(milliseconds(%))**	**Exception**
20	594	407	187(31.48)	
40	828	453	375(45.29)	
60	969	500	469(48.40)	
80	1,265	516	749(59.21)	
100	1,312	562	750(57.16)	
300	3,235	906	2,329(71.99)	Sync
400	4,031	1,062	2,969(73.65)	Sync
1,000	12,343	1,906	10,437(84.55)	Sync
100,000		87,672		Sync

Table 3.3 Sync/async RMI over local calls.

Threads	**Sync(A)** **(milliseconds)**	**Future(B)** **(milliseconds)**	**A-B((A-B)/A)** **(milliseconds(%))**	**Exception**
20	406	406	0(0.00)	
40	610	422	188(30.82)	
60	3,454	1,187	2,267(65.63)	
80	3,704	1,328	2,376(64.15)	
90	6,922	641	6,281(90.74)	
95	3,469	1,485	1,984(57.19)	
100	3,546	703	2,843(80.17)	
300	10,031	2,203	7,828(78.04)	
400	14,812	2,625	12,187(82.28)	
1,000	21,156	7,313	13,843(65.43)	
10,000		38,688		Sync

Table 3.4 Sync/async RMI over remote calls.

Table 3.3 displays the execution costs for synchronous and asynchronous RMI calls with future objects on local modes. This table shows that asynchronous calls have less execution time than synchronous calls. We encountered java.net.ConnectionException on the synchronous RMI call with more than 300 threads. This exception is because of the limited number of concurrently opened sockets. On the contrary, the asynchronous calls with future objects show very stable executions until it finishes its processes.

Table 3.4 shows the execution costs for synchronous and asynchronous RMI calls with future objects on remote modes. The same exceptions started to occur on the synchronous RMI call with more than 100,000 threads.

The following figures show the graphs for Table 3.3 and 3.4 respectively.

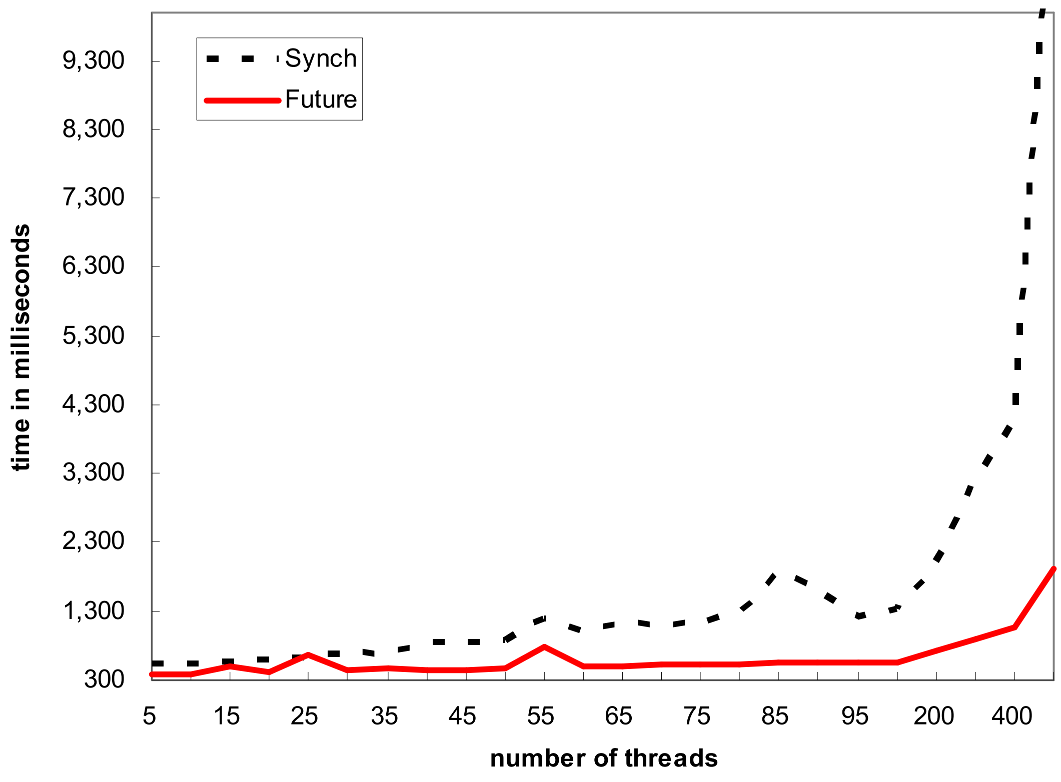


Figure 3.2 Sync/async RMI over local calls.

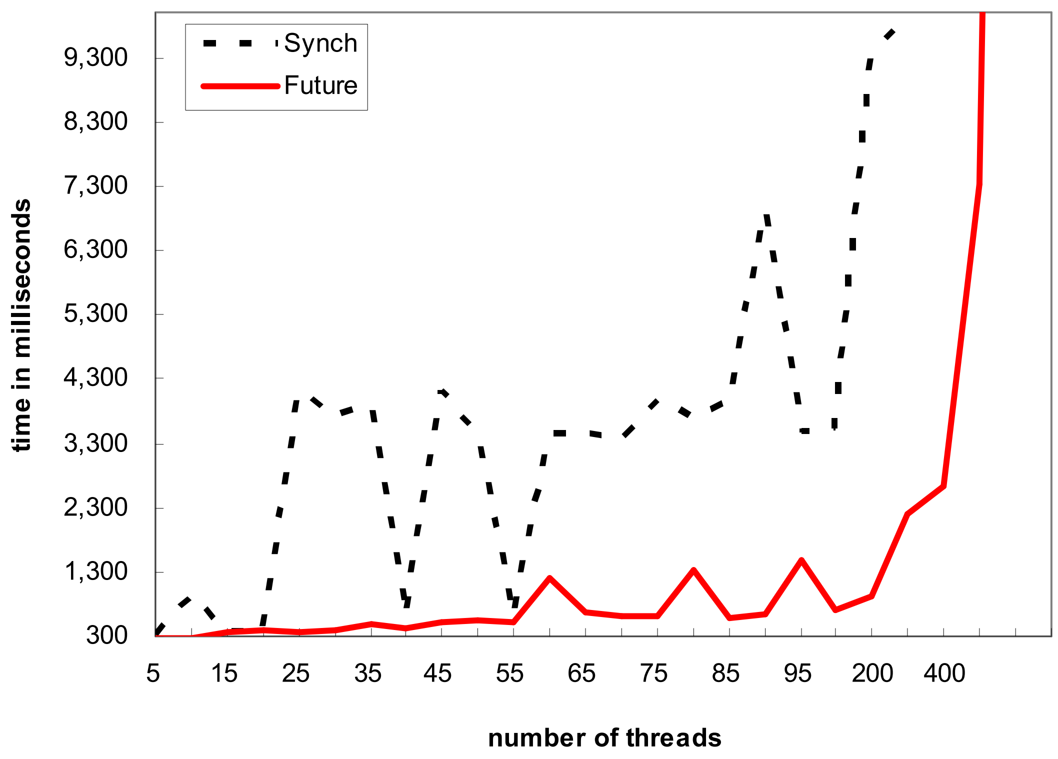


Figure 3.3 Sync/async RMI over remote calls.

As Figure 3.2 and 3.3 shows, the asynchronous communication with future objects decreased the execution time significantly on both local and remote calls. And also the exceptions for connection failures were decreased when we chose the asynchronous way of communication. With these considerable experiments, we reached the conclusion that the future object will contribute to performance improvement on our PARMI framework.

3.2 PARMI Framework Architecture

We added a new function to the adapter class to provide a publish/subscribe paradigm. The function of the adapter class is to keep track of those instances that are interested in the method's results. Thus, the adapter needs to watch the status of items holding interests and results, to determine when a result is ready. We chose the Observer-Observable design pattern for this function. The *Observer-Observable design* is a very useful pattern for maintaining one-way communication between one object and a set of other objects[41]. This design pattern consists of observers and observables and the communication is strictly from the observables to the observers. *Observers* only receive communications from the observable. Observers do not have a reference back to the Observable. *Observable* is the object that Observers are watching. Observers are always notified via their update methods whenever the Observable's update method is called. We have applied the observable to an item and the observer to a subscriber in PARMI system. If a subscriber is interested in an item, the subscriber is notified via its method whenever the item is changed.

When publishers and subscribers execute their jobs, they need to invoke methods in the adapter asynchronously for better performance. Based on the experimental result achieved with a future object, we can implement asynchronous invocations with the future object. Further, RMI supports an object-oriented communication framework for distributed computation in a heterogeneous network on remote address space.

3.2.1 Conceptual Architecture

In the previous chapter reviewed, we have introduced three design models for Publish/Subscribe: the topic-based, the content-based, and the type-based system. Each design model for the publish/subscribe system offers different degrees of expressiveness and performance overhead. *The topic-based* design model is rather static and primitive, but can be implemented very efficiently. *The content-based* one is highly expressive but requires sophisticated protocols that have higher runtime overhead. Therefore, we select the type-based one for PARMI which is suitable for an object-oriented system. In *the type-based* design model, an object can be a class, a method, or a variable. The object is implemented by generic code which provides the Meta programming environment and guarantees run-time type safety.

Also there is a study of formal approaches for a publish/subscribe communication system with respect to the semantic notification of information[6]. Therefore, we also can consider the formal approach for terminologies and operations for each process.

3.2.2 Terminology

We have four main components to provide the publish/subscribe communication: *publishers, subscribers, the adapter,* and *items*. The publish/subscribe system is composed by a set of process, $\Pi = \{p_1 \ldots\ p_n\}$ that communicate by exchanging information items.

Each process p_i can be either a publisher or a subscriber. If p_i is *a publisher*, then it sends the data to the adapter when it is ready. On the contrary, p_i is *a subscriber* S_i, it registers itself with an item in the adapter to receive the data interested in, or releases itself from an item in the adapter if the subscriber doesn't want to subscribe anymore. p_i interacts with other processes by subscribing the class of events interested in and by publishing event notification. Both cases invoke methods of the adapter whenever they execute their operations. In the PARMI, publishers and subscribers carry out the same role as clients in the existing RMI system.

The adapter is a central entity which keeps a set of all available items for publishers and subscribers in a hierarchical structure. The set is present as a form of Hashtable €= $\{x_1 \ldots x_m\}$ where x_i is an item. The set has a unique topic as a key mapping with an item x_i for a value as Hashtable. The adapter collects subscriptions and forwards events to subscribers. The adapter executes the same role as a server in the existing RMI system.

An item x_i is an element in the adapter holding the mapping information including a topic, a value, and a remote reference collection ₤= $\{S_1 \ldots S_m\}$, which subscriber S_i is interested in. It can be generated by either a publisher or a subscriber when submitting the value v about a specific topic to the adapter at the very first time. When a subscriber informs the adapter that it is interested in a topic, the adapter searches the set € if the item x_i with the topic exists or not. If the adapter already has the item x_i, then returns it. Otherwise, the adapter creates a new item. When a publisher sends a value with a topic, the adapter creates an item if it does not exist on the set €. Otherwise, the adapter updates the value of the existing item x_i.

Figure 3.4 presents the four elements in the PARMI system. Because *publishers* and *subscribers* use the same objects and methods from the RMI system, their structures preserve the stub/skeleton and the remote reference layer. As contrasted with these elements, the adapter and items are newly created objects that provide the publish/subscribe communication. *The adapter* is a storage unit providing the event-based service between publishers and subscribers. Basically, the adapter carries out an intermediary role to invoke a method and to forward the result from a subscriber. The detailed operations will be presented in the next section.

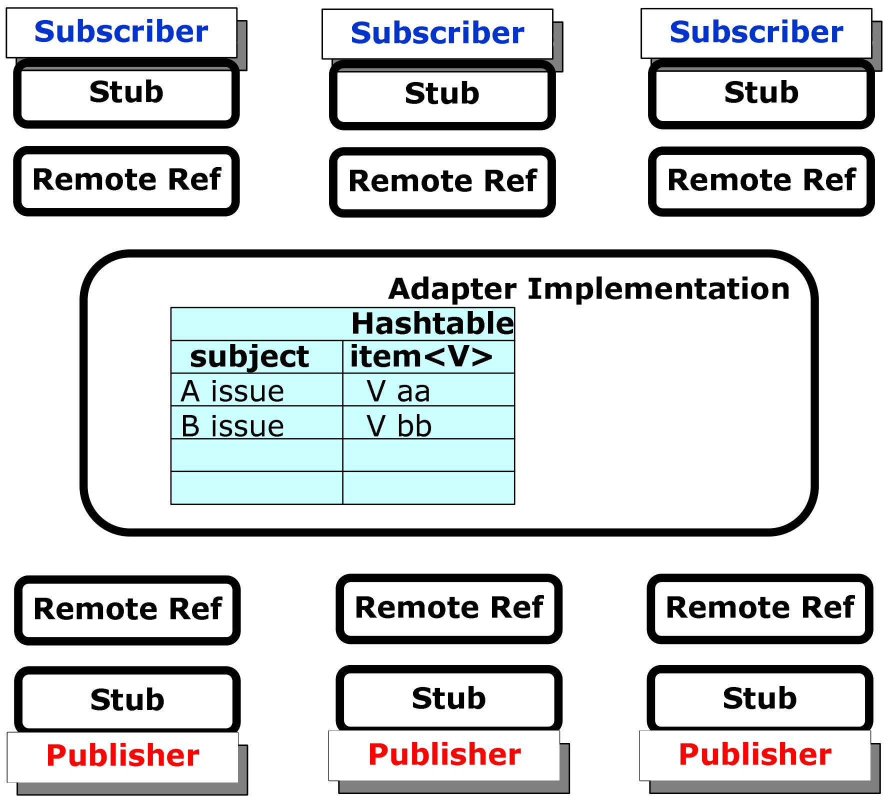


Figure 3.4 PARMI organization

3.2.3 Operation of Asynchronous RMI

Each process p_i executes the following operations: *publish*$_i$*(v, t)*, *register*$_i$*(*S_i*, t)*, *unregister*$_i$*(*S_i*, t)*, where v is a value of an information item, t is a topic for publish and subscribe, and S is a subscriber itself. Because all three methods exist in the adapter and return types are void, p_i invokes each method using RMI with a future object and passes the control to the adapter. All reference data such as method's name, method's parameter, and remote reference, are handed over to the adapter, but p_i does not need to wait until the method finishes its work.

Each process p_i can submit a value v with a topic t to the adapter by executing *publish*$_i$*(v, t)* operation. And p_i also can receive a value v submitted by other processes by executing an upcall to *subscribe*$_i$*(v, t)*. Subscriptions are respectively installed and removed at each process by calling *register*$_i$*(S, t)* and *unregister*$_i$*(S, t)* operations. That is p_i receives a value v only after registering its interest by calling *register*$_i$*(S, t)*. After that, if p_i carries out *unregister*$_i$ *(S, sub)*, then it is excluded from the subscription list. Figure 3.5 and 3.6 present how the PARMI system works when each method is called.

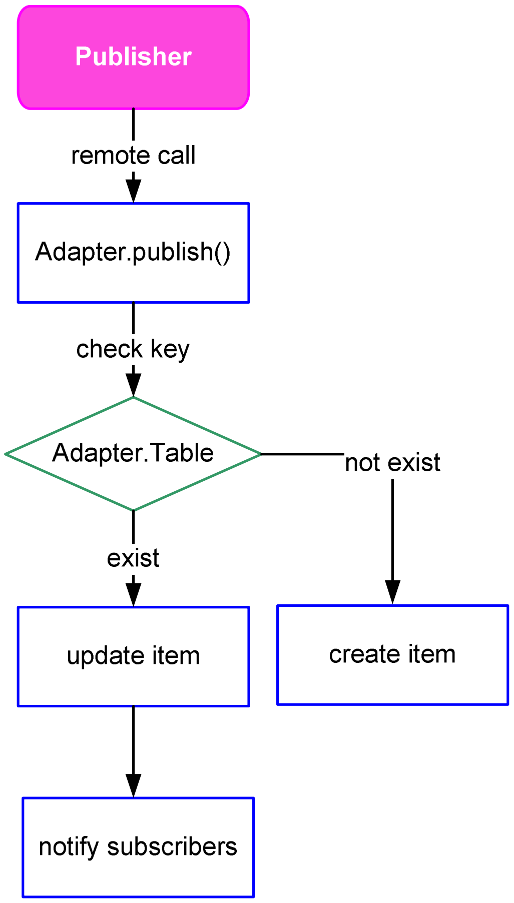
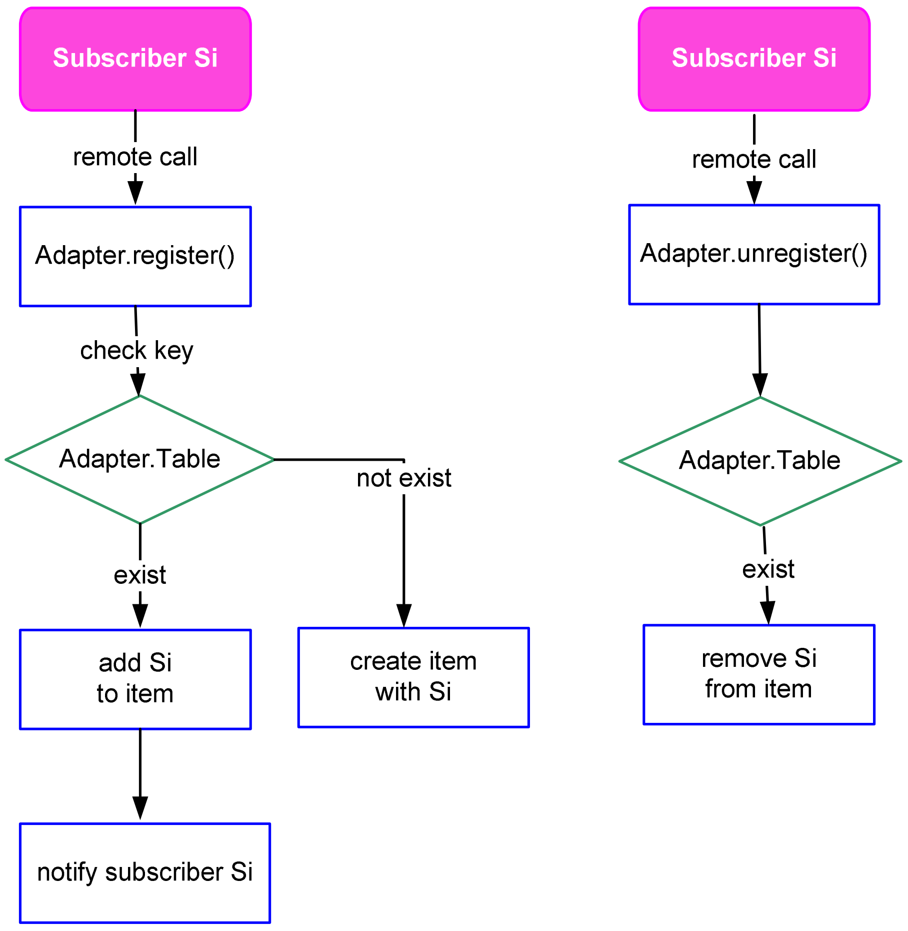


Figure 3.5 Flow chart for a publisher

As shown in Figure 3.5, when *publish*$_i$*(v, t)* method is invoked by process p_i, the adapter checks its Hashtable € whether it contains the item with the key t. If the item x already exists, the adapter updates the item's value v. If not, the adapter creates x with v.

After updating with the value, the adapter notifies to all subscribers £= {S_1 … S_m} who registered themselves to *x*.



(a) register (b)unregister

Figure 3.6 Flow chart for a subscriber

Figure 3.6 shows the way a subscriber registers and unregisters its interests to the adapter. The adapter checks Hashtable € with a key *t* for *register$_i$(v, t)* method which is invoked by a subscriber S_i. If an item *x* with *t* already exists, the adapter adds S_i to *x* and notifies only to the current added S_i who wants to receive *x* at this instant. If not, the adapter creates x with adding S_i to subscription collection £. On the other hand, the adapter removes S_i after *unregister$_i$(v, t)* method is called.

PARMI provides an Item<*V*> class with generic forms which enables type-safety and simple interfaces to create automated data objects. <V> can be any type of object such as Integer, Boolean, Double, String, Arrays, or even Class.

The Item class has three attributes: a topic as a String type, a value as a generic form, and a collection of subscriber' remote references to interested in the topic. Whenever a corresponding event occurs, an item keeps adding or removing three attributes. When notifySubscribers method is locally invoked, it notifies all *subscribe* methods of each subscriber registered. And this mechanism is inspired by the observer-observable design pattern provided by java.util.* libraries in the form of interface Observer and class Observable.

3.3 Application of PARMI with Jacobi Iteration

To provide a parallel grid computing with the Jacobi method, we use *centralized summation algorithm* that is composed of a central *master* and *N workers*[21]. *The master* partitions a matrix into N number, assigns a unique id to each process, keeps the information item which is provided by workers, and collects all convergence rates from each worker. *The workers* are labeled with a unique id which indicates the position of the matrix. Each worker calculates interior points of the matrix, exchanges boundaries of the matrix, and computes the maximum difference between the old matrix and the new matrix. Figure 3.7 and 3.8 shows the detailed Jacobi iteration processes on the RARMI between master and workers.

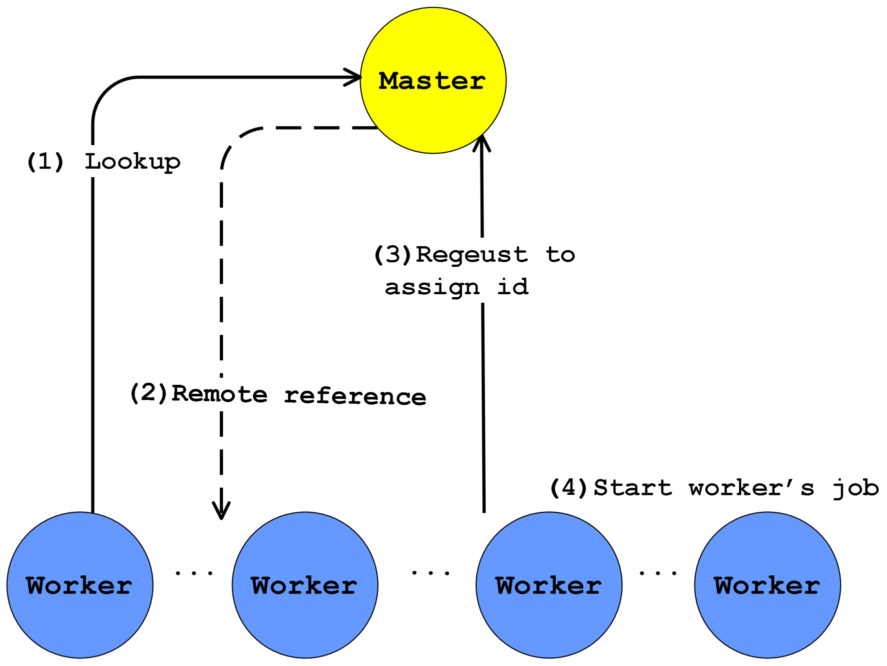


Figure 3.7 The Jacobi iteration processes between a master and workers: part 1

Figure 3.7 shows communications between a master and workers before workers start their own calculation. (1)A master registers itself with the RMI registry on its host name using a unique name, “Jacobi”. A worker requests the remote reference of the master to the RMI registry using lookup service. (2)The master returns the remote reference of itself. Now, each worker is ready to invoke methods of the master. (3)The master assigns ids to all the workers at the same time. If there is a worker that has not requested an id, then the rest of the workers wait until the entire workers request ids. As soon as the last worker requests an id, all the workers are notified and ids are simultaneously assigned. (4) After that, the workers can start their calculating methods.

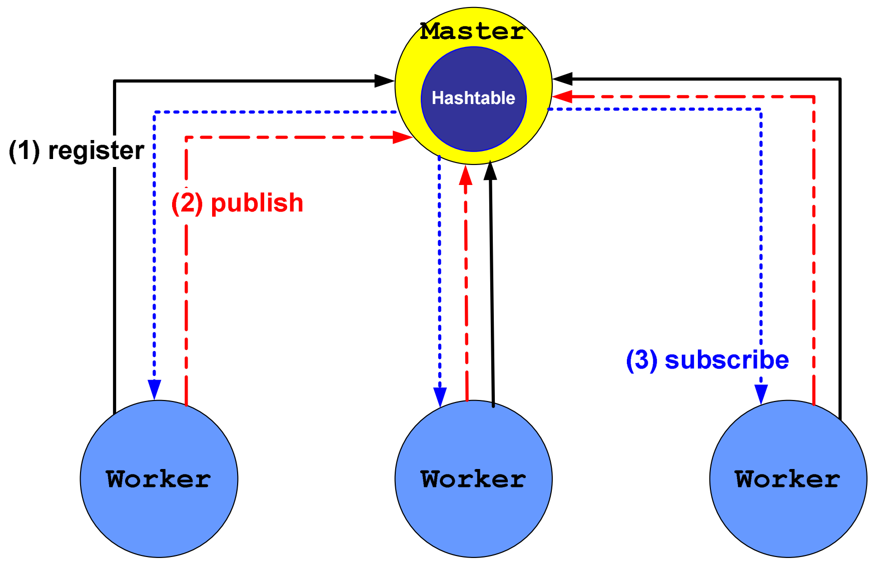


Figure 3.8 The Jacobi iteration processes between a master and workers: part 2

Figure 3.8 illustrates the remaining communications between a master and workers after workers start their own calculation. (1)A worker registers it to the master which topic they are interested in, i.e. the topic is the boundaries of the assigned matrix. And it calculates the two matrix values and computes the maximum difference between two matrixes. (2)A worker publishes the boundary values of the matrix. (3) If a worker registered its topic and that topic's value is published, then it automatically subscribes the topic's value from the master. Because we cannot anticipate the speed of each worker, the information items in the Hashtable are created by register method or publish method. Each worker repeats its process until the maximum difference reaches the convergence rate. When the condition is satisfied, a worker sends its convergence rate to the master and master notifies its decision to stop the worker if all the workers reach the local convergence.

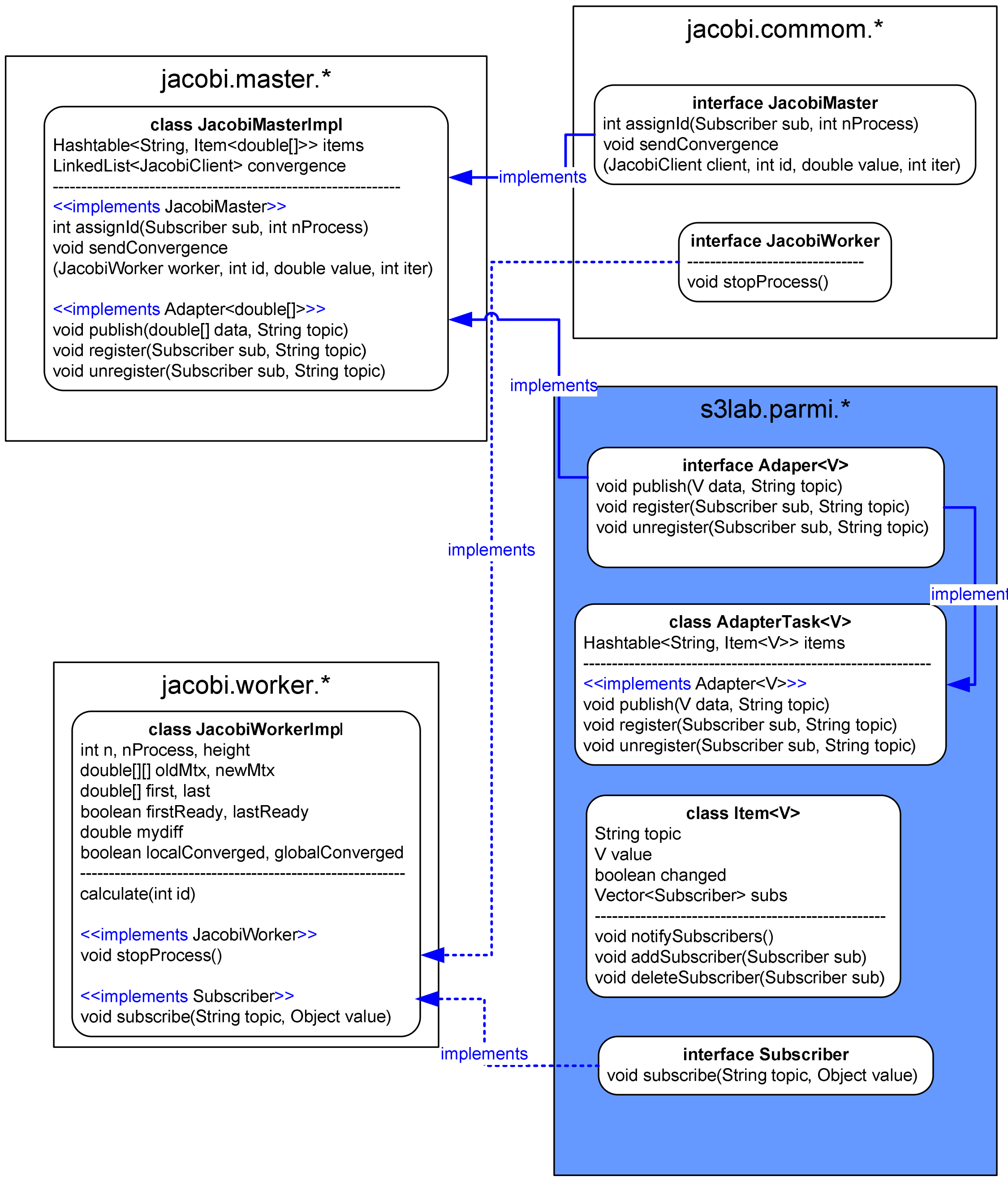
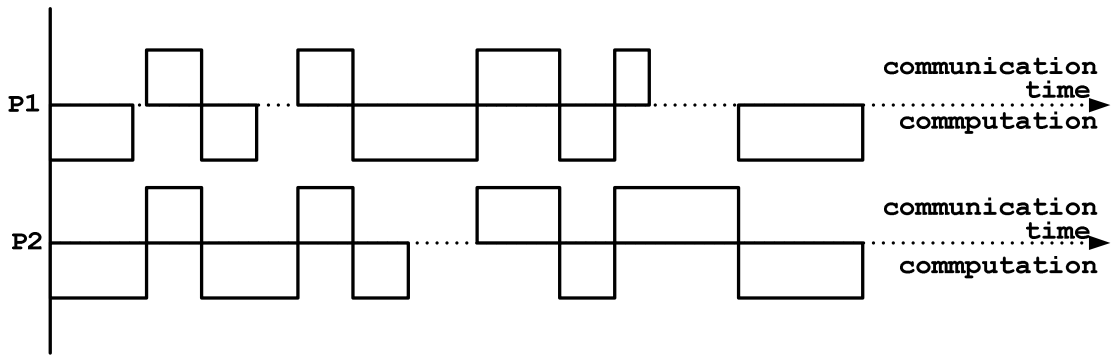
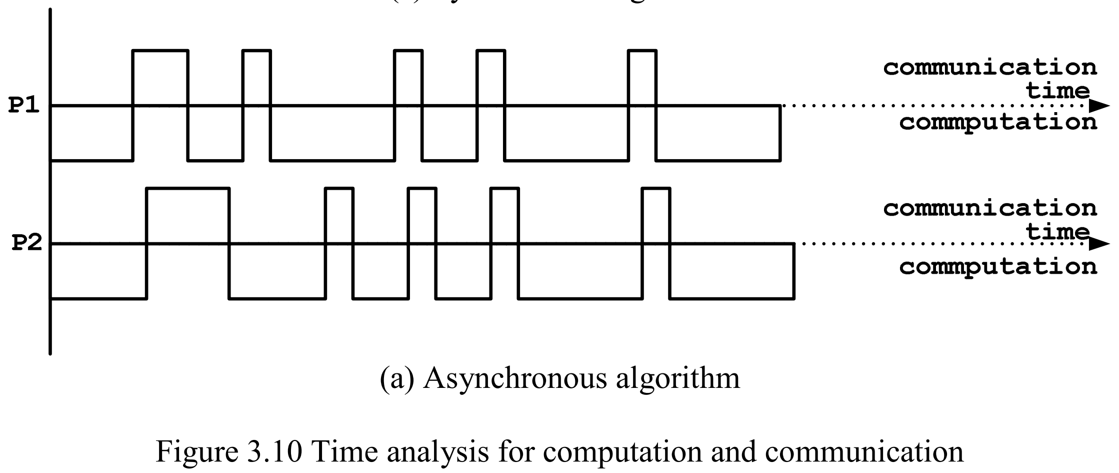

jacobi.master.*
class JacobiMasterImpl
Hashtable<String, Item<double[]>> items
LinkedList<JacobiClient> convergence
<<implements JacobiMaster>>
int assignId(Subscriber sub, int nProcess)
void sendConvergence
(JacobiWorker worker, int id, double value, int iter)
<<implements Adapter<double[]>>>
void publish(double[] data, String topic)
void register(Subscriber sub, String topic)
void unregister(Subscriber sub, String topic)
jacobi.commom.*
interface JacobiMaster
int assignId(Subscriber sub, int nProcess)
void sendConvergence
(JacobiClient client, int id, double value, int iter)
interface JacobiWorker
void stopProcess()
implements
implements
implements
s3lab.parmi.*
interface Adaper<V>
void publish(V data, String topic)
void register(Subscriber sub, String topic)
void unregister(Subscriber sub, String topic)
implements
class AdapterTask<V>
Hashtable<String, Item<V>> items
<<implements Adapter<V>>>
void publish(V data, String topic)
void register(Subscriber sub, String topic)
void unregister(Subscriber sub, String topic)
class Item<V>
String topic
V value
boolean changed
Vector<Subscriber> subs
void notifySubscribers()
void addSubscriber(Subscriber sub)
void deleteSubscriber(Subscriber sub)
interface Subscriber
void subscribe(String topic, Object value)
jacobi.worker.*
class JacobiWorkerImpl
int n, nProcess, height
double[][] oldMtx, newMtx
double[] first, last
boolean firstReady, lastReady
double mydiff
boolean localConverged, globalConverged
calculate(int id)
<<implements JacobiWorker>>
void stopProcess()
<<implements Subscriber>>
void subscribe(String topic, Object value)
implements

Figure 3.9 Implementations of Jacobi application using PARMI framework

Finally, Figure 3.9 is a diagram for the final asynchronous Jacobi implementations. The PARMI framework provides the package, *s3lab.parmi.**. A user needs to implement Adapter and Subscriber interface. For the first time user, we provide AdapterTask class which already implemented three methods in Adapter interface using Item class. For Jacobi iteration, we made three different package, *jacobi.master.**, *jacobi.commnon.**, and *jacobi.worker.** according to the RMI syntax. The master implements Adapter interface and the worker implements Subscriber interface respectively. For remote site calls, the *jacobi.commnon.** package need to be located on both master and worker sides. A worker in jacobi.worker.* package has the roles that registers its interested parts, calculates its own part, publish the edges of its own matrix, and send its id, value, and iteration time to the master node when reaches the convergence rate. A master in jacobi.master* package has the role that divides the whole matrix into the number of worker, allocates the matrix to each worker, collects the convergence rate and notifies its decision to stop worker if all worker reach the convergence rate.



(a) Synchronous algorithm



(a) Asynchronous algorithm

Figure 3.10 Time analysis for computation and communication

As we mentioned in section 2.5.4, we uses asynchronous iterative algorithm in our Jacobi application. Figure 3.10 compares the synchronous and asynchronous iterations and communications to analyze the waiting time and difference of communications. Different from synchronous algorithm, asynchronous one does not has synchronization between two iterations. Therefore, there is no idle time in asynchronous algorithm.

CHAPTER 4

Evaluation

In this chapter, we present the results of our experiment and analysis. We have performed our experiments on two different environments to evaluate the system, because these experiments are restricted to a single-site supercomputer because of the firewall. First, we conducted our experiments on a cluster with up to 50 Linux machines on OSU's Tulsa campus. Secondly, we evaluate our system performance on 1 Unix, 4 Linux, and 15 Windows machines on OSU's Stillwater campus. These results show that PARMI provides good speedup in both cases.

4.1 System Setup

We did not measure the initial response time of each worker. All workers wait until the last one requests its id and completes its connection to a master. We cannot predict the time between workers and a master because each machine has a different time interval between these processes. Due to this unpredictability, we did not include the time before all processes connected to the master to be assigned ids.

Measurements were performed using the *System.currentTimeMillis()* method of Java. We set up the convergence criterion to 0.03, and then we carried out our experiments. When the convergence criterion was set to less than 0.03, one or two workers didn't converge in asynchronous algorithm. This is because even if a sequential iterative algorithm converges with the right solution, its asynchronous parallel counterpart may not converge[5].

We measured the execution time on both sides of a master and workers. The measurements on the master are useful for analyzing the overall performance. Because the majority of methods are invoked by workers, we can extract the communication time from the total execution time and compare the time involved in computation and communication. This comparison provides enough evidence based on the asynchronous applications on PARMI to be suitable for computational science in a grid context.

The synchronous version was implemented using the current existing RMI communication and synchronous iterative algorithm. The asynchronous version was implemented using the PARMI framework and the asynchronous iterative algorithm.

4.1.1 System Configurations for a Local Heterogeneous Cluster

All experiments were conducted on a Rocks cluster with 60 processors. The cluster is combined with a server and 4 racks:

- The server machine is a Dell PowerEdge 4800 with dual Xeon 2.4 GHz and 2 GB memory.
- On the rack 0, there are 9 Pentium-D 3.0 GHz, 4 Pentium4 3.0 GHz, and 1 CeleronD 2.66 GHz machines all with 1 GB memory and 80 GB HDD.
- On the rack 1, we have 7 various Dell(1.5 to 2.4 GHz and 0.25 to 1 GB memory) and 4 CeleronD machines with 2.66 GHz, 1 GB memory, and 80 GB HDD.
- On the rack 2, 12 CeleronD 2.66 GHz machines with 1 GB memory and 80 GB HDD.

- On the rack 3, we have 14 CeleronD 2.66GHz machines with 1 GB memory and 80GB HDD.

We used the server as a master and the 4 racks as workers. The server's hostname is *eaton* and the racks' hostnames start from *compute-0-0* to *compute-3-14*. Most of them have either one or dual processors and only *eaton* has quad processors in it. All of the machines are running on Linux 2.6.9. All experiments were conducted using SUN's JDK version 1.5.0. All machines were connected to each other by GB interconnection

4.1.2 System Configurations for Remote Heterogeneous Machines

Experiments for the master were conducted on a Sun Fire 880 machine whose hostname is *csa*. The configuration of the machines is described as follows:

- The *csa* has quad processors and all processors' speed is 900 MHz. Memory size is 8.19 GB. The machine is running on Sun OS 5.9 and JDK version is 1.5.0.

Experiments for workers were conducted on a cluster of Linux machines and individual 15 Windows machines:

- The cluster is combined with 4 machines. These machines have quad processors each with dual CPU and their hostnames start from *csx0* to *csx3* sequentially. All the processor speeds are 1000 MHz and all the memory sizes are 3.60 GB. All of the machines are running on Linux 2.6.9 and JDK versions are 1.6.0. Therefore, the total number of processors for the cluster is 16.
- For Windows machines, we used 15 machines on the OSU Kerr computer lab. They all have dual processors and all processors' speed is Pentium 4 CPU 3.2 GHz. Their memory size is 1.00 GB running on Windows XP Professional

Version 2002 Service Pack 2 and JDK version is 1.5.0. All machines connected to each other by a standard 100Mb/s Ethernet network.

4.2 Results

4.2.1 The Experiments on a Local Heterogeneous Cluster

We conducted our experiments on a cluster. Basically, all commands were executed on eaton node. After we started a master on the eaton node, we connected to compute-0-0 to compute-3-14 nodes using Secure Shell(SSH). We employed a shell script to establish SSH. Table 4.1 shows the code for the shell script. We assumed that the possible total number of processors for workers is 60 based on the following conditions: Because a node only has 255MB memory, we excluded this node from the total number of processors. After that, we figured out the sum of processor numbers. If the machine had dual processors, then we added 2. Again, we excluded 4 processors from 64 processors because 4 processors existed on the eaton node for master.

```
#!/bin/bash
processN=xxx
i=0

while [ "$i" -lt $processN ]
do
  echo "**start current id:$i"
  (( c = $i % 60 ))
  if [ "$c" -eq 0 ]; then a=0 b=0
  elif [ "$c" -eq 1 ]; then a=0 b=1
  elif [ "$c" -eq 2 ]; then a=0 b=2
  elif [ "$c" -eq 3 ]; then a=0 b=3
  elif [ "$c" -eq 4 ]; then a=0 b=4
  elif [ "$c" -eq 5 ]; then a=0 b=6
  elif [ "$c" -eq 6 ]; then a=0 b=8
  elif [ "$c" -eq 7 ]; then a=0 b=10
  elif [ "$c" -eq 8 ]; then a=0 b=12
  elif [ "$c" -eq 9 ]; then a=0 b=14
  elif [ "$c" -eq 10 ]; then a=0 b=15
  elif [ "$c" -eq 11 ]; then a=0 b=18
  elif [ "$c" -eq 12 ]; then a=0 b=20
  elif [ "$c" -eq 13 ]; then a=0 b=22
  elif [ "$c" -eq 14 ]; then a=1 b=1
  # elif [ "$c" -eq 15 ]; then a=1 b=2//memory is too small
  elif [ "$c" -eq 15 ]; then a=1 b=3
  elif [ "$c" -eq 16 ]; then a=1 b=4
  elif [ "$c" -eq 17 ]; then a=1 b=5
  elif [ "$c" -eq 18 ]; then a=1 b=6
  elif [ "$c" -eq 19 ]; then a=1 b=7
  elif [ "$c" -eq 20 ]; then a=1 b=8
  elif [ "$c" -eq 21 ]; then a=1 b=9
  elif [ "$c" -eq 22 ]; then a=1 b=10
  elif [ "$c" -eq 23 ]; then a=1 b=11
  elif [ "$c" -eq 24 ]; then a=2 b=0
  elif [ "$c" -eq 25 ]; then a=2 b=1
  elif [ "$c" -eq 26 ]; then a=2 b=2
  elif [ "$c" -eq 27 ]; then a=2 b=3
  elif [ "$c" -eq 28 ]; then a=2 b=4
  elif [ "$c" -eq 29 ]; then a=2 b=5
  elif [ "$c" -eq 30 ]; then a=2 b=6
  elif [ "$c" -eq 31 ]; then a=2 b=7
  elif [ "$c" -eq 32 ]; then a=2 b=8
  elif [ "$c" -eq 33 ]; then a=2 b=9
  elif [ "$c" -eq 34 ]; then a=2 b=10
  elif [ "$c" -eq 35 ]; then a=2 b=11
  elif [ "$c" -eq 36 ]; then a=2 b=12
  elif [ "$c" -eq 37 ]; then a=2 b=13
  elif [ "$c" -eq 38 ]; then a=3 b=0
  elif [ "$c" -eq 39 ]; then a=3 b=1
  elif [ "$c" -eq 40 ]; then a=3 b=2
  elif [ "$c" -eq 41 ]; then a=3 b=3
  elif [ "$c" -eq 42 ]; then a=3 b=4
  elif [ "$c" -eq 43 ]; then a=3 b=5
  elif [ "$c" -eq 44 ]; then a=3 b=6
  elif [ "$c" -eq 45 ]; then a=3 b=7
  elif [ "$c" -eq 46 ]; then a=3 b=8
  elif [ "$c" -eq 47 ]; then a=3 b=9
  elif [ "$c" -eq 48 ]; then a=3 b=10
  elif [ "$c" -eq 49 ]; then a=3 b=11
  elif [ "$c" -eq 50 ]; then a=3 b=12
  elif [ "$c" -eq 51 ]; then a=3 b=13
  elif [ "$c" -eq 52 ]; then a=3 b=14
# dual processor start
  elif [ "$i" -eq 53 ]; then a=0 b=6
  elif [ "$i" -eq 54 ]; then a=0 b=8
  elif [ "$i" -eq 55 ]; then a=0 b=10
  elif [ "$i" -eq 56 ]; then a=0 b=12
  elif [ "$i" -eq 57 ]; then a=0 b=14
  elif [ "$i" -eq 58 ]; then a=0 b=18
  elif [ "$i" -eq 59 ]; then a=0 b=20
  elif [ "$i" -eq 60 ]; then a=0 b=22
# dual processor end
  fi
   ssh -T compute-$a-$b <<EOI
    echo " - compute-$a-$b to execute scriptX"
     cd Thesis/asynch_future
```

```
      bash scriptX </dev/null>&result &
      echo " - output of scriptX is redirected to result"
      exit
EOI
   echo "**stop"
   (( i = $i + 1))
done
```

Table 4.1 Shell script controlling SSH shells

Figure 4.1 and Figure 4.2 display the measurement on the master.

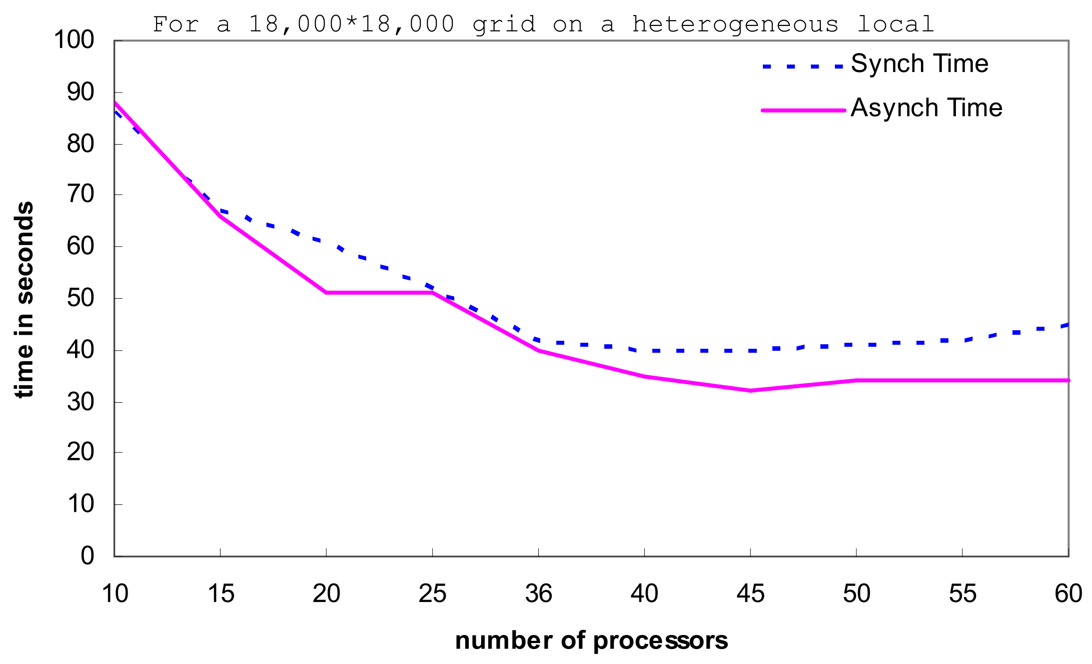


Figure 4.1 Execution cost for sync/async versions

Figure 4.1 shows the execution costs for the synchronous and asynchronous versions in the eaton cluster in order to analyze the overall performance. We can see that the asynchronous version is faster than the synchronous one. This comes from two following reasons: First, in the synchronous algorithm, the fast machines are delayed by the slow ones. Although the number of iterations is greater in the asynchronous case, the fast machines tend to speed up the slow ones[5]. Secondly, asynchronous case reduces the communication time by asynchronous method invocation.

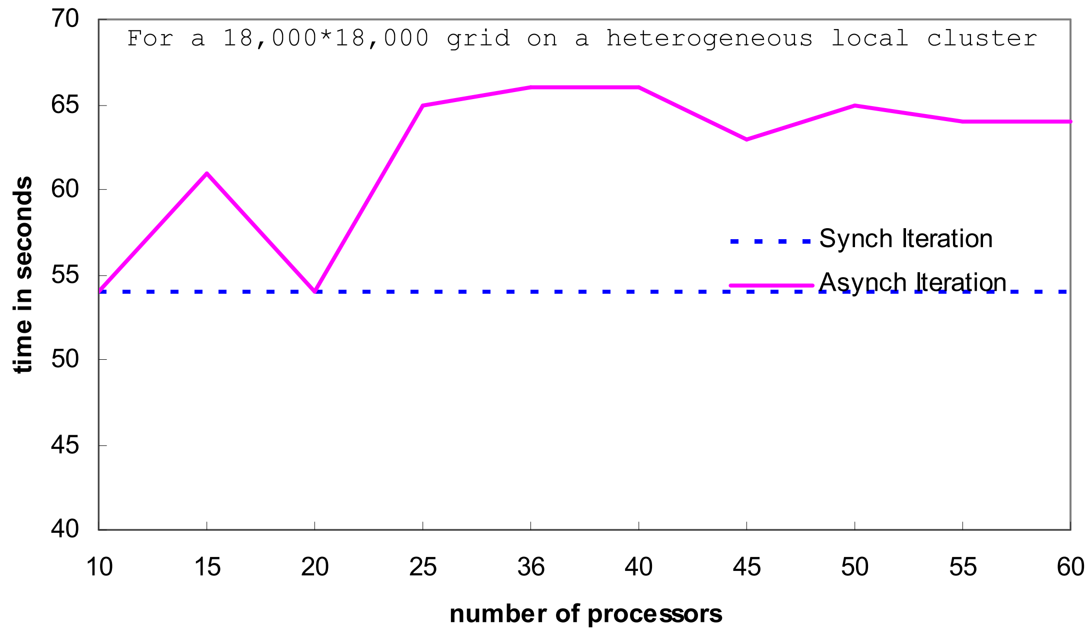


Figure 4.2 The number of iteration for sync/async versions

In Figure 4.2, we can see that the number of iterations of the asynchronous version is irregular. Nevertheless, with 20 processors the number of iterations falls; this is due to the computation time dropped. And this computation time decrease comes from the size of grid decreased as Figure 4.6 shows. The previous studies from other researchers indicates that the number of iterations relative to the asynchronous executions is not significant because each processor computes at its own speed, and the number of iterations can dramatically vary from one processor to another[5].

Figure 4.3 and Figure 4.4 show the time measurement on each worker. For communication time, we measured the time before a worker invoked master's method and after. Then, we calculated the difference between these times. After aggregating the total of computation time, we divided the times with worker numbers. There is the last worker which decides the global convergence. To approach more accurate value for communication time, we took the middle value between the average computation time and the computation time of the last worker. Next, we subtracted communication time from total time to calculate computation time.

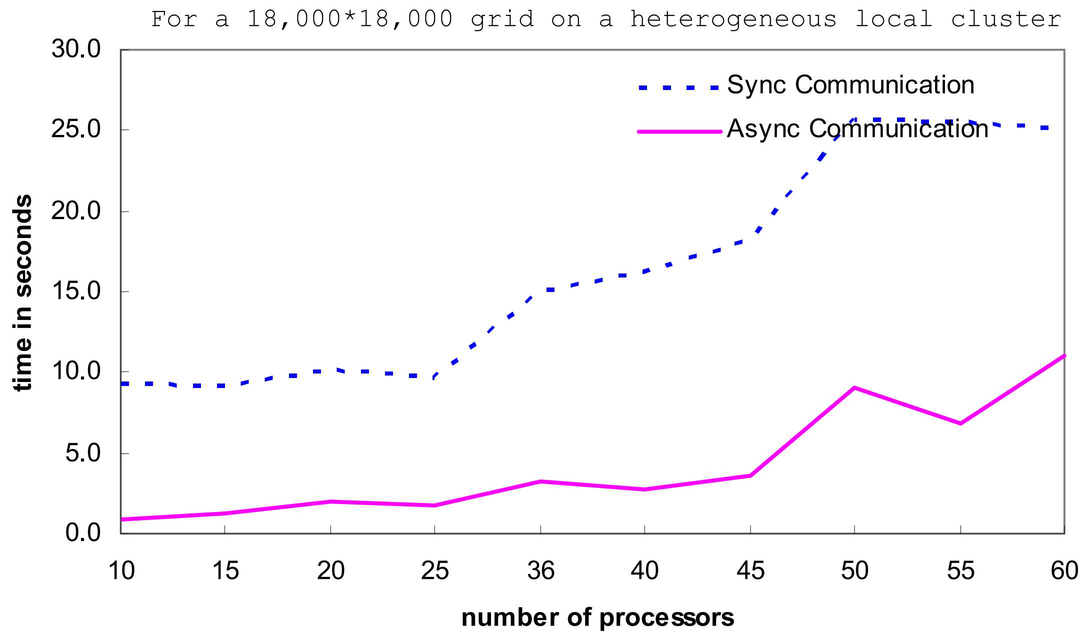


Figure 4.3 Average communication cost for sync/async versions

Figure 4.3 shows asynchronous versions are faster than their synchronous counterparts for communication time. We can see that the asynchronous version achieves better results when the number of processors increases. In the synchronous version, workers become idle until the process of a master are completed and they get the data needed for their next iteration. Furthermore, the convergence detection is done with a gather-scatter operation at each iteration, which takes longer time than a totally asynchronous detection.

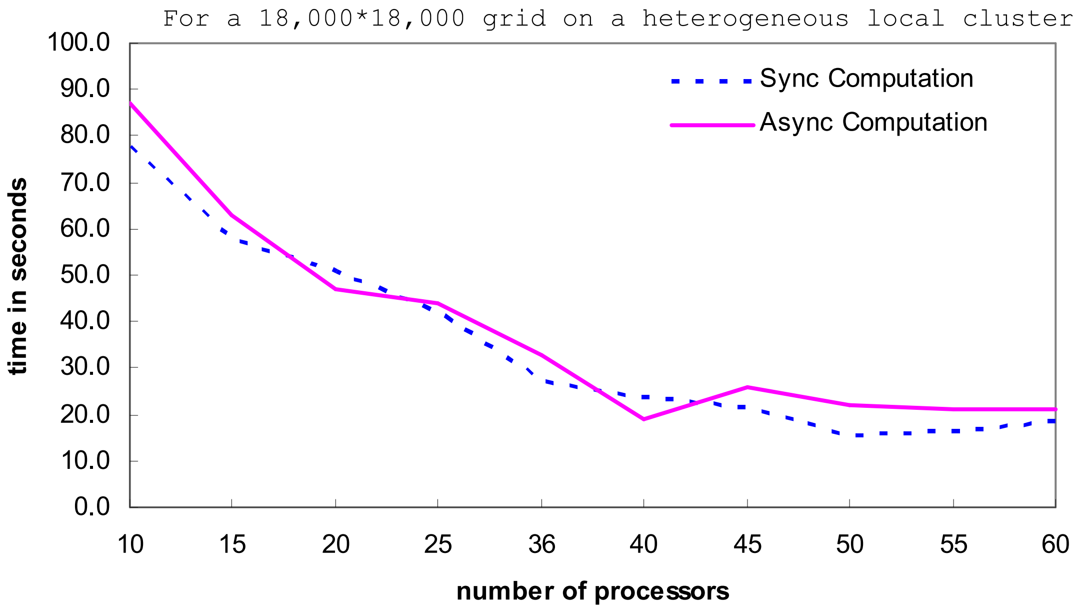


Figure 4.4 Average computation cost for sync/async versions

Figure 4.4 demonstrates that the synchronous version has much less computation time than the asynchronous one because its iteration number is always less than the asynchronous one, as shown in Figure 4.2.

4.2.2 The Experiments on Remote Heterogeneous Machines

Total Processors	Processors(Machines)	
	Linux	Windows
8	4(4)	4(4)
10	4(4)	6(6)
20	16(4)	4(4)
24	16(4)	8(4)
27	16(4)	11(6)
30	16(4)	14(7)
36	16(4)	20(10)

40	16(4)	24(12)
45	16(4)	29(15)

Figure 4.5 The number of processors: Linux versus Windows

In order to demonstrate the practical relevance of PARMI, we have evaluated the performance of synchronous and asynchronous versions using PARMI on the remote heterogeneous machines. A master was simulated on CSA, and workers were executed on Linux CSXs and separate Windows machines, as Figure 4.5 shows.

Figure 4.6 and Figure 4.7 show measurements involved in a master. After the total processors for workers reached at a point, the asynchronous version is faster than the synchronous one. In our experiment, that point was 24, as shown in Figure 4.6. Figure 4.7 displays the iteration numbers for synchronous and asynchronous versions. As in previous experiments, asynchronous versions have an irregular number because of the asynchronous algorithm.

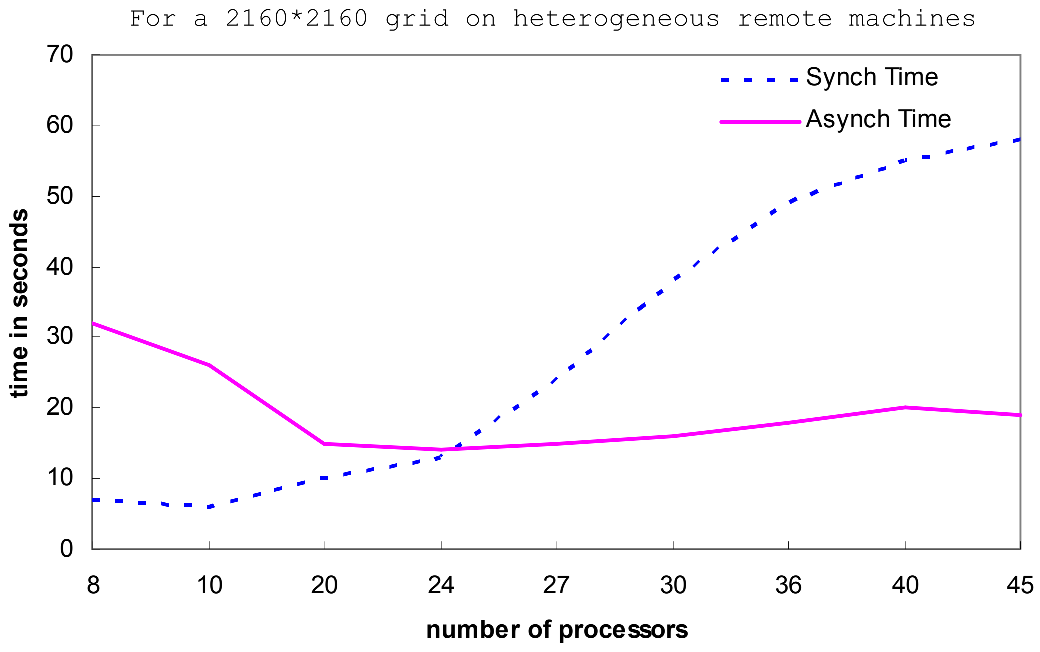


Figure 4.6 Execution cost for sync/async versions

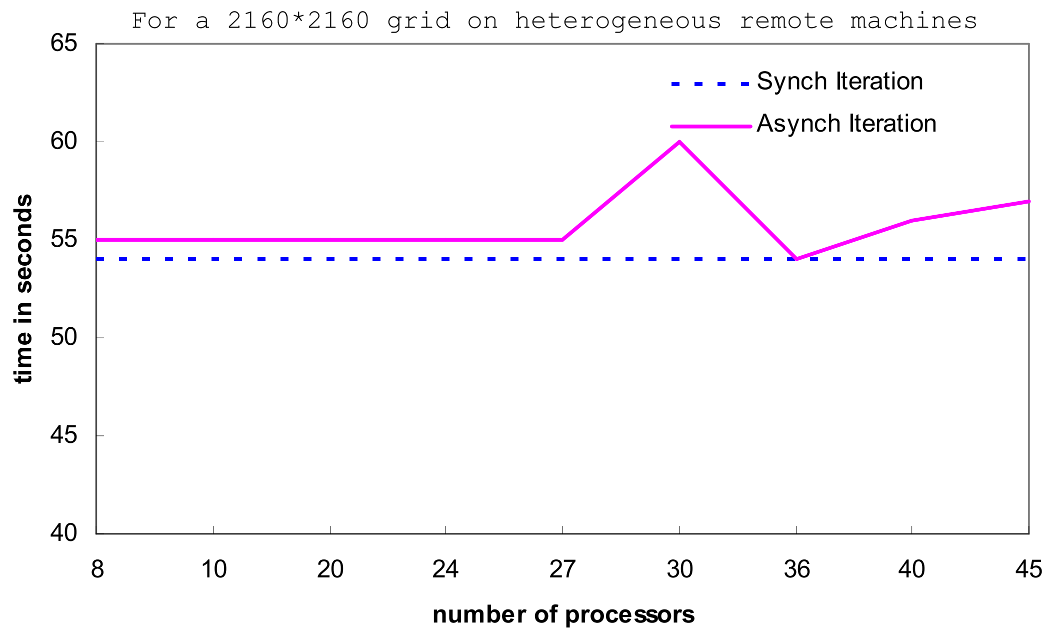


Figure 4.7 The number of iteration for sync/async versions

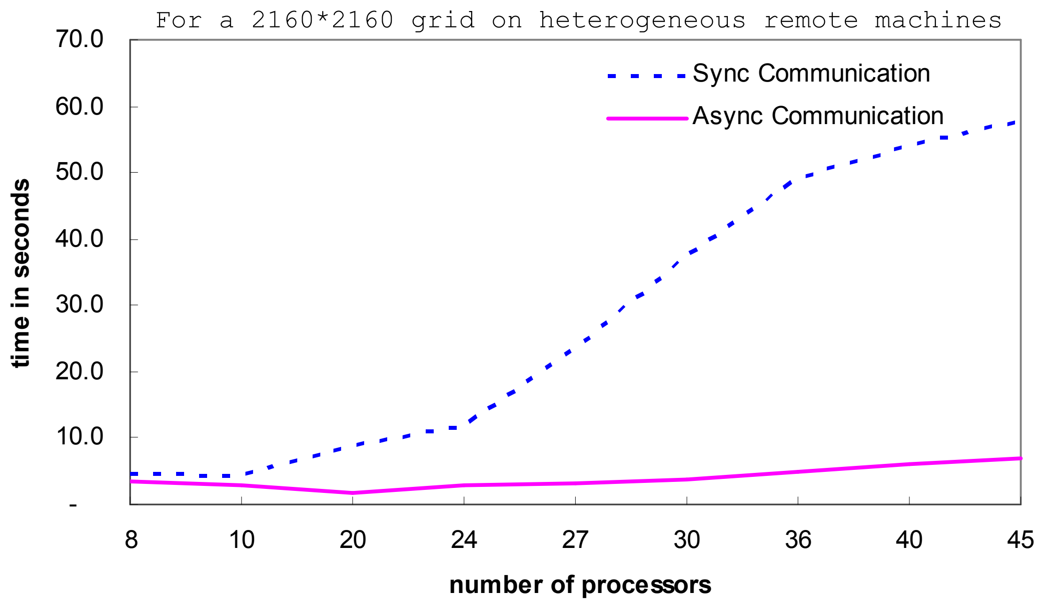


Figure 4.8 Average communication cost for sync/async versions

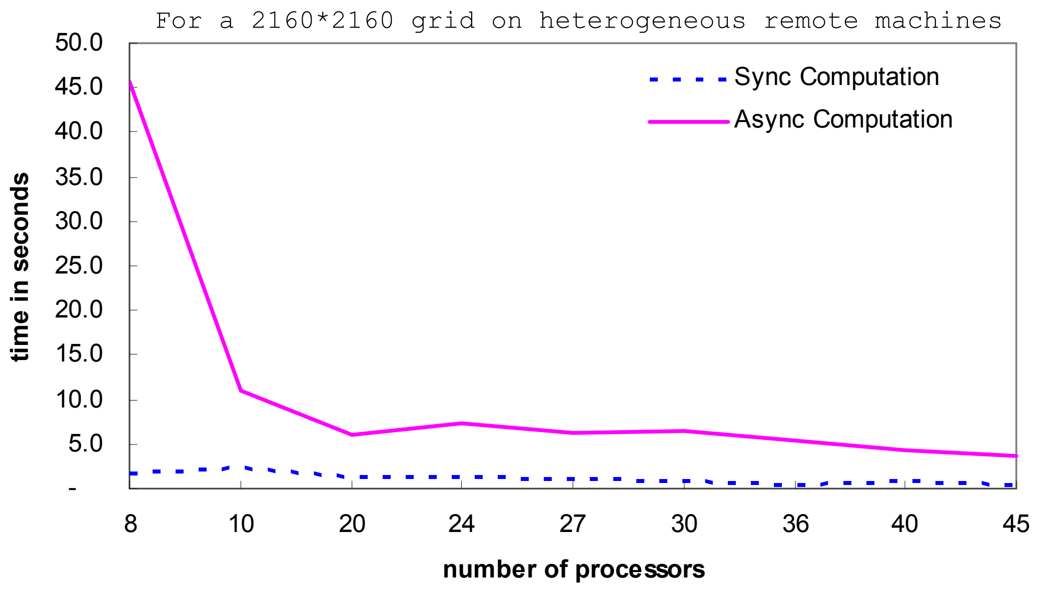


Figure 4.9 Average computation cost for sync/async versions

Figure 4.8 and Figure 4.9 show that asynchronous version has better performance than the synchronous one on communication time. In case of computation time, the synchronous version has less time than the asynchronous one.

4.3 Conclusion

PARMI has excellent overall performance and high throughput. This is mainly caused by the improvement of communication time. We desynchronized the communication using asynchronous algorithms and asynchronous RMI method invocations. These suppress all the idle time and so reduce the whole execution times.

Finally, Table 4.2 summarizes all the results from the previous section. These results demonstrate that the speedup of synchronous versus asynchronous communication improves when a grid size increases and an experiment is conducted on a remote environment. In the experiments, we can see that when the ratio of computation time to communication time increases, the ratio of synchronous time to asynchronous time decreases because computation time is significant when compared to communication times. Hence, for very large problems, it is necessary to involve even more processors in order to reduce computation times and preserve an efficient ratio of synchronous time to asynchronous time[3].

(t): total execution cost, (c): average communication cost.
Speedup is the time on synchronous versus asynchronous versions.

(time in seconds)

N of procs	5	10	20	30	35	42	50	60
Sync(t)	5	5	7	9	11	12	12	16
Async(t)	3	2	4	6	7	7	8	10
Speedup(t)	1.7	2.5	1.8	1.5	1.6	1.7	1.5	1.6
Sync(c)	1.6	4.6	6.4	8.4	10.5	11.7	11.6	15.0
Async(c)	0.4	0.7	1.9	4.4	4.1	5.1	6.6	8.1
Speedup(c)	4.0	6.6	3.4	1.9	2.6	2.3	1.8	1.9

(a) for a 2100*2100 grid on a local heterogeneous cluster

(time in seconds)

N of procs	5	10	16	20	25	40	50	76
Sync(t)	30	17	22	18	15	17	19	28
Async(t)	32	16	19	16	15	15	17	26
Speedup(t)	0.9	1.1	1.2	1.1	1.0	1.1	1.1	1.1
Sync(c)	2.4	2.4	14.2	9.3	7.4	13.6	16.7	23.0
Async(c)	0.2	0.8	0.9	1.3	1.4	6.3	8.6	17.6
Speedup(c)	12.0	3.0	16.2	7.4	5.3	2.2	1.9	1.3

(b) for a 7600* 7600 grid on a local heterogeneous cluster

(time in seconds)

N of procs	8	10	20	24	27	30	36	40	45
Sync(t)	7	6	10	13	24	38	49	55	58
Async(t)	32	26	15	14	15	16	18	20	19
Speedup(t)	0.2	0.2	0.7	0.9	1.6	2.4	2.7	2.8	3.1
Sync(c)	4.8	4.3	9.0	11.9	23.4	37.3	49.3	54.0	58.0
Async(c)	3.5	3.0	1.7	2.9	3.1	3.7	4.9	6.0	7.0
Speedup(c)	1.4	1.4	5.4	4.1	7.6	10.2	10.0	9.0	8.2

(c) for a 2160*2160 grid on remote heterogeneous machines

Table 4.2 Comparisons of speedup using different environment.

CHAPTER 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we have investigated how RMI can be made suitable for dynamic parallel and distributed systems. Our goal was to design a framework that provides highly efficient communication for scientific computing, preferably using communication models that integrate cleanly into Java and are easy to use. For this reason, we have taken the existing RMI model as a starting point in our work.

We have given a description of the RMI model and analyzed the former studies for asynchronous RMI implementations to evaluate their suitability for high-performance parallel programming. This analysis showed that these existing RMI implementations are not efficient enough to fully utilize a high-performance network because of point-to-point and asynchronous communication characters.

To solve this problem, we designed and implemented PARMI, a high-performance RMI framework that is specifically optimized for parallel programming on a heterogeneous cluster computer. To overcome point-to-point and asynchronous communication, we adapted a publish/subscribe communication model. Also we used Generics to provide a flexible and scalable object-oriented typed system.

A scientific application using the Jacobi iteration method has been developed to demonstrate the performance gain using PARMI communication framework compared to using RMI mechanisms. To augment the performance improvement, we chose synchronous and asynchronous iterative algorithms. The synchronous version was implemented using the current existing RMI communication and synchronous iterative algorithm. The asynchronous version was implemented using PARMI framework and asynchronous iterative algorithm. We showed that the asynchronous application using PARMI significantly increases the speedup of parallel applications. We have also showed that the performance improvement is mainly due to communication overhead decrease.

5.2 Future Work

While our framework tries to provide features which will make it possible to implement high performance communication on a heterogeneous remote cluster, it is certainly not a finished product. A lot more simulations and improvements are possible as described in the follows:

- As we mentioned on the starting point of chapter 4, the more simulation on geographically multi-sites environments is helpful to prove the performance enhancement of PARMI framework.
- More applications using PARMI framework needed to demonstrate publish/subscribe communication benefit. While a publish/subscribe communication supports one-to-one, one-to-many, many-to-one, and many-to-many communication between publishers and subscribers, the Jacobi application only need one-to-one communication.
- As a result of our simulations, we found that Jacobi application using PARMI only support 150 workers per a master. If more than 150 workers concurrently

publish their data to a master, then communication overhead of a master increases and some workers cannot get the connection between the master and themselves. By using the middleware, a master distributes its workload in case two many of worker require its answer.

REFERENCES

[1] E. Andersson, *Calculation of Pi Using the Monte Carlo Method*, 2006.

[2] G. R. Andrews, *Foundations of multithreaded, parallel, and distributed programming* Addison-Wesley, Reading, Mass, 2000.

[3] J. Bahi, S. Contassot-Vivier and R. Couturier, *Coupling Dynamic Load Balancing with Asynchronism in Iterative Algorithms on the Computational Grid*, *17th IEEE and ACM int. conf. on International Parallel and Distributed Processing Symposium*, IEEE computer society press, Nice, France, 2003, pp. 40a, 9 pages.

[4] J. M. Bahi, R. Couturier and P. Vuillemin, *Asynchronous Iterative Algorithms for Computational Science on the Grid: Three Case Studies*, *High Performance Computing for Computational Science*, Springer, Valencia, Spain, 2004, pp. 302-314.

[5] J. M. Bahi, R. Couturier and P. Vuillemin, *Solving Nonlinear Wave Equations In The Grid Computing Environment: An Experimental Study*, Journal of Computational Acoustics, 14 (2006), pp. 113-130.

[6] R. Baldoni, M. Contenti, S. T. Piergiovanni and A. Virgillito, *Modeling publish/subscribe communication systems: towards a formal approach*, 2003, pp. 304-311.

[7] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom and D. C. Sturman, *An efficient multicast protocol for content-based publish-subscribe systems*, 1999, pp. 262-272.

[8] B. Barney, *Introduction to Parallel Computing*, 2006.

[9] D. P. Bertsekas and J. N. Tsitsiklis, *Convergence rate and termination of asynchronous iterative algorithms*, *Proceedings of the 3rd international conference on Supercomputing*, ACM Press, Crete, Greece, 1989.

[10] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and distributed computation : numerical methods* Prentice Hall, Englewood Cliffs, N.J. , 1989.

[11] A. D. Birrell and B. J. Nelson, *Implementing Remote procedure calls*, *SOSP '83: Proceedings of the ninth ACM symposium on Operating systems principles*, Bretton Woods, New Hampshire, United States, 1983, pp. 3.

[12] G. Bracha, *Generics in the Java Programming Language*, 2004.

[13] D. Chazan and W. Miranker, *Chaotic relaxation*, Linear Algebra and its Applications, 2 (1969), pp. 199-222.

[14] P. T. Eugster, P. A. Felber, R. Guerraioui and A.-M. Kermarrec, *The Many Faces of Publish/Subscribe*, ACM Computing Surveys, 35 (2003), pp. 114-131.

[15] P. T. Eugster, P. Guerraoui and J. Sventek, *Distributed Asynchronous Collections: Abstractions for Publish/Subsribe Interaction*, *ECOOP*, 2000, pp. 252-276.

[16] P. T. Eugster, P. Guerraoui and J. Sventek, *Type-Based Publish/Subscribe*, Technical Report (2000).

[17] P. T. Eugster and R. Guerraoui, *Distributed Programming with Typed Events*, *IEEE Software*, 2004.

[18] P. T. Eugster, R. Guerraoui and C. H. Damm, *On Objects and Events*, *Object-Oriented Programming, Systems, Languages, and Applications*, 2001, pp. 254-269.

[19] K. E. K. Falkner, P. D. Coddington and M. J. Oudshoorn, *Implementing Asynchronous Remote Method Invocation in Java*, *Technical Report DHPC-072*, 1999.

[20] D. Flanagan, J. Farley, W. Crawford and K. Magnusson, *Java™ Enterprise in a Nutshell: A Desktop Quick Reference*, O'Reilly & Associates, 1999.

[21] I. Foster, *Designing and Building Parallel Programs*, 1996.

[22] L. E. Heindel and V. A. Kasten, *Highly reliable synchronous and asynchronous remote procedure calls*, 1996, pp. 103-107.

[23] M. Hughes, 1972- *Java network programming* Manning, Greenwich 1997.

[24] M. Izatt, P. Chan and T. Brecht, *Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications*, Concurrency: Practice and Experience, 12 (2000), pp. 667-685.

[25] D. Kurzyniec and V. Sunderam, *Semantic aspects of asynchronous RMI: the RMIX approach*, 2004, pp. 157.

[26] D. Lyon, *Asynchronous RMI for CentiJ*, Journal of Object Technology, 3 (2004), pp. 49-64.

[27] D. J. Maassen, *Method Invocation Based Communication Models for Parallel Programming in Java*, Vruhe Universiteit, 2003.

[28] S. Microsystems, *Getting Started Using RMI*, 2003.

[29] B. Oki, M. Pfluegl, A. Siegel and D. Skeen, *The Information Bus: an architecture for extensible distributed systems Proceedings of the fourteenth ACM symposium on Operating systems principles* Asheville, North Carolina, United States 1993, pp. 58-68

[30] R. Raje, J. Williams and M. Boyles, *An asynchronous Remote Method Invocation (ARMI) mechanism for Java*, Concurrency: Practice and Experience, 9 (1997), pp. 1207-1211.

[31] Sun Microsystems, *Java Remote Method Invocation: Distributed computing for Java*, 1994.

[32] Sun Microsystems, *Java™ 2 SDK, Standard Edition Documentation - rmiregistry*, 2001.

[33] Sun Microsystems, *Java™ Remote Method Invocation Specification*, *Remote Method Invocation Specification*, 1996.

[34] Sun Microsystems, *RMI System Overview*, 1997-2003.

[35] T. Sysala and J. Janecek, *Optimizing remote method invocation in Java*, 2002, pp. 29-33.

[36] M. Torgersen, C. P. Hansen, E. Ernst, P. v. d. Ahe, G. Bracha and N. Gafter, *Adding wildcards to the Java programming language*, *Symposium on Applied Computing (SAC2004)*, ACM Press, Nicosia, Cyprus, 2004, pp. 1289-1296.

[37] E. F. Walker, R. Floyd and P. Neves, *Asynchronous remote operation execution in distributed systems*, 1990, pp. 253-259.

[38] Wikipedia, *Publish/subscribe - Wikipedia, the free encyclopedia*, 2006.

[39] Wikipedia, *Scientific computing*, 2007.

[40] A. Wollrath, R. Riggs and J. Waldo, *A Distributed Object Model for the Java System*, *USENIX Conference on Object-Oriented Technologies*, Toronto, Ontario, Canada, 1996.

[41] S. Wong, *Object Oriented Programming Tips and Resources*, 2005.

VITA

Hee Jin Son

Candidate for the Degree of

Master of Science

Thesis: PARMI: A Publish/Subscribe Based Asynchronous RMI Framework

Major Field: Computer Science

Biographical:

Personal Data: Born in Seoul, Korea, the daughter of S. K. Son and J. S. So.

Education: Graduated from Myungji High School, Seoul, Korea in February 1993. Received Bachelor of Art degree in International Trade from Kyung Hee University, Seoul, Korea in February 1998. Completed the requirements for the Master of Science degree with major in Computer Science at Oklahoma State University in May, 2007

Experience:
Worked in Hyundai Marine & Fire Insurance Co. Ltd from 1997 to 2001. Worked as Sr. Programmer in Checkfree from 2001 to 2002. Worked as IT specialist in Aon Korea, 2002. Worked as System designer in ING Life Korea from 2002 to 2003. Worked as Teaching /Research assistant in Oklahoma State University from 2005 to 2006.

Name: Hee Jin Son Date of Degree: May, 2007

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: A Publish/Subscribe-Based Asynchronous RMI Framework

Pages in Study: 75 Candidate for the Degree of Master of Science

Major Field: Computer Science

Scope and Method of Study: This thesis designs a publish/subscribe-based asynchronous RMI Framework (PARMI) residing on different machines over a network. The objectives of this thesis are: (1) Explore the description of the RMI model and analyze the performance of an existing RMI implementation; (2) Study the related programming models for designing asynchronous RMI structure. Introduce the structure and concepts for a new asynchronous way of communication in RMI; (3) Design a new PARMI framework based on publish/subscribe paradigm, realizing asynchronous communication and computation and decoupling objects in space and time; (4) Evaluate the performance of the PARMI framework on the local/remote and homogeneous/heterogeneous environments. An example scientific application based on the Jacobi iteration numerical method is developed. Extensive experimental evaluation on up to 60 processors demonstrates the performance improvement using the PARMI framework.

Findings and Conclusions: In PARMI, we design the adapter module to provide the publish/subscribe communication. The adapter behaves as a storage space for holding the services that offer the event-based services between service providers and service users. It is responsible for collecting subscriptions and forwarding events to service users, holding the mapping information of remote references, method names, input parameters, and results for methods. A scientific application using the Jacobi iteration method has been developed to demonstrate the performance gain using PARMI communication framework compared to using RMI mechanisms. We showed that the asynchronous application using PARMI significantly increases the speedup of parallel applications. We have also showed that the performance improvement is mainly due to communication overhead decrease.

ADVISER'S APPROVAL: Xiaolin Li