Java for High Performance SPMD and Message Passing Programming

by

SUNG-HOON KO

Degree

Dissertation

Submitted in partial fulfillment of the requirements for the

degree of Doctor of Philosophy in Computer Science

in the Graduate School of Syracuse University

May 18, 2000

Approved ______________________

Professor Geoffrey C. Fox

Date ______________________

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The initial growth of interest in Java was driven largely by its role in bringing a new generation of interactive pages to the World Wide Web. Undoubtedly various features of the language—compactness, byte-code portability, security, and so on—make it particularly attractive as an implementation language for applets embedded in Web pages and servlets embedded in servers. But it was always clear that the ambitions of the Java development team go well beyond enhancing the functionality of HTML documents. According to [56]

> "Java is designed to meet the challenges of application development in the context of heterogeneous, network-wide distributed environments. Paramount among these challenges is secure delivery of applications that consume the minimum of system resources, can run on any hardware and software platform, and can be extended dynamically."

Several of these concerns are mirrored in developments in the High Performance Computing world over a number of years. A decade ago the focus of interest in the parallel computing community was on parallel hardware. A parallel computer was typically built from specialized processors integrated through a proprietary high-performance communication switch. If the machine also had to be programmed in a proprietary language, that was an acceptable price for the benefits of using a supercomputer. This attitude was not sustainable as one parallel architecture gave way to another, and the cost of porting software became exorbitant. For several years now, portability across platforms had been a central concern in parallel computing [19, 23, 71, 42, 64].

More fundamentally, the assumption that high performance computing will be done primarily on specialized supercomputers is questioned increasingly. Rapid strides in performance and connectivity of ordinary workstations and PCs make it look equally possible that the future of parallel computing will be on local area networks, or even the Internet [51, 50].

With Java positioned to become a standard programming language on the Internet, and scientific parallel processing edging towards network-based computation, it is natural to ask how these two technologies will interact. How suitable is Java for scientific computing, and do lessons from research in parallel computing have implications for the development of Java?

Popular acclaim aside, there are some reasons to think that Java may be a good language for scientific and parallel programming.

- Java is a descendant of C++. C and C++ are used increasingly in scientific programming:they are already used almost universally by *implementers* of parallel libraries and compilers. In recent years numerous variations on the theme of C++ for parallel computing have appeared. See, for example [11, 27, 38, 41, 58, 96]

- Java omits various features of C and C++ that are considered "difficult"—notably, pointers. Poor compiler analysis has often been blamed on these features. The inference is that Java, like Fortran, may be a suitable source language for highly optimizing compilers(although direct evidence for this belief is still lacking).

- Java comes with builtin multi-threading. On a shared memory platform independent threads may be scheduled on different processors by a suitable runtime. Even without physical shared memory, muliti-threading can be very convenient in explicit message-passing styles of parallel programming [67].

1.1 Research Objectives

1.2 Issues

1.2.1 Approaches to Parallelism in Java

Java already supports concurrency through the thread mechanism and monitor synchronization built into the language. In this dissertation we are interested in truly *parallel* computation, involving multiple CPUs. Such parallelism could be introduced into Java in a number of ways.

It could be achieved through automatic parallelization of sequential code, but it is unclear why this easier for Java than for other languages. Alternatively, the Java virtual machine for a shared memory multiprocessor can schedule the threads of a multi-threaded Java program on different processors. Some success with these approaches in reported in [16]. For computation on a network(or distributed memory computer) realistic options include language extensions or directives akin to HPF, or provision of libraries—class libraries—to support task parallelism or data parallelism.

A popular approach in C++ has been to defer language extensions and concentrate on class library support for parallel programming. The similarities between the two languages suggest this may be a fruitful avenue in Java too. The success of this analogy is by no means automatic, however. Features such as *templates* and user-defined *operator overloading* make the C++ language inherently more customizable than Java. In C++ library-defined types can be used on an identical footing to primitive types—*inline* methods mean they can be almost as efficient as primitive types. Less importantly, but conveniently, new control constructs can often be simulated in C or C++ through use of macros. On grounds, presumably, of simplicity and transparency many of these features have been omitted from Java.

Such caveats notwithstanding, this dissertation will concentrate on class libraries rather than language extensions. We have been working with class libraries implemented in the standard Java development environment.

1.2.2 Running a Parallel Java Program

Another open question is how multiple interacting Java tasks are to be initiated. Conventionally in network computing, instructions to execute a particular user task are sent to a specialized daemon or secure server running on the target host. Alternatively standard operating system daemons (*rshd*, *rexecd*, ...) can be used to the same effect. Java brings some options of its own. One is to use standard or enhanced Web servers to execute Java byte-codes. This approach is predicated on the existence of cooperative servers, probably running suitable CGI scripts. Another option is to exploit Java-enabled browsers. Applets downloaded from a particular server can perform computations and return the results to that server. In any case the substantive improvement over conventional network computing is that the byte-codes can be downloaded on the fly to target hosts, even in heterogenous networks. Conventionally, it was often necessary to install and compile an application on each target host before starting the task remotely.

1.2.3 Communication in Java

The standard Java API provides a simplified interface to Internet sockets and RMI, a Java version of Remote Procedure Call (RPC).

The socket interface hides much of the ugly detail involved in socket-programming at the traditional C/Unix level. The *java.net* interfaces provides less flexibility than using the system calls directly. On the other hand, Java's built-in support for threads adds some flexibility in scheduling communications that is missing from raw C.

We will give an example of socket programming in Section 2.3.1, but traditionally this has not been a popular paradigm in the parallel processing world. Scientific programmers have expected to program inter-process communication at a higher level, if at all. More successful schemes include

- Message-passing through language-level support [67, 45] or higher-level library interfaces [71].
- Data parallelism, which we take to mean the style of programming in which parallelism is achieved through operations on distributed arrays, with synchronization typically limited to bulk synchronization occurring naturally through collective array operations.
- Communication through shared memory or shared objects, involving some more or less intricate mechanism for inter-process synchronization.

As observed in the previous section, communication through true shared memory is already implicit in the Java thread model. Communication through remote objects(i.e RMI) is undoubtedly a natural and important paradigm in Java, especially for access to remote services. However, RMI is too restrictive and its overhead for dealing with network problems is too verbose for the cluster setting. It suffers from a significant startup time and marshaling/unmarshaling of data is more costly than in the socket method.

An orthogonal issue is whether the high-level libraries used to implement these paradigms (presuming class-library implementations) should be written in Java on top of the standard API, or whether they should be implemented as direct interfaces to native code.

1.3 Organization of The Thesis

The outline of this thesis is as follows. Chapter 2 provides a general overview of the Java Grande Forum, its activities, and the process. We also discuss Java for scientific parallel programming Issues and look at related work of Java Grande in Chapter 2. Chapter 3 introduce the design HPJava framework and its core features. HPJava is a programming language extended from Java to support parallel programming style called *structured SPMD programming*. In Chapter 4, we discuss the design of the `mpiJava` API, an object-oriented Java interface to MPI, and the issues associated with its development. Chapter 5 describes use of the Java object serialization model for marshalling communication data in Java bindings of the MPI. Chapter 6 presents applications and analyzes the performance of Ising model simulation. Finally, Chapter 7 contains conclusion and the directions for future work.

Chapter 2

Java for Scientific Parallel Programming

In this chapter we discuss a general overview of the Java Grande Forum, its activities, and the process. Java Grande Forum is addressing many of the issues that arise in using Java for High Performance Computing and Grande Applications. In the following sections we present the outline various options for parallel programming in Java–possible ways to express parallelism, and ways to handle inter–process communication. Finally we look at related work of Java Grande.

2.1 Java Grande Forum

The attractiveness of Java for scientific computing is being encouraged by bodies like Java Grande [4]. The first meeting of the Java Grande Forum was held at Java '98 on February 28 and March 1, 1998, in Palo Alto, California. The Java Grande forum is motivated by the notion that Java has the potential to be a better language than previous languages such as C and Fortran for developing large science and Grande applications, particularly on distributed and high-performance computers.

A Grande Application is an application of large scale nature, potentially requiring any combination of computers, networks, I/O, and memory. For examples, financial modeling, aircraft simulation, climate and weather, satellite image processing and earthquake prediction.

The major goal of the Java Grande Forum is to develop community consensus and recommendations for either changes to Java or establishment of standards for Grande libraries and services.

The forum is open and operates with a mix of small working groups and public dissemination and request for comments on its recommendations. Currently, two working groups are exist: Numeric Working Group and Concurrency/Applications Working Group.

The goal of the Numeric Working Group is to assess the suitability of Java for numerical computing, and to work towards community consensus on actions which can be taken to improve the language and its environment for applications with significant requirements for floating-point computation. In its initial assessment of Java, the working group has focused on five critical areas where improvements to the Java language are needed:

- floating-point arithmetic
- complex arithmetic
- multidimensional arrays
- lightweight classes
- operator overloading

Lightweight classes and operator overloading provide key components to proposed improvements for complex arithmetic and multidimensional arrays. Information about the Numeric Working Group activities is maintained at the JavaNumeric web page at http://math.nist.gov/javanumerics.

The goal of the Concurrency/Applications Working Group is to assess the suitability of Java for parallel and distributed computing and so-called problem-solving environments(applications) and to work towards community consensus on actions which can be taken to improve the language and environment. The focus has been on the Java-specific frameworks designed to support concurrent and high-performance computing methodology. The working group is organized as follows:

- **critical JDK issues**
  highest priority issues, mostly related to performance of RMI and Object Serialization
- **benchmarking JVM performance**
- **seamless(desktop) access to high-performance computers(computing portals)**
- **other parallel and distributed computing issues**
  the availability of MPI-like message passing facilities for programming networks of computers

Information about the Java Grande Concurrency and Applications Working Group activities can be found at http://www.javagrande.org and following the direct link to Working Groups.

2.2 Data Parallelism in Java

The most comprehensive statement of the data parallel model of computation is the High Performance Fortran standard [42, 64]. That document is supposed to embody much of the collective experience of the scientific parallel programming community. Presumably, then, any attempt to incorporate data parallelism into Java should build on the HPF model wherever possible.

The HPF definition consists of a large set of directives that can be used to annotate a standard Fortran program, a small handful of langauge extension, and a library of new functions for operating on arrays. An initial data-parallel Java may well be implemented through a class-library. This library would assume the roles of the directives and language extensions in HPF as well as the HPF library.

We loosely distinguish two different levels at which a library implementation of the HPF semantics can operate.

- The first is the level of the so-called *run-time libraries* [?, 29, 33, 26]. This kind of library provides functions for scheduling and executing specific patterns of collective communication already identified by a compiler (in the HPF case) or else by an application programmer using the library directly. Such a library may also provide functions for translating between global subscripts and local, node-level subscripts–i.e, for computing the mapping of a distributed array into the address spaces of individual processors.

- Alternatively, a library can operate at a higher level that conceals all aepects of data localization and transfer from the user. The only responsibility of the user is to specify the distribution format od arrays when they are declared. subsequently the user just tells the library to do particular operations on particular distributed arrays. It is left to the library to work out whether or not a communication is implied. In effect the library is operating at the same level as the HPF language. An example of such a library is A++/P++ [80]

In either case a *class library* version is likely to include classes to describe the elements of the HPF data model, such as processor arrangement and the distributed arrays themselves.

2.2.1 Parallel arrays and collective communication

At the run-time level, a class library implementation of the HPF model is likely to include

- Classes to describe process arrays and distributed data arrays.
- Classes or functions to simplify access to locally held elements of a distributed array (including parallel iteration).
- Functions for collective communication through operations on distributed arrays: regular "copying" operations including shifts and transposes, arithmetic reduction operations, irregular gather/scatter operations, and so on.

Our first experiments with a Java binding only touch the surface of the full HPF semantics, but they provide some hints about a general framework. The interface given here borrows from the C++ class library, Adlib [26]. We will present an overview of Adlib in more detail in Section **??**.

A distributed array is parametrized by a member of the *Array* class. In C++ *Array* would naturally be a *template* for a *container class*. In Java, generic container classes are problematic. Without the template mechanism, the obvious options are that a container holds items of type *Object*, the base class for all non-primitive types, or that a separate container class is provided for each allowed type of element. The first option doesn't allow for array elements of primitive type, and prevents compile-time type-checking (reminiscent of using *void** in C). The second approach presumably involves restricting elements to the finite set of primitive types (*int*, *float*, ...)[1]. For now we have side-stepped the issue by leaving the data elements

[1]Perhaps a good compromise is to provide one container class for each primitive type and one for *Object*.

out of the *Array* class. *Array* defines the shape and distribution of an array, but space for elements is allocated in a separately declared vector of the appropriate type[2].

The constructor for an *Array* defines its shape and distribution format. This is expressed through two auxilliary classes: the *Procs* and *Range* classes. The *Procs* class corresponds directly to the HPF *processor arrangement*. It maps the set of physical processes on which the program is executing to a multi-dimensional grid. A *Range* describes a single dimension of an HPF array. It embodies an array extent (the size of the array in the dimension concerned), and a mapping of the subscript range to a dimension of a *Procs* grid.

2.3 Message Passing in Java

A distributed memory system consists of processors and their local memories connected by interconnection network. Since there is no global memory, it is necessary to move data from one local memory to another by means of message-passing. This is typically done through *send/receive* API calls which must be written into the application program. Message-passing remains one of the most effective and widely used communication paradigms in parallel computing. In this section we compare two approaches to message-passing in Java, in the context of a scientific application. The first approach is to use the socket interface in the standard Java API. The second is to work through a Java interface to the message-passing standard, MPI [71].

To minimize distracting details, our application will be elementary: Conway's Life automaton.

[2]Confusingly enough, this makes our *Array* more akin to an HPF *template* than an HPF array. Needless to say, there is no connection between C++ templates and HPF templates.

2.3.1 Java sockets

The Java socket interface `java.net` provides two kinds of socket classes, the streams-based `Socket` class and the datagrams-based `DatagramSocket` class. A datagram is a low-level networking interface and uses a protocol called UDP(Unreliable Datagram Protocol) to tramsmit data. The `Socket` class allows programmers to connect to a specified port and read and write data using the `InputStream` and `OutputStream` of the `java.io` package. The `DatagramSocket` class uses `DatagramPacket` for data transmission and reception. Here the application creates a datagram packet with the source and destination addresses and the relevant data and uses `send()` and `receive()` to transmit data over the network.

The socket model is most suitable for programming client-server applications. Typical scientific parallel programs do not fit directly into this model. Before a SPMD program can start two conditions must obtain: a pool of symmetric peer processes must have been created, and each peer must be able to address a message to any other. This situation is typically bootstrapped as follows. The program is invoked on one host. This host creates a listening socket. It sends instructions to servers on some other hosts (either explicitly or through a command such as *rsh*) to start remote invocations of the program (somehow sending the number of the port on which it is listening in the initiation message). Each new process creates a listening socket and sends its port number to the original process. The original process broadcasts these address to all its slaves. At this point each process knows the port number on which each of its peers is listening. Either they establish all-to-all connections now, before entering the SPMD main program, or the main program starts immediately, and connections are established dynamically when a message needs to be sent.

Figures 2.1, 2.2 give a schematic outline of a distributed Life program using *java.net*. The fairly intricate code sketched above for initialization and establishment

```
class life_java {
  static public void main(String[] args) throws Exception {
    hpj HPJava = new hpj(args);

    int np = HPJava.num_processor();
    int id = HPJava.my_id();

    ... compute local ‘blockSize’, ‘blockBase’ (avoiding empty blocks).

    // ‘block’ has ‘blockSize + 2’ columns.  This allows for ghost cells.

    byte block[][] = new byte[blockSize+2][N];

    ... initialize local block with some pattern

    // Main update loop.

    int next = (id + 1) % np;
    int prev = (id + np - 1) % np;

    for(int iter = 0 ; iter < NITER ; iter++) {

      // Shift this block’s upper edge into next neighbour’s lower ghost edge

      HPJava.Output(next).write(block[blockSize]);
      HPJava.Output(next).flush();
      HPJava.Input(prev).read(block[0]);

      // Shift this block’s lower edge into prev neighbour’s upper ghost edge

      HPJava.Output(prev).write(block[1]);
      HPJava.Output(prev).flush();
      HPJava.Input(next).read(block[blockSize+1]);

      ... Calculate a block of neighbour sums.
      ... Update block of board values.
    }
  }
}
```

Figure 2.1: Skeleton of socket-based Life program.

```
int sums[][] = new int[blockSize][N];

....

  //  Calculate block of neighbour sums.

  for(i = 0, ib = 1 ; i < blockSize ; i++, ib++)
    for(y = 0 ; y < N ; y++) {
      int y_n = (y+N-1) % N ;
      int y_p = (y+1) % N ;
      sums[i][y] =
          block[ib - 1][y_n] + block[ib - 1][y] + block[ib - 1][y_p] +
          block[ib]    [y_n] +                    block[ib]    [y_p] +
          block[ib + 1][y_n] + block[ib + 1][y] + block[ib + 1][y_p];
    }

  // Update block of board values.

  for(i = 0, ib = 1 ; i < blockSize ; i++, ib++)
    for(y = 0 ; y < N ; y++)
      switch (sums[i][y]) {
        case 2 : break;
        case 3 : block[ib][y] = 1; break;
        default: block[ib][y] = 0; break;
      }
```

Figure 2.2: Socket-based Life program (detail).

of socket connections has been absorbed into the definition of an auxilliary class *hpj*. Its interface is

```
class hpj {
  hpj(String[] args) ;

  int my_id() ;
  int num_processor() ;

  DataInputStream Input(int id) ;
  DataOutputStream Output(int id) ;
  ...
}
```

The members *Input* and *Output* return streams associated with sockets connected to peer processes. In the example an N by N Life board is divided blockwise in one dimension, each processor holding a local block of width *blockSize*.

We note

- Initialization is a complex procedure and clearly it should not be coded anew for each application program.

- In this example the messages were contiguous byte vectors that could be transmitted efficiently through the *read* and *write* methods of the Java socket API. In general the messages will have more complex types and the data may not be contiguous in memory. Using the typed primitives of the standard API may then incur extra costs of copying and type-conversion.

For reasons such as these we suspect that direct socket programming will remain unattractive to scientific parallel programmers, even with the simplified Java socket API.

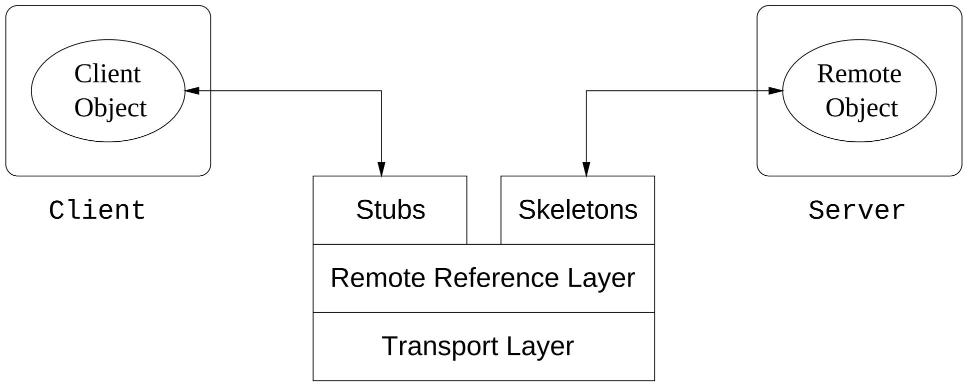

Figure 2.3: RMI Architecture.

2.3.2 Java RMI

Java Remote Method Invocation(RMI) which is an object-oriented version of Remote Procedure Call(RPC), has been included with the Java Development Kit release 1.1. RMI provides an simple and powerful Java-to-Java communication model for invoking member functions on objects that exist in other Java virtual machines exactly as if it were local objects running in the same virtual machines. As most distributed processing is conceived, RMI is based on the Client-Server model. An overview of the Java RMI model is shown in figure 2.3.

The RMI architecture consists of three independent layers between the application program and the Java virtual machine. Each layer is independent and can be replaced by an alternate implementation. The three layers are the Stub and Skeleton Layer, the Remote Reference Layer, and the Transport Layer.

The respective applications reside on both the client and server machines. The client and server communicate with the RMI system through a stub and skeleton respectively. These methods reside on top of the Remote Reference Layer and Transport Layer, allowing the RMI system to handle the interface. When a single request occurs, it travels down through the layers on one computer and up through the layers at the other end.

The stub and skeleton layer is responsible for marshaling and unmarshaling the data and transmitting and receiving them to/from the Remote Reference Layer. The data are marshaled and unmarshaled using a parameter-passing mechanism called *object serialization*. Object Serialization is an essential feature needed by Java's RMI implementation. It provides a program the ability to read or write a whole object to and from a raw byte stream. Object Serialization allows Java objects and primitives and to be encoded into a byte stream suitable for streaming to some type of network or to a file-system. In order to provide this functionality, an object must implement the Serializable interface.

The Remote Reference Layer is responsible for carrying out the invocation. It can be behave differently depending on the parameters passed by the calling program. For example, this layer can determine whether the request is to call a single remote service or multiple remote programs as in a multi-cast.

The Transport Layer is responsible for setting up connections, managing request, monitoring them and listening for incoming calls.

High performance distributed computing can be done with Java RMI, but this has a few drawback since RMI is designed for client-server programming in Web based systems over slow networks. In parallel programming, fast RMI with low latency, high bandwidth, and better reliability are required. A better serialization is also needed for a fast RMI, since Java's object serialization often takes 25%-50% of the time [82] needed for a remote invocation.

2.3.3 Message Passing Libraries in Java

The most popular message-passing libraries are PVM and MPI. Several projects are under way to make message-passing packages to support programs written in Java. Current message-passing in Java can be classified into two categories. First category consists of Java based implementations that use Java as wrapper for existing message-passing libraries that are called through the Java native interface (JNI). The use of native methods achieves better performance and allows programmers to reuse existing codes. However, it restricts jobs to run only on system where both Java and corresponding message-passing library are supported.

The second category of Java based implementations use only pure Java libraries. Typically, this case the underlying communication libraries are Java's RMI or socket. The pure Java implementation obtains high degree of code portability among platforms but relatively lower performance. The performance problem is expected to be overcome with just-in-time (JIT) compilation and other compiler technologies.

In the next section we will discuss Java interface to two popular message-passing standard, PVM and MPI.

2.3.3.1 Parallel Virtual Machine (PVM) in Java

PVM developed at Oak Ridge National Laboratory is a message-passing system that permits a heterogeneous collection of networked computing systems to be used as a single large parallel machine. Thus large computational problems can be solved more cost effectively by using the aggregate power and memory of many computers. PVM consists of two parts: a daemon process, *pvmd*, that any user can install on a machine, and a user library that contains routines for initiating processes on other machine. PVM provides a rich set of dynamic resource management and process control functions (eg., load balancing). It also allows the development of fault-tolerant applications with host or task failure and has good interoperability between heterogeneous hosts. PVM supports programs written in C, C++, and Fortran. Currently there are two groups working on making PVM support programs written in Java.

- **Java-to-PVM interface**

  JavaPVM (or jPVM) [1] is an interface written using the Java native methods capability which allows Java applications to use the Parallel Virtual Machine (PVM) software. Since it is using native method, it limits cross-platform portability.

- **Pure Java implementation of PVM**

  JPVM [40] is a pure Java implementation of PVM. Therefore unlike JavaPVM, it is better matched Java programming styles and the standardization of the JVM makes the JPVM code much simpler to maintain across heterogeneous machine. The performance is poor compared to JavaPVM. The detail comparison of communication benchmark of native PVM, JavaPVM, and JPVM is reported in [98].

2.3.3.2 Message-Passing Interface(MPI) in Java

MPI [71, 59] is a library of functions and macros that can be used in C, FORTRAN, and C++ programs. MPI is intended for use in programs that exploit the existence of multiple processors by message-passing. MPI was developed by a group of researchers from academia, industry, and government. As such, it is one of the first standards for programming parallel processors, and it is the first that is based on message-passing. MPI provides a rich set of communication libraries, application topology and user defined data types. The first phase of MPI (MPI 1) was focused on developing a message-passing standard. The second phase of MPI effort (MPI 2) is underway adding parallel I/O, fault tolerance, language binding for C++ and additional communications. Currently there are several known efforts towards the design of early MPI interfaces for Java.

- **Automatic generation of Java wrappers to MPI**

  An approach comparable to our work to creating full Java MPI interfaces has been taken by University of Westminster [73, 54]. In their work Java wrappers were automatically generated from the C MPI header using Java-to-C interface generator (JCI). This eases the implementation work, but does not lead to a fully object-oriented API.

- **Pure Java implementations of MPI**

  MPIJ [2] is a pure Java-based subset of MPI developed at Brigham Young University as part of the Distributed Object Group Meta-computing Architecture (DOGMA) [63] system. MPIJ implements a large subset of MPI functionality including point-to-point commnunication, intra-communicator operations, groups, user-defined reduction operations. MPIJ communications uses native marshaling of primitive Java types. This technique allows MPIJ to achieve communication slpeds comparable to, and frequently exceeding that, of native MPI implementations.

  jmpi [36, 37] is a pure Java implementation of MPI built on top of JPVM. Due to additional wrapper layer to JPVM routines, its performance is poor compared to JavaPVM.

  JMPI [62] is a commercial effort underway at MPI Software Technology, Inc. to develop a message-passing framework and parallel support environment for Java. It targets to build a pure Java version of MPI-2 standard specialized for commercial applications.

- **Java Grande Forum's MPI-like API**

  MPJ [13] is the Java Grande Forum's MPI-like message-passing API focused on both Java wrappera and a pure-Java versions. The aim is to provide an implementation of MPJ that is maximally portable.

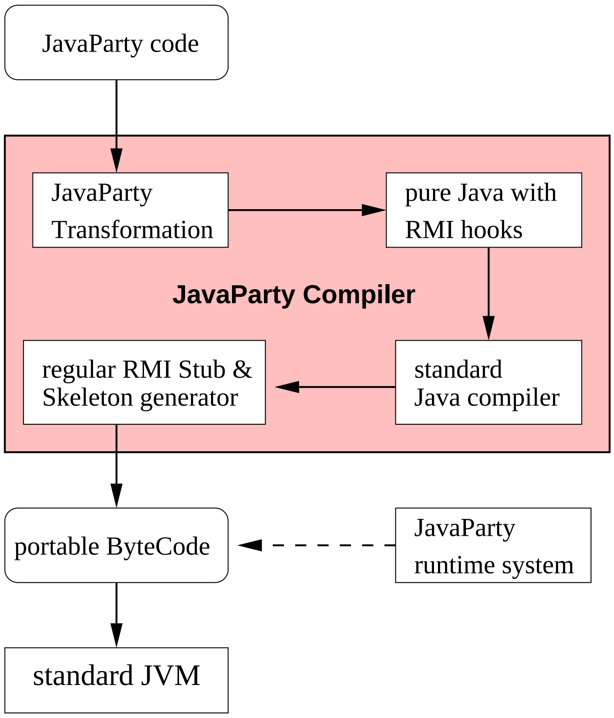

Figure 2.4: The JavaParty system.

2.4 Related Work of Java Grande

Currently several projects are under way that achieve increased bandwidth, reduced latency, and better reliability within workstation clusters using Java. In this section we briefly discuss some works related to Java Grande.

2.4.1 JavaParty

The JavaParty [83] system of the University of Karlsruhe, Germany, provides a mechanism for parallel cluster programming in Java. JavaParty adds remote objects to Java purely by declaration avoiding disadvantages of explicit socket communication and the programming overhead of RMI. The `remote` keyword is used to identify which objects can be called remotely. JavaParty is location transparent, i.e., the programmer does not need to map remote objects and remote threads to specific nodes of the network; compiler and runtime system deal with locality and communication optimization.

JavaParty is implemented as a pre-processing phase into Java compilers, EspressoGrinder [77] and Pizza [78]. As shown in the figure 2.4, JavaParty code is transformed into regular Java code plus RMI hooks. The resulting RMI portions are then fed into RMI compiler(stub & skeleton generator).

The runtime system consists of a central component, called *RuntimeManager*. In addition, each host runs a *LocalJP* that is registered at the manager. A host and its LocalJP can be added dynamically to the system. The manager knows all LocalJPs and knows the location of all class objects, i.e., for each class that is loaded the manager knows which host implements the static part. This information is replicated in the LocalJPs to reduce manager load. Neither manager nor LocalJPs need to know the location of individual remote objects. LocalJPs are needed to call constructors in class objects and to implement either side of migration.

JavaParty also supports object migration. For actually migrating an object, the runtime system offers two options. An object can move to the position of the caller or it can be moved to the location of different remote object. A migration is possible if no method is executing on the object, no other migration is in progress, and the object does not have its `resident` flag set. Then the object's internal state is serialized into a byte array which is sent to the runtime system on the target host. The receiving runtime system unpacks the byte array and returns an RMI stub reference to the runtime system of the original host. Upon receipt, the original host completes the migration.

2.4.2 Javelin

Javelin [30] of the University of California, Santa Barbara, is internet-based global computing infrastructures that support Java. In Javelin, there are three kinds of

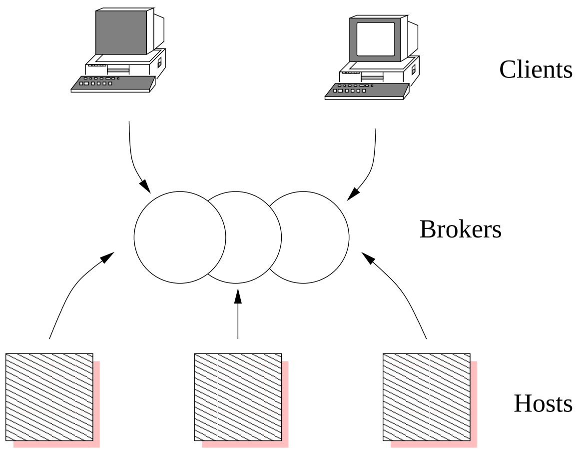

Figure 2.5: The Javelin Architecture.

participating entities: clients, hosts and brokers. A *client* is a process seeking computing resources; a *host* is a process offering computing resources; a *broker* is a process that coordinates the supply and demand for computing resources. Figure 2.5 illustrates Javelin architecture. Clients register with the broker their tasks to be run; hosts register with the broker their intention to run tasks. The broker assigns tasks to hosts that, then, run the tasks and send results back to the clients.

One of the most important goals of Javelin is *simplicity*, i.e., to enable everyone connected to the Internet or an intranet to *easily* participate in Javelin. To this end, the design is based on widely used components: Web browsers and the portable Java. By simply pointing their browser to a known URL of a broker, users automatically make their resources available to host parts of parallel computations. This is achieved by downloading and executing an applet that spawns a small daemon thread that waits and "listens" for tasks from the broker. The simplicity of this approach makes it easy for a host to participate–all that is needed is a Java-capable web browser and the URL of the broker.

In Figure 2.6 shows the steps involved in the remote execution of an applet. These steps are:

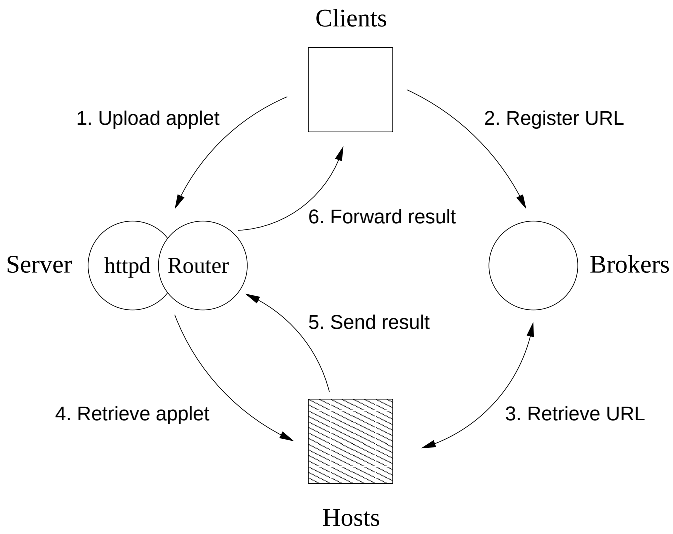

Figure 2.6: Steps involved in the remote execution of an applet.

1. The client uploads the applet and an embedding HTML page on an HTTP server. Clients running their own HTTP server may skip the first step.

2. The client registers the corresponding URL with a broker.

3. The host registers with the broker its intention to execute tasks, and retrieves the URL of a task.

4. The host downloads the HTML page from the HTTP server, and executes its embedded applet.

5. The host stores the result at the server site. If communication between co-operating hosts is required, messages are stored at, and retrieved from, the server site.

6. The client retrieves the result.

The server is an HTTP server that performs two separate functions. First, it stores clients' applets and serves them to hosts; secondly, it provides point-to-point communication among hosts by serving as smart router.

Tasks are represented as applets embedded in HTML pages. This design decision implies certain limitations due to Java applet security: E.g., all communication must be routed through the broker and every file access involves network communication. Therefore, in general, coarse-grained applications with a high computation to communication ratio are well suited to Javelin.

2.4.3 Efficient RMI and Object Serialization

One of the goals of Java Grande Forum is to investigate into methods to optimize Java RMI and Object Serialization performance. RMI implementations are built on top of TCP/IP, which was not designed for High Performance Computing. Several research groups are pursuing research to improve efficiency of Java RMI and object serialization implementation and its performance.

As part of the JavaParty [83] project at the University of Karlsruhe, Germany, a more efficient RMI and object serialization have been studied. Philippsen et al [81] describe detailed analysis of RMI and object serialization and it's performance problem. They also suggest modifications to improve RMI and object serialization. Based on these suggestions, they implement more efficient version of RMI and object serialization called KaRMI and UKA-serialization. KaRMI [74] improves the performance of RMI by clean interfaces between layers and the separation of RMI's design from transport level sockets. They also modify RMI to support non-TCP/IP communication networks. KaRMI cannot handle code that uses socket factory or port numbers. UKA-serialization [82] uses explicit marshaling/unmarshaling routines instead of reflection and a minimal encoding of type information to achieve better performance.

Henri Bal's group of Vrije University, Netherlands, study an alternative approach for implementing RMI, called Manta RMI [68], based on native compilation for better optimization. Manta [91] is compiler-based Java system and uses a native compiler that generates executable code rather than byte code. Manta system supports two different communication protocols. Between Manta nodes, it can communicate through Manta's own a fast RMI protocol using Manta's own serialization format. Manta node can communicate with another JVM through a JDK-compliant protocol using Sun's serialization format. With Manta, the serialization code is generated by the compiler which make it possible to avoid the overhead of dynamic inspection of the object structure.

Krishnaswamy et al [65] improve RMI performance by adding object caching at the reference layer and using UDP(User Datagram Protocol)-based reliable message delivery protocol.

Another implementation of RMI is provided by Thiruvathukal et al. Reflective RMI(RRMI) [90] adopted the reflection feature of Java programming language to invoke remote methods. RRMI makes it possible of use RMI without stubs and skeletons. A remote object is created dynamically at the run time by implementing its own class loader on the server side. It also supports both synchronous and a asynchronous remote method invocations.

Raje et al implement ARMI (Asynchronous RMI) [86] which allows an asynchronous remote method to RMI. It achieve asynchronous communication by replacing the RMI client-side stub by adding thread mechanism.

NexusRMI [20] implements Java RMI on top of NexusJava [47], which is a Java interface to the Nexus communication system developed at Argone National Laboratory. NexusRMI also has support for object serialization of all object types except for exception objects.

At the group of the University of California, Berkeley, NinjaRMI [94] has been developed to support Ninja project [93]. NinjaRMI has some features that are not found in Sun's RMI including multiple communication protocols.

2.4.4 Other parallel and distributed computing in Java

Most Java based parallel systems can be classified into four classes.

- **Use Java as wrapper**

  Java/DSM is heterogeneous computing system which implements a JVM on top of Tread-Marks distributed shared memory(DSM) system. In DSM system, no explicit communication is necessary and that eliminates the need for the programmer to write message-passing code. Because of the portability feature of Java, the combination of Java and DSM completely hides the hardware differences and the distributed nature of the system from the programmer view, providing a better platform for heterogeneous computing environments.

- **Use pure Java libraries**

  JavaNow [15] is a pure Java based framework on shared memory system that includes features found in Linda, Actors, and Message Passing Interface(MPI). It creates a PVM like virtual parallel machine and provides distributed associative shared memory. JavaNow is currently available as stand-alone software and must be used with a remote (or secure) shell tool in order to run on a network of workstations.

- **Extended Java language**

  Titanium [99] is a Java based language and system for high-performance parallel computing. It has additional features over Java like immutable classes, multidimensional arrays, an explicitly parallel SPMD model of computation with a global address space, and a mechanism for programmer to control memory management. Immutable classes allow the compiler pass objects by value. Because Titanium compiler translates Titanium into C, users do not have full access to the AWT and certain other Java libraries directly from Titanium.

Spar [92] is a programming language for semi-automatic parallel programming, in particular for the programming of array-based applications, supporting multidimensional and sparse arrays. Spar has been designed as part of the Automap project, in which a compiler and run-time system are being developed for distributed memory systems.

javar [17] is a simple Java restructuring compiler, written in C, that can automatically transform a Java program into a form in which implicit parallelism is made explicit by means of multi-threading.

- **Web oriented Java based frameworks**

  WebFlow [60] of Northeast Parallel Architectures Center, Syracuse University, is a particular programming paradigm implemented over a virtual Web accessible metacomputer and given by a dataflow-programming model. WebFlow application is given by a computational visually edited by end-users, using Java applets. Modules are written by module developers, people who have only limited knowledge of the system on which the modules will run. They not need concern themselves with issues such as: allocating and running the modules on various machines, creating connections among the modules, sending and receiving data across these connections, or running several modules concurrently on one machine. The WebFlow system hides these management and coordination functions from the developers, allowing them to concentrate on the modules being developed.

  IceT [57] is an inherently collaborative, multi-user environment. IceT provides mechanism by which resources can be made available to the system for use by users who do not have log-on privileges. The attractive goals of this metacomputing-based approach include better utilization of larger resource bases, as well as an enhanced collaborative environment for high performance

Category	Wrapper	Pure Java	Extended Java	Web oriented
	mpiJava	MPJ	HPJava	WebFlow
System	JavaMPI	MPIJ	Titanium	Javelin
	JavaPVM	JPVM	Spar	IceT
	Java/DSM	JavaNOW	javar	
			JavaParty	

Table 2.1: Categories of Java Grande Related Work

computing. Among the challenges introduced by this idea are the numerous security issues introduced by code-upload capabilities for non-privileged uses.

The major categories of Java Grande Related Work are shown in Table 2.1. We will discuss mpiJava, MPJ and HPJava in more detail in the next chapter.

Chapter 3

HPJava: Data Parallel Extensions to Java

This chapter discusses some design and implementation issues in the *HPJava* language. The language is briefly reviewed, then the class library that forms the foundation of the translation scheme is described. Through example codes, we illustrate how HPJava source codes can be translated straightforwardly to ordinary SPMD Java programs calling this library. This is followed by a discussion of the rationale for introducing the language in the first place, and of how various language features have been designed to facilitate efficient implementation.

3.1 HPJava

HPJava is a programming language extended from Java to support parallel programming, especially (but not exclusively) data parallel programming on message passing and distributed memory systems, from multi-processor systems to workstation clusters.

The *HPJava* framewrok would be a general environment for parallel computation. Ultimately it should combine tools, class libraries, and language extensions to support various established paradigms for parallel computation, including shared memory programming, explicit message-passing, and array-parallel programming. Other paradigms (for example, Linda or coarse-grained data-flow) may come later, together with bindings to higher-level libraries and application-specific libraries such as CHAOS [35], ScaLAPACK [18], Global Arrays [75] or DAGH [79].

This is a large vision, and the current dissertation we only discuss some first steps towards a general framework. In particular we will make specific proposals for the sector of HPJava most directly related to its namesake: High Performance Fortran. We will be concentrating on array-parallel programming.

For now we do not propose to import the full HPF programming model to Java. After several years of effort by various compiler groups, HPF compilers are still quite immature. It seems difficult justify a comparable effort for Java before success has been convincingly demonstrated in Fortran. In any case there are features of the HPF model that make it less attractive in the context of the integrated parallel programming environment we envisage. Although an HPF program *can* interoperate with modules written in other parallel programming styles through the HPF extrinsic procedure interface, that mechanism is quite awkward. Rather than follow the HPF model directly, we propose introducing some of the characteristic ideas of HPF—specifically its distributed array model and array intrinsic functions and libraries—into a basically SPMD programming model. Because the programming model is SPMD, direct calls to MPI or other communication packages are allowed from the HPJava program.

The language outlined here provides HPF-like distributed arrays as language primitives, and new *distributed control* constructs to facilitate access to the local elements of these arrays. In the SPMD mold, the model allows processors the freedom to independently execute complex procedures on local elements: it is not limited by SIMD-style array syntax. All access to *non-local* array elements must go through library functions—typically collective communication operations. This puts an extra onus on the programmer; but making communication explicit encourages the programmer to write algorithms that exploit locality, and simplifies the task of the compiler writer. On the other hand, by providing distributed arrays as language primitives we are able to simplify error-prone tasks such as converting between local and global array subscripts and determining which processor holds a particular element. As in HPF, it is possible to write programs at a natural level of abstraction where the meaning is insensitive to the detailed mapping of elements. Lower-level styles of programming are also possible.

Our compiler will be implemented as a translator to ordinary Java with calls to a suitable run-time library. The underlying library is already available [25], and the Java interface needed by the translator is under development. The translator itself is being implemented in a compiler construction framework developed in the PCRC project [33, 100].

3.2 The overview of Adlib and other runtime libraries

3.2.1 Adlib

The Adlib [26] runtime library was initially designed to support HPF translation. Early development took place in the *shpf* [70] project at Southampton, UK. Subsequently the library was redesigned and reimplemented at Syracuse during in the PCRC project, and delivered as the NPAC PCRC runtime kernel [25]. It has been used as a foundation of two experimental HPF compilation systems [70, 100], (one in Europe and one at Syracuse), and is currently being used as a basis of the HPJava translator.

The Adlib kernel is C++ class library, built on MPI. Fortran, C++ and Java interfaces are available. It provides communication primitives similar to Multiblock PARTI, plus the Fortran 90 transformational intrinsics for arithmetic on distributed arrays. It also provides some collective gather/scatter operations for irregular access. The existing Adlib communication library emphasizes collective communication rather than one-sided communication. Benchmarks reported in [100] suggested Adlib provides superior performance to the then-current version of the commercial PGHPF [84] compiler.

The array descriptor of Adlib supports the full HPF 1.0 distributed array model—including all standard distribution formats, all alignment options including replicated alignment. The runtime array descriptor of the HPspmd languages will be an enhanced version of the Adlib descriptor. The Adlib collective communication library will provide initial library support for regular applications in HPspmd.

Array descriptor An important part of the Adlib library is its characteristic *Distributed Array Descriptor* (DAD). Adlib is implemented in C++, and the underlying array descriptor is implemented as an object of type `DAD`. A DAD object describes how elements of a particular array are distributed across available processors. It has a simple structure containing three fields: an integer value (r, say) defining the *rank* (dimensionality) of the array, a process *group* object, and a vector of r *map* objects, one for each dimension of the array. The group defines a multidimensional process grid embedded in the set of processes executing the program, or some subset. Each map object consists of an integer local *memory stride* and a *range* object. A range object describes the *extent* (size) and distribution format of one dimension of a distributed array. Adlib defines a class hierarchy of different kinds of range object. Each subclass represents a different kind of distribution format for an array dimension.

Communication schedules The Adlib communication library supports *collective operations* on distributed arrays. A call to a collective operation must be invoked simultaneously by all members of some *active process group*, which may or may not be the entire set of processes executing the program.

Communication patterns supported include HPF/F90 intrinsics such as CSHIFT and TRANSPOSE. More importantly they include the regular-section copy operation, `remap`, which copies elements between shape-conforming array sections regardless of source and destination mapping. Another function, `writeHalo`, updates ghost areas of a distributed array. Various collective gather and scatter operations allow irregular patterns of access. The library also provides essentially all F90 arithmetic transformational functions on distributed arrays and various additional HPF library functions.

All collective operations in the library are based on communication *schedule* objects. Each kind of operation has an associated class of schedules. Particular instances of these schedules, involving particular data arrays and other parameters, are created by the class constructors.

3.2.2 Related libraries and languages

....

ScaLAPACK , PetSc....

Multiblock Parti , Global Array toolkit...

CHAOS, DAGH...

....

We mentioned two extended Java based languages related projects in Section 2.4.4, Spar and Titanium. Spar [92] is a Java-based language for array-parallel programming. Like our language it introduces multi-dimensional arrays, array sections, and a parallel loop. There are some similarities in syntax, but semantically Spar is very different to HPJava. Spar expresses parallelism but not explicit data placement or communication—in this sense it is a higher level language—closer to HPF.

Titanium [99] is another Java-based language for high-performance computing. It provides multi-dimensional arrays and a global address space, with an SPMD programming model. It does not provide any special support for distributed arrays, and the programming style is quite different to HPJava.

F-- [76] is an extended Fortran dialect for SPMD programming. The approach is quite different to the one proposed here. In F--, array subscripting is local by default, or involves a combination of local subscripts and explicit process ids. There is no analogue of global subscripts, or HPF-like distribution formats. In F-- the logical model of communication is built into the language—remote memory access with intrinsics for synchronization—where we follow the philosophy of providing communication through separate libraries. While F-- and our approach share an underlying programming model, we believe that our framework offers greater opportunities for exploiting established library technologies.

ZPL [87] is a new programming language for scientific computations. Like Spar, it is an array language. It has an idea of performing computations over a *region*, or set of indices. Within a compound statement prefixed by a *region specifier*, aligned elements of arrays distributed over the same region can be accessed. This idea has certain similarities to our *overall* construct. Communication is more explicit than, say, Spar, but not as explicit as in the language discussed in this dissertation.

3.3 Multidimensional arrays

First we describe a modest extension to Java that adds a class of true multi-dimensional arrays to the standard Java language. The new arrays allow regular section subscripting, similar to Fortran 90 arrays. The syntax described in this section is a subset of the syntax introduced later for parallel arrays and algorithms: the only motivation for discussing the sequential subset first is to simplify the overall presentation. No attempt is made to integrate the new multidimensional arrays with the standard Java arrays: they are a new kind of entity that coexists in the language with ordinary Java arrays. There are good technical reasons for keeping the two kinds of array separate [1].

The type-signatures and constructors of the multidimensional array use double brackets to distinguish them from ordinary arrays:

```
int [[,]] a = new int [[5, 5]] ;

float [[,,]] b = new float [[10, n, 20]] ;

int [[]] c = new int [[100]] ;
```

`a`, `b` and `c` are respectively 2-, 3- and one- dimensional arrays. Of course `c` is very similar in structure to the standard array `d`, created by

```
int [] d = new int [100] ;
```

`c` and `d` are not identical, though[2].

Access to individual elements of a multidimensional array goes through a subscripting operation involving single brackets, for example

[1]The run-time representation of our multi-dimensional arrays includes extra descriptor information that would simply encumber the large class "non-scientific" Java applications.

[2]For example, `c` allows section subscripting, whereas `d` does not.

```
for(int i = 0 ; i < 4 ; i++)
  a [i, i + 1] = i + c [i] ;
```

For reasons that will become clearer in later sections, this style of subscripting is called *local subscripting*. In the current sequential context, apart from the fact that a single pair of brackest may include several comma-separated subscripts, this kind of subscripting works just like ordinary Java array subscripting. Subscripts always start at zero, in the ordinary Java or C style (there is no Fortran-like lower bound).

In general our language has no idea of Fortran-like array assignments. In

```
int [[,]] e = new int [[n, m]] ;
...
a = e ;
```

the assignment simply copies a handle to object referenced by `e` into `a`. There is no element-by-element copy involved. Similarly we introduce no idea of elemental arithmetic or elemental function application. If `e` and `a` are arrays, the expressions

```
e + a
Math.cos(e)
```

are type errors.

Our HPJava *does* import a Fortran-90-like idea of array *regular sections*. The syntax for *section subscripting* is different to the syntax for local subscripting. Double brackets are used. These brackets can include scalar subscripts or *subscript triplets*.

A section is an object in its own right—its type is that of a suitable multi-dimensional array. It describes some subset of the elements of the parent array. This is slightly different to the situation in Fortran, where sections cannot usually be captured as named entities[3].

[3]Unless a section appears as an actual argument to a procedure, in which case the dummy argument names that section, or it is the target of a pointer assignment.

```
int [[]] e = a [[2,  2 :]] ;

foo(b [[ : , 0, 1 : 10 : 2]]) ;
```

`e` becomes an alias for the 3rd row of elements of `a`. The procedure `foo` should expect a two-dimensional array as argument. It can read or write to the set of elements of `b` selected by the section. As in Fortran, upper or lower bounds can be omitted in triplets, defaulting to the actual bound of the parent array, and the stride entry of the triplet is optional. The subscripts of `e`, like any other array, start at 0, although the first element is identified with `a [2, 2]`.

In our language, unlike Fortran, it is not allowed to use vectors of integers as subscripts. The only sections recognized are regular sections defined through scalar and triplet subscripts.

The language provides a library of functions for manipulating its arrays, closely analogous to the array transformational intrinsic functions of Fortran 90:

```
int [[,]] f = new int [[5, 5]] ;
HPJlib.shift(f, a, -1, 0, CYCL) ;

float g = HPJlib.sum(b) ;

int [[]] h = new int [[100]] ;
HPJlib.copy(h, c) ;
```

The `shift` operation with shift-mode `CYCL` executes a cyclic shift on the data in its second argument, copying the result to its first argument—an array of the same shape. In the example the shift amount is -1, and the shift is performed in dimension 0 of the array—the first of its two dimensions. The `sum` operation simply adds all elements of its argument array. The `copy` operation copies the elements of its second argument to its first—it is something like an array assignment. These functions may have to be overloaded to apply to some finite set of array types, eg they may be defined for arrays with elements of any suitable Java primitive type, up to some maximum rank of array. Alternatively the type-hierarchy for arrays can be defined in a way that allows these functions to be more polymorphic.

3.4 Process arrays

HPJava adds class libraries and some additional syntax for dealing with *distributed arrays*. These arrays are viewed as coherent global entities, but their elements are divided across a set of cooperating processes. As a pre-requisite to introducing distributed arrays we discuss the *process arrays* over which their elements are scattered.

An abstract base class `Procs` has subclasses `Procs1`, `Procs2`, ..., representing one-dimensional process arrays, two-dimensional process arrays, and so on.

```
Procs2 p = new Procs2(2, 2) ;
Procs1 q = new Procs1(4) ;
```

These declarations set `p` to represent a 2 by 2 process array and `q` to represent a 4-element, one-dimensional process array. In either case the object created describes a group of 4 processes. At the time the `Procs` constructors are executed the program should be executing on four or more processes. Either constructor selects four processes from this set and identifies them as members of the constructed group[4].

`Procs` has a member function called `member`, returning a boolean value. This is `true` if the local process is a member of the group, `false` otherwise.

```
if(p.member()) {
  ...
```

[4]There is no cooperation between the two constructor calls for `p` and `q`, so an individual physical process might occur in both groups or in neither. As an option not illustrated here, vectors of ids can be passed to the `Procs` constructors to specify exactly which processes are included in a particular group.

```
}
```

The code inside the `if` is executed only if the local process is a member `p`. We will say that inside this construct the *active process group* is restricted to `p`.

The multi-dimensional structure of a process array is reflected in its set of *process dimensions*. An object is associated with each dimension. These objects are accessed through the inquiry member `dim`:

```
Dimension x = p.dim(0) ;
Dimension y = p.dim(1) ;

Dimension z = q.dim(0) ;
```

The object returned by the `dim` inquiry has class `Dimension`. The members of this class include the inquiry `crd`. This returns the coordinate of the local process with respect to the process dimension. The result is only well-defined if the local process is a member of the parent process array. The inner body code in

```
if(p.member())
  if(x.crd() == 0)
    if(y.crd() == 0) {
      ...
    }
```

will only execute on the first process from `p`, with coordinates $(0, 0)$.

3.5 Distributed arrays

Some or all of the dimensions of a multi-dimensional array can be declared to be *distributed ranges*. In general a distributed range is represented by an object of class `Range`. A `Range` object defines a range of integer subscripts, and defines how they are mapped into a process array dimension. In fact the `Dimension` class introduced in the previous section is a subclass of `Range`. In this case the integer range is just the range of coordinate values associated with the dimension. Each value in the range is mapped, of course, to the process (or slice of processes) with that coordinate. This kind of range is also called a *primitive range*. More complex subclasses of `Range` implement more elaborate maps from integer ranges to process dimensions. Some of these will be introduced in later sections. For now we concentrate on arrays constructed with `Dimension` objects as their distributed ranges.

The syntax of section 3.3 is extended in the following way to support distributed arrays

- A distributed range object may appear in place of an integer extent in the "constructor" of the array (the expression following the `new` keyword).
- If a particular dimension of the array has a distributed range, the corresponding slot in the type signature of the array should include a `#` symbol.
- In general the constructor of the distributed array must be followed by an `on` clause, specifying the process group over which the array is distributed. Distributed ranges of the array must be distributed over distinct dimensions of this group[5].

Assume `p`, `x` and `y` are declared as in the previous section, then

```
float [[#,#,]] a = new float [[x, y, 100]] on p ;
```

defines `a` as a 2 by 2 by 100 array of floating point numbers. Because the first two dimensions of the array are distributed ranges—dimensions of `p`—`a` is actually realized as four segments of 100 elements, one in each of the processes of `p`. The process in `p` with coordinates `i`, `j` holds the section `a [[i, j, :]]`.

[5]The `on` clause can be omitted in some circumstances—see Section 3.6.

The distributed array `a` is equivalent in terms of storage to four local arrays defined by

```
float [] b = new float [100] ;
```

But because `a` is declared as a collective object we can apply collective operations to it. The `HPJlib` functions introduced in section 3.3 apply equally well to distributed arrays, but now they imply inter-processor communication.

```
float [[#,#,]] a = new float [[x, y, 100]] on p,
               b = new float [[x, y, 100]] on p ;

HPJlib.shift(a, b, -1, 0, CYCL) ;
```

The `shift` operation causes the local values of `a` to be overwritten with values of `b` from a processor adjacent in the `x` dimension.

There is a catch in this. When subscripting the distributed dimensions of an array it is *simply disallowed* to use subscripts that refer to off-processor elements. While this:

```
int i = x.crd(), j = y.crd() ;

a [i, j, 20] = a [i, j, 21] ;
```

is allowed, this:

```
int i = x.crd(), j = y.crd() ;

a [i, j, 20] = b [(i + 1) % 2, j, 20] ;
```

is forbidden. The second example could apparently be implemented using a nearest neighbour communication, quite similar to the `shift` example above. But our language imposes an strict policy distinguishing it from most data parallel languages:

while library functions may introduce communications, language primitives such as array subscripting *never* imply communication.

If subscripting distributed dimensions is so restricted, why are the `i`, `j` subscripts on the arrays needed at all? In the examples of this section these subscripts are only allowed one value on each processor. Well, the inconvience of specifying the subscripts will be reduced by language constructs introduced later, and the fact that only one subscript value is local is a special feature of the *primitive ranges* used here. The higher level distributed ranges introduced later map multiple elements to individual processes. Subscripting will no longer look so redundant.

3.6 The *on* construct and the active process group

In the section 3.4 the idiom

```
if(p.member()) {
  ...
}
```

appeared. Our language provides a short way of writing this construct

```
on(p) {
  ...
}
```

In fact the *on* construct provides some extra value. Informally we said in section 3.4 that the *active process group* is restricted to `p` inside the body of the `p.member()` conditional construct. The language incorporates a more formal idea of an active process group (APG). At any point of execution some process group is singled out as the APG. An `on(p)` construct specifically changes the value of the APG to `p`. On exit from the construct, the APG is restored to its value on entry.

Elevating the active process group to a part of the language allows some simplifications. For example, it provides a natural default for the `on` clause in array constructors. More importantly, formally defining the active process group simplifies the statement of various rules about what operations are legal *inside* distributed control constructs like *on*.

3.7 Higher-level ranges and locations

The class `BlockRange` is a subclass of `Range` which describes a simple block-distributed range of subscripts. Like `BLOCK` distribution format in HPF, it maps blocks of contiguous subscripts to each element of its target process dimension[6]. The constructor of `BlockRange` usually takes two arguments: the extent of the range and a `Dimension` object defining the process dimension over which the new range is distributed.

```
Procs2 p = new Procs2(3, 2) ;

Range x = new BlockRange(100, p.dim(0)) ;
Range y = new BlockRange(200, p.dim(1)) ;

float [[#,#]] a = new float [[x, y]] on p ;
```

`a` is created as a 100×200 array, block-distributed over the 6 processes in `p`. The fragment is essentially equivalent to the HPF declarations

```
!HPF$ PROCESSORS p(3, 2)

REAL a(100, 200)
```

[6]Other higher-level ranges include `CyclicRange`, which produces the equivalent of `CYCLIC` distribution format in HPF.

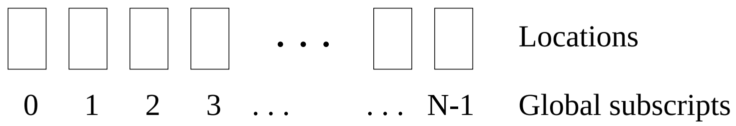

Figure 3.1: A range regarded as a set of locations, or slots.

```
!HPF$ DISTRIBUTE a(BLOCK, BLOCK) ONTO p
```

Subscripting distributed arrays with non-primitive ranges introduces some new problems. An array access such as

```
a [17, 23] = 13 ;
```

is perfectly legal *if* the local process holds the element in question. But deterimining whether an element is local is no longer so easy. When arrays had only *primitive* distributed ranges, it was straightforward to check that accesses were local—the subscript simply had to be equal to the local coordinate. With higher-level ranges, that simple condition no longer holds.

In practise it is unusual to use integer values directly as local subscripts in distributed array dimensions. Instead the idea of a *location* is introduced. A location can be viewed as an abstract element, or "slot", of a distributed range. Conversely, a range can be thought of as a set of locations. This model of a range is visualized in figure 3.1. An individual location is described by an object of the class `Location`. Each `Location` element is mapped to a particular slice of a process grid. In general two locations are identical only if they come from the same position in the same range. A subscripting syntax is used to represent location `n` in range `x`:

```
Location i = x [n]
```

This is an important idea in HPJava. By working in terms of abstract locations—elements of distributed ranges—one can usually respect locality of reference without resorting explicitly to low-level local subscripts and process ids. In fact the location can be viewed as an abstract data type incorporating these lower-level offsets.

Publically accessible fields of `Location` include `dim` and `crd`. The first is the process dimension of the parent range. The second is coordinate in that dimension to which the element is mapped. So the access to element `a [17, 23]` could now be guarded by conditionals as follows:

```
Location i = x [17], j = y [23] ;

if(i.crd == i.dim.crd())
  if(j.crd == j.dim.crd())
    a [17, 23] = 13 ;
```

This is still quite verbose and error-prone. The language provides a second *distributed control construct* (analogous to *on*) to deal with this common situation. The new construct is called *at*, and takes a location as its argument. The fragment above can be replaced with

```
Location i = x [17], j = y [23] ;

at(i)
  at(j)
    a [17, 23] = 13 ;
```

This is more concise, but still involves some redundancy because the subscripts 17 and 23 appear twice. A natural extension is to allow locations to be used directly as array subscripts:

```
Location i = x [17], j = y [23] ;

at(i)
  at(j)
    a [i, j] = 13 ;
```

Locations used as array subscripts must be elements of the corresponding ranges of the array.

The range class has a member function

```
int Range.idx(Location i)
```

which can be used to recover the integer subscript, given a location in the range.

There is a restriction that an `at(i)` construct should only appear at a point of execution where `i.dim` is a dimension of the active process group. In the examples of this section this means that an `at(i)` construct, say, should normally be nested directly or indirectly inside an `on(p)` construct.

3.8 Distributed loops

Good parallel algorithms don't usually expend many lines of code assigning to isolated elements of distributed arrays. The *at* mechanism of the previous section is often useful, but a more pressing need is a mechanism for *parallel* access to distributed array elements. The last and most important distributed control construct in the language is called *over*. It implements a distributed parallel loop. Conceptually it is quite similar to the `FORALL` construct of Fortran, except that the *over* construct specifies exactly where its parallel iterations are to be performed. The argument of *over* is a member of the special class `Index`. The class `Index` is a subclass of `Location`, so it is syntactically correct to use an index as an array subscript[7]. Here is an example of a pair of nested *over* loops:

```
float [[#,#]] a = new float [[x, y]],
              b = new float [[x, y]] ;
...
```

[7]But the effect of such subscripting is only well-defined inside an *over* construct parametrised by the index in question.

```
Index i, j ;
over(i = x | :)
  over(j = y | :)
    a [i, j] = 2 * b [i, j] ;
```

The body of an *over* construct executes, conceptually in parallel, for every location in the range of its index (or some subrange if a non-trivial triplet is specified)[8]. An individual "iteration" executes on just those processors holding the location associated with the iteration. In a particular iteration, the location component of the index (the base class object) is equal to that location. The net effect of the example above should be reasonably clear. It assigns twice the value of each element of `b` to the corresponding element of `a`. Because of the rules about *where* an individual iteration iterates, the body of an *over* can usually only combine elements of arrays that have some simple alignment relation relative to one another. The `idx` member of range can be used in parallel updates to give expressions that depend on global index values.

With the *over* construct we can give some useful examples of parallel programs. Figure 3.2 is the famous Jacobi iteration for a two dimensionsional Laplace equation. We have used cyclic `shift` to implement nearest neighbour communications[9].

Copying whole arrays into temporaries is not an efficient way of accessing nearest neighbours in an array. Because this is such a common pattern of communication, the standard library supports *ghost regions*. Distributed arrays can be created in such a way that the segment stored locally is extended with some halo. This halo caches values stored in the segments of adjacent processes. The cached values are

[8]Formally | is being used here as an operator that combines a range and a triplet to return an object of the iterator class `Index`.

[9]Laplace's equation with cyclic boundary conditions is not particularly useful, but it illustrates the language features. More interesting boundary conditions can easily be incorporated later. Incidentally, this is a suitable place to mention that the array arguments of `shift` must be *aligned* arrays—they must have identical distributed ranges.

```
Procs2 p = new Procs2(2, 2) ;

on(p) {
  Range x = new BlockRange(100, p.dim(0)) ;
  Range y = new BlockRange(200, p.dim(1)) ;

  float [[#,#]] u = new float [[x, y]] ;

  // ... some code to initialise 'u'

  float [[#,#]] unx = new float [[x, y]], upx = new float [[x, y]],
                uny = new float [[x, y]], upy = new float [[x, y]] ;

  HPJlib.shift(unx, u,  1, 0, CYCL) ;
  HPJlib.shift(upx, u, -1, 0, CYCL) ;
  HPJlib.shift(uny, u,  1, 1, CYCL) ;
  HPJlib.shift(upy, u, -1, 1, CYCL) ;

  Index i, j ;
  over(i = x | :)
    over(j = y | :)
      u [i, j] = 0.25 * (unx [i, j] + upx [i, j] +
                         uny [i, j] + upy [i, j]) ;
}
```

Figure 3.2: Jacobi iteration using `shift`.

explicitly bought up to date by the library operation `writeHalo`.

An optimized version of the Jacobi program is give in figure 3.3. This version only involves a singe array temporary. A new constructor for `BlockRange` is provided. This allows the width of the ghost extensions to be specified. The arguments of `writeHalo` itself are an array with suitable extensions and two vectors. The first defines in each dimension the width of the halo that must actually be updated, and the second defines the treatment at the ends of the range—in this case the ghost edges are updated with cyclic wraparound. The new constructor and new `writeHalo` function are simply standard library extensions. One new piece of syntax is needed: the addition and subtraction operators are overloaded so that integer offsets can be added or subtracted to `Location` objects, yielding new, shifted, locations. The usual access rules apply—this kind of shifted access is illegal if it implies access to off-processor data. It only works if the subscripted array has suitable ghost extensions.

3.9 Other features

We have already described most of the important *language* features we propose to implement. Two additional features that are quite important in practice but have not been discussed are *subranges* and *subgroups*. A subrange is simply a range which is a regular section of some other range, created by syntax like `x [0 : 49]`. Subranges are created tacitly when a distributed array is subscripted with a triplet, and they can also be used directly to create distributed arrays with general HPF-like alignments. A *subgroup* is some slice of a process array, formed by restricting process coordinates in one or more dimensions to single values. Again they may be created implicitly by section subscripting, this time using a scalar subscript. They also formally describe the state of the active process group inside *at* and *over* constructs.

```
Procs2 p(2, 2) ;

on(p) {
  Range x = new BlockRange(100, p.dim(0), 1) ;  // ghost width 1
  Range y = new BlockRange(200, p.dim(1), 1) ;  // ghost width 1

  float [[#,#]] u = new float [[x, y]] ;

  // ... some code to initialise ‘u’

  int [] widths = {1, 1} ;        // Widths actually updated
  Mode [] modes = {CYCL, CYCL} ;  // Wraparound at ends.

  HPJlib.writeHalo(u, widths, modes) ;

  float [[#,#]] v = new float [[x, y]] ;

  Index i, j ;
  over(i = x | :)
    over(j = y | :)
      v [i, j] = 0.25 * (u [i + 1, j] + u [i - 1, j] +
                         u [i, j + 1] + u [i, j + 1]) ;

  HPJlib.copy(u, v) ;
}
```

Figure 3.3: Jacobi iteration using `writeHalo`.

The framework described is much more powerful than space allows us to demonstrate in this paper. This power comes in part from the flexibility to add features by extending the libraries associated with the language. We have only illustrated the simplest kinds of distribution format. But any HPF 1.0 array distribution format, plus various others, can be incorporated by extending the `Range` hierarchy in the run-time library. We have only illustrated `shift` and `writeHalo` operations from the communication library, but the library also includes much more powerful operations for remapping arrays and performing irregular data accesses. Our intention is to provide minimal language support for distributed arrays, just enough to facilitate further extension through construction of new libraries.

3.10 Discussion

We have described a conservative set of extensions to Java. In the context of an explicitly SPMD programming environment with a good communication library, we claim these extensions provide much of the concise expressiveness of HPF, without relying on very sophisticated compiler analysis. The object-oriented features of Java are exploited to give an elegant parameterization of the distributed arrays of the extended language. Because of the relatively low-level programming model, interfacing to other parallel-programming paradigms is more natural than in HPF. With suitable care, it is possible to make direct calls to, say, MPI from within the data parallel program. In Chapter 4 we suggest a concrete Java binding for MPI.

Chapter 4

mpiJava: An Object-Oriented Java Interface to MPI

It is generally recognized that the vast majority of scientific and engineering applications are written in either C, C++ or Fortran. The recent popularity of Java has led to it being seriously considered as a good language to develop scientific and engineering applications, and in particular for parallel computing[34, 49, 52, 53, 54]. Sun's claims, on behalf of Java, that it is simple, efficient and platform-neutral–a natural language for network programming–makes it attractive to scientific programmers who wish to harness the collective computational power of parallel platforms as well as networks of workstations or PCs, with interconnections ranging from LANs to the Internet. The attractiveness of Java for scientific computing is being encouraged by bodies like Java Grande[4]. The Java Grande forum has been set up to co-ordinate the communities efforts to standardize many aspects of Java and so ensure that its future development makes it more appropriate for scientific programmers.

Developers of parallel applications generally use the Single Program Multiple Data (SPMD) model of parallel computing, wherein a group of processes cooperate by executing identical program images on local data values. A programmer using the SPMD model has a choice of explicit or implicit means to move data between the cooperating processes. Today, the normal explicit means is via message passing and the implicit means is via data-parallel languages, such as HPF. Although using implicit means to develop parallel applications is generally thought to be easier, explicit message passing is more often used since currently developers can produce more efficient and effective parallel applications using message passing.

A basic prerequisite for message passing is a good communication API. Java comes with various ready-made packages for communication, notably an interface to BSD sockets, and the Remote Method Invocation (RMI) mechanism. Both these communication models are optimized for client-server programming, whereas the parallel computing world is mainly concerned with 'symmetric' communication, occurring in groups of interacting peers.

This symmetric model of communication is captured in the successful Message Passing Interface standard (MPI), established a few years ago[71]. MPI directly supports SPMD model of parallel computing. Reliable point-to-point communication is provided through a shared, group-wide communicator, instead of socket pairs. MPI allows numerous blocking, non-blocking, buffered or synchronous communication modes. It also provides a library of true collective operations (broadcast is the most trivial example). An extended standard, MPI 2, allows for dynamic process creation and access to memory in remote processes.

The existing MPI standards specify language bindings for Fortran, C and C++. In this chapter we discuss a binding of MPI 1.1 for Java, and describe an implementation using Java wrappers to invoke C MPI calls through the Java Native Interface(JNI)[66, 55]. The software is publically available from:

`http://www.npac.syr.edu/projects/pcrc/mpiJava`

The releases of mpiJava include complete source, makefiles, configuration scripts, compiled libraries for WMPI, release test codes (based on the IBM MPI test suite), example applications, javadoc documentation, and installation and usage notes.

In this chapter we describe and then discuss the syntax, functionality and performance of one such interface, `mpiJava`, an object-oriented Java interface to MPI. We first discuss the design of the `mpiJava` API and the issues associated with its development. We then move on to briefly outline the steps necessary to 'port' `mpiJava` onto a range of operating systems, including Windows NT, Linux and Solaris. We also present and then discuss some performance measurements made of communications bandwidth and latency to compare `mpiJava` on these systems.

4.1 Introduction to the `mpiJava` API

The MPI standard is explicitly object-based. The C and Fortran bindings rely on 'opaque objects' that can be manipulated only by acquiring object handles from constructor functions, and passing the handles to suitable functions in the library. The C++ binding specified in the MPI 2 standard collects these objects into suitable class hierarchies and defines most of the library functions as class member functions. The `mpiJava` API follows this model, lifting the structure of its class hierarchy directly from the C++ binding. The major classes of `mpiJava` are illustrated in Figure 4.1.

The class MPI only has static members. It acts as a module containing global services, such as initialization of MPI, and many global constants including the default communicator `COMM_WORLD`.

The most important class in the package is the communicator class `Comm`. All communication functions in `mpiJava` are members of `Comm` or its subclasses. As usual in MPI, a communicator stands for a 'collective object' logically shared by a group of processors. The processes communicate, typically by addressing messages

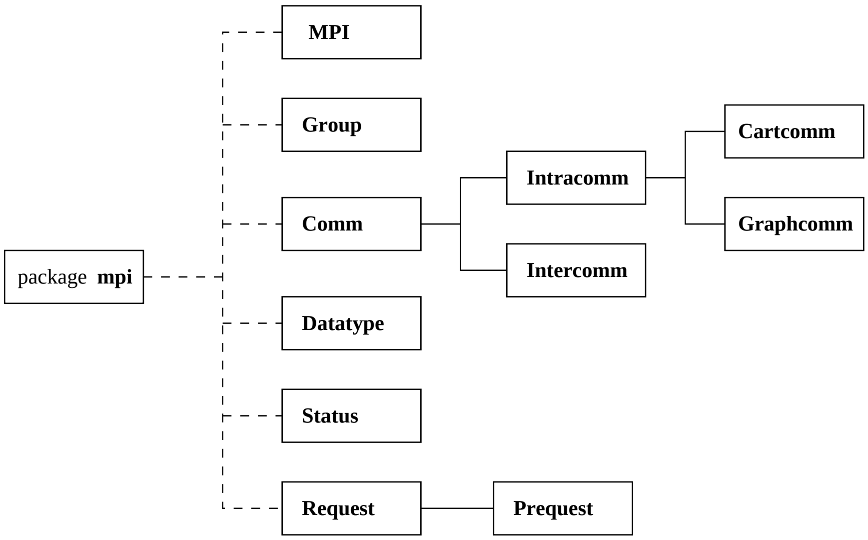

Figure 4.1: Principal classes of `mpiJava`

to their peers through the common communicator.

Another class that is important for the discussion below is the `Datatype` class. This describes the type of the elements in the message buffers passed to send, receive, and all other communication functions. Various basic datatypes are predefined in the package. These mainly correspond to the primitive types of Java, shown in Table 4.1.

The standard send and receive operations of MPI are members of `Comm` with interfaces:

```
public void Send(Object buf, int offset, int count, Datatype
                 datatype, int dest, int tag)


public Status Recv(Object buf, int offset, int count, Datatype
                   datatype, int source, int tag)
```

In both cases the actual argument corresponding to `buf` must be a Java array. In the current implementation they must be arrays with elements of primitive type.

MPI datatype	Java datatype
MPI.BYTE	`byte`
MPI.CHAR	`char`
MPI.SHORT	`short`
MPI.BOOLEAN	`boolean`
MPI.INT	`int`
MPI.LONG	`long`
MPI.FLOAT	`float`
MPI.DOUBLE	`double`
MPI.PACKED	

Table 4.1: Basic datatypes of `mpiJava`

By implication they must be one-dimensional arrays, because Java 'multidimensional arrays' are really arrays of arrays. In these and all other `mpiJava` calls, the buffer array argument is followed by an offset that specifies the element of in array where the message actually starts.

4.1.1 Language Binding

Naming Conventions All MPI classes belong to the package `mpi`. Conventions for capitalization, etc, in class and member names generally follow the C++ MPI bindings.

Data types Opaque objects are presented as Java objects. This introduces the option of simplifying the user's task in managing these objects. MPI destructors can be absorbed into Java object destructors, which are called automatically by the Java garbage collector. We adopt this strategy as the general rule. Explicit calls to MPI destructor functions are typically omitted from the Java user interface (they are absorbed into `finalize` methods). Exceptions are made for the `Comm` and `Request` classes. `MPI_COMM_FREE` is a collective operation, so the user must ensure that calls are made at consistent times by all processors involved—the call can't be left to the vagaries of the garbage collector. A similar case can be made for `MPI_REQUEST_FREE`.

Restrictions on *struct* derived type. Some options allowed for *derived data types* in the C and Fortran binding are deleted in mpiJava. The Java VM does not incorporate a concept of a global linear address space. Passing physical addresses to data type definitions is not allowed. The use of the *MPI_TYPE_STRUCT* datatype constructor is also restricted in a way that makes it impossible to send mixed *basic datatypes* in a single message. Since, however, the set of basic datatypes recognized by MPI is extended to include serializable Java *objects*, this should not be a serious restriction in practice. We will discuss derived type in detail in Section 4.1.2.

Multidimensional arrays and offsets. The C and Fortran languages define a straightforward mapping (or "sequence association") between their multidimensional arrays and equivalent one-dimensional arrays. So in C or Fortran a multidimensional array passed as a message buffer argument is first interpreted as a one-dimensional array with the same element type as the original multidimensional array. Offsets in the buffer (such as offsets occurring in derived data types) are then interpreted in terms of the effective one-dimensional array (or—equivalent up to a constant factor—in terms of physical memory). In Java the relationship between multidimensional arrays and one dimensional arrays is different. An "n-dimensional array" is equivalent to a one-dimensional array of $(n-1)$-dimensional arrays. In mpiJava, message buffers are always one-dimensional arrays. The element type *may* be an object, which *may* have array type. Hence multidimensional arrays can appear as message buffers, but the interpretation is subtly different. In distinction to the C or Fortran case *offsets in multidimensional message buffers are always interpreted as offsets in the outermost one-dimensional array.*

Start of message buffer. C and Fortran both have devices for treating a section of an array, offset from the beginning of the array, as if it was an array in its own right. Java doesn't have any such mechanism. To provide the same flexibility, an `offset` parameter is associated with any buffer argument. This defines the position of the first actual buffer element in the Java array.

Error codes. Unlike the standard C and Fortran interfaces, the mpiJava interfaces to MPI calls do not return explicit error codes. The Java exception mechanism will be used to report errors. The exception mechanism is very widely used by Java libraries. It is inconvenient to use up the single return value of a Java function with an error code. (Java doesn't allow function arguments to be passed by reference, so returning multiple values tends to be more clumsy than in other languages.)

Multiple return values. The `mpiJava` API is modeled as closely as practical on the C++ binding defined in the MPI 2.0 standard (currently we only support the MPI 1.1 subset). A number of changes to argument lists are forced by of the restriction that arguments cannot be passed by reference in Java. In general outputs of `mpiJava` methods come through the result value of the function. In many cases MPI functions return more than one value. This is dealt with in `mpiJava` in various ways. Sometimes an MPI function initializes some elements in an array and also returns a count of the number of elements modified. In Java we typically return an array result, omitting the count. The count can be obtained subsequently from the `length` member of the array. Sometimes an MPI function initializes an object conditionally and returns a separate flag to say if the operation succeeded. In Java we return an object handle which is `null` if the operation fails. Occasionally an extra field is added to an existing MPI class to hold extra results - for example the `Status` class has an extra field, index, initialized by functions like `Waitany`.

Rarely none of these methods work and we resort to defining auxiliary classes to hold multiple results from a particular function.

As a result of these changes `mpiJava` argument lists are often more concise than the corresponding C or C++ argument lists.

Array count arguments. The mpiJava binding often omits array size arguments, because they can be picked up within the function by reading the `length` member of the array argument. A major exception is for message buffers, where an explicit count is always given. In the mpiJava, message buffers have explicit `offset` and `count` arguments whereas other kinds of array argument typically do not. Message buffers aside, typical array arguments to MPI functions (eg, vectors of request structures) are small arrays. If subsections of these must be passed to an MPI function, the sections can be copied to smaller arrays at little cost. In contrast message buffers are typically large and copying them is expensive, so it is worthwhile to pass the extra arguments. Also, if derived data types are being used, the required value of the `count` argument is always different to the buffer length.

Concurrent access to arrays. In JNI-based wrapper implementations it may be necessary to impose some non-interference rules for concurrent read and write operations on arrays. When an array is passed to an MPI method such as a send or receive operation, the wrapper code will probably extract a pointer to the contents of the array using a JNI `Get...ArrayElements` routine. If the garbage collector *does not* support "pinning" (temporarily disabling run-time relocation of data for specific arrays—see [?] for more discussion), the pointer returned by this `Get` function may be to a temporary copy of the elements. The copy will be written back to the true Java array when a subsequent call to `Release...ArrayElements` is made. If two operations involving the same array are active concurrently, this copy-back may

```
// Simple program

import mpi.*;

class Hello {
  static public void main(String[] args){
    MPI.Init(args);
    int myrank = MPI.COMM_WORLD.Rank();
    if(myrank == 0)}
      char [] message = "Hello, there".toCharArray() ;
      MPI.COMM_WORLD.Send(message,0,message.length, MPI.CHAR, 1, 99);
    } else {
      char [] message = new char [20] ;
      MPI.COMM_WORLD.Recv(message, 0, 20, MPI.CHAR, 0, 99) ;
      System.out.println("received:" + new String(message) + ":");

    }
    MPI.Finalize();
  }
}
```

Figure 4.2: Minimal `mpiJava` program (run in two processes)

result in failure to save modifications made by one or more of the concurrent calls.

Such an implementation may have to enforce a safety rule such as: *when several MPI send or receive (etc) operations are active concurrently, if any one of those operations writes to a particular array, none of the other operations must read or write* any portion *of that array.*

If the garbage collector supports pinning, this problem does not arise.

4.1.2 Derived datatype vs Object serialization

In MPI new *derived types* of class `Datatype` can be created using suitable library functions. The derived types allow one to treat contiguous, strided, or indirectly indexed segments of program arrays as individual message elements. The corresponding array subsections can then be communicated in a single function call, potentially exploiting any special hardware or software the platform provides for exchanging scattered data between user space and the communication system.

Currently `mpiJava` provides all the derived datatype constructors of standard MPI, with one limitation: it places significant restrictions on its binding of `MPI_TYPE_STRUCT`. In C or Fortran this function can be used to describe an entity combining fields of different primitive (or derived) type. Because of the assumption that buffers are one-dimensional arrays with elements of primitive type, `mpiJava` imposes a restriction that all the types combined by its `Datatype.Struct` member must have the same *base type*, which must agree with the element type of the buffer array. Also `mpiJava` does not provide an analogue of `MPI_BOTTOM` buffer address, or the `MPI_ADDRESS` function for finding offsets relative to this absolute member base. In C or Fortran these functions allow buffers to include fields from separately declared variables or arrays, but the mechanism does not sit well with the pointer-free Java language model.

Approaches based on the MPI derived datatype model do not seem to be the best way to alleviate this restriction. A better option is probably to exploit the run-time type information already provided in Java objects. We have developed a version of `mpiJava` that adds one new predefined datatype:

```
MPI.Object
```

A message buffer can then be an array of any serializable Java objects. The objects are serialized automatically in the wrapper of send operations, and unserialized at their destination. We will discuss object serialization for Marshalling data in more detail in Chapter 5.

The absence of true multi-dimensional arrays in Java limits another use of derived data types. In MPI the `MPI_TYPE_VECTOR` function creates a derived datatype representing a strided section of an array. In C or Fortran this strided section can be identified with a section of a multi-dimensional array. It could describe, say, an edge of the local patch of a two-dimensional distributed array. In Java there is no equivalence between a multi-dimensional array and a contiguous patch of memory, or a one-dimensional array. The programmer may choose to linearize all multi-dimensional arrays in the algorithm, representing them as one-dimensional arrays with suitable index expressions. In this case derived datatypes can be used to send and receive sections of the array. Alternatively the programmer may use Java arrays of arrays to represent multi-dimensional arrays. This simplifies the index arithmetic in the program. Sections of the array are then explicitly copied to one-dimensional buffers for communication. The latter option seems to be more popular with programmers.

Although, for reasons of conformance of with MPI standards, we expect to continue supporting derived datatypes in `mpiJava`, their value in the Java domain is less clear-cut than in C or Fortran. Allowing serializable objects as buffer elements is probably a more powerful facility.

4.2 `mpiJava` implementations

To 'port' `mpiJava`, it is necessary to have a native MPI library, a version of the Java Development Toolkit (JDK) and a C compiler. `mpiJava` consists of two main parts: the MPI Java classes and the C stubs that binds the MPI Java classes to the underlying native MPI implementation. We create these C stubs using JNI - the means by which Java can call and pass parameters to and from a native API, see

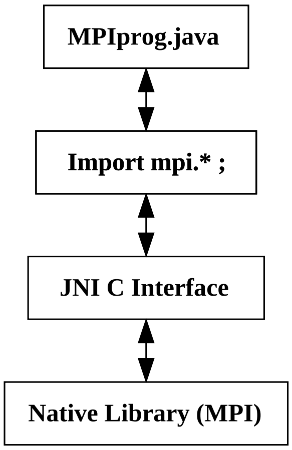

Figure 4.3: Software Layers

next section for detail. Figure 4.3 provides a simple schematic view of the software layers involved.

To 'port' `mpiJava` onto a new platform, generally two steps need to be undertaken:

- Create a native library out of the compiled JNI C stubs.
- Compile MPI Java in class libraries - ensuring that the correctly named stub library is loaded by the Java `System.loadLibrary("stublib")` call in the main source file `MPI.java`.

4.2.1 UNIX Platforms

The development and testing of `mpiJava` was undertaken on various Sun and SGI UNIX platforms using MPICH. As interfacing Java to MPI is not always trivial, in earlier implementation we often saw low-level conflicts between the Java runtime and the interrupt mechanisms used in the MPI implementations. The situation is improving as the JDK matures, in particular version 1.2 allows the use of *green* or *native* threads.

In JDK 1.1, Java threads are implemented on Solaris with a user-level thread package known as green threads. In JDK 1.2, green threads are the default, but the developer can optionally use Solaris native threads for Java threads. With the green threads, a Java program creates and controls multiple threads executing within a single CPU. On the other hand, native threads make it possible for Java programs to take advantage of multiple CPUs on a single machine, are more scalable, and work better with third-party native methods. These new *native* thread feature has eliminated the interrupt problem that we encountered with earlier releases of the JDK. `mpiJava` is now stable on UNIX platforms using MPICH and JDK 1.2 .

4.2.2 PC Platforms

As Java is a platform-neutral language there is much interest in 'porting' `mpiJava` to PC-based systems, in particular Windows NT and Linux.

4.2.2.1 Windows NT

`mpiJava` is stable on NT platforms using WMPI and JDK 1.1 or later. To test `mpiJava` under Windows NT we had the choice of a number of MPI implementations to pick from [14]. We chose WMPI [9] from the Instituto Supererior de Engenharia de Coimbra, Portugal. WMPI is a full implementation of MPI for Microsoft Win32 platforms. WMPI is based on MPICH and includes a p4 [22] device standard. P4 provides the communication internals and a startup mechanism. The WMPI package is a set of libraries (for Borland C++, Microsoft Visual C++ and Microsoft Visual FORTRAN). The release of WMPI provides libraries, header files, examples and daemons for remote start-up. WMPI can co-exist and interact with MPICH/ch_p4 in a cluster of mixed UNIX and Win32 platforms. WMPI is still under development and is freely available.

mpiJava under WMPI: To create a release of `mpiJava` for WMPI the following steps were undertaken:

- *Step 1* - Compile the `mpiJava` JNI C interface into a Win32 Dynamic Link Library (`mpiJava.dll`).
- *Step 2* - Modify the name of the library loaded by the `mpiJava` interface (`MPI.java`) to that of the of the newly compiled library.
- *Step 3* - Compile the Java MPI interface into class libraries.
- *Step 4* - Create a JNI interface to WMPI. This was necessary as under WMPI a master process is first spawned. Its purpose is to first read in a job configuration file and use the information within it to set up and run the actual MPI processes. An idiosyncrasy of WMPI is that all MPI processes must have a file name with the extension *.EXE*. This led to the need to produce a JNI to WMPI so that the JVM was loaded and the 'main' method of `mpiJava` Java class started.

4.2.2.2 Linux

With previous version of JDK1.1.7 for Linux from Blackdown [5], we had problems similar to those encountered during our early attempts to create the interface on Solaris, mentioned in section 4.2.1 . Sun's releases the JDK for Solaris and NT platforms first. On other platforms, such as Linux, it is necessary for developers to 'port' the JDK.

`mpiJava` is now stable on Linux platforms using MPICH and Blackdown's JDK1.2pre-v2 and IBM's JDK1.1.6 . The IBM alpha-version 1.1.6 of JDK [3] for Linux was released in 1999. IBM JDK1.1.6 uses *native* threads as the underlying implementation. Blackdown JDK1.2 supports both *green* and *native* threads.

Both Blackdown and IBM JDK require Linux kernel version 2.2 and glibc version 2.1 or greater. LinuxThreads is the Linux POSIX threads implementation package built into glibc2 which is the GNU C runtime library version 2. In version 2.0 of glibc, LinuxThreads did not have all the functionality required by the software. Version 2.1 of glibc has this required functionality, but some of this functionality is also dependent on the 2.2 version of the Linux kernel.

We have tested `mpiJava` with Mandrake 6.0 [6] and Redhat 6.0 [8] Linux. These Linux packages incorporate both glibc version 2.1 and Linux kernel version 2.2.5 .

4.2.3 Functionality Tests

An integral part of the development of this project was to produce or translate a number of basic MPI test codes to `mpiJava`. An obvious starting point was the C test suite originally developed by IBM . This suite had been modified to comply fully with the MPI standard and to be compatible with the MPICH. The suite consists of fifty-seven C programs that test the following MPI calls and data types; collective operations, communicators, data types, environmental inquiries, groups, point to point and virtual topologies. These codes were all translated to `mpiJava`.

Under WMPI and Solaris-MPICH these codes were run either as multiple processes on a single machine (Shared Memory mode - SM) or as multiple processes running on separate machines (Distributed Memory mode - DM). Under WMPI and Solaris-MPICH all the codes ran in both modes without alterations.

4.3 Simple Communications Performance Measurement

4.3.1 Introduction

At this early stage of our project we have decided to restrict performance measurements to those that will give some indication of the basic inter-processor communications performance. The actual computational performance of each process is felt to be dependent on the local JVM and associated technologies used by specific vendors to increase the performance of Java. A real application performance will be presented in Section 6.4.

4.3.2 PingPong Communications Performance Tests

In this program increasing sized messages are sent back and forth between processes - this is commonly called PingPong. This benchmark is based on standard blocking `MPI_Send/MPI_Recv`. PingPong provides information about latency of `MPI_Send/MPI_Recv` and uni-directional bandwidth. To ensure that anomalies in message timings are minimized the PingPong is repeated many times for each message size. The codes used for these tests were those developed by Baker and Grassl [7]. The three existing codes (MPI-C, MPI-Fortran and Winsock-C) were used for comparison and we implemented an `mpiJava` version for our purposes.

The main problem encountered running the PingPong code was that under WMPI on Win32 `MPI_Wtime()` had been implemented with a millisecond resolution. It was necessary to adapt each of the codes to use an alternative timer with microsecond (μs) resolution. The performance tests shown in the next section were run on two similar systems:

- Two dual processor (P6 200 MHz) NT 4 workstations each with 128 MBytes of DRAM.

- Two dual processor (UltraSparc 200 MHz) Solaris workstations with 256 MBytes of DRAM.

Both systems were connected via 10BaseT Ethernet and the tests were carried out when there was little network activity and on quiet machines.

4.3.3 Message Startup Latencies

In Table 4.2 we show the transmission time in microseconds (s) to send a 1 byte message in each of the environments tested. In SM the `mpiJava` wrapper adds an extra 94μs (140%) and 226μs (152%) compared to WMPI and MPICH C respectively. In DM the `mpiJava` wrapper adds an extra 66μs (11%) and 282μs (42%) compared to WMPI and MPICH C respectively. The Wsock figures are those for a WinSock implementation of PingPong benchmark using TCP.

4.3.4 Results in Shared Memory Mode (Figure 4.4)

The `mpiJava` curve mirrors that of C with an almost constant offset up to 8K, thereafter the curves converge meeting at 256K. Under MPICH, the curves for C and `mpiJava` mirror each other in a similar fashion to those under WMPI, again there is a constant offset and convergence at around 256K.

	Wsock	**WMPI-C**	**WMPI-J**	**MPICH-C**	**MPICH-J**	**Linux-C**	**Linux-J**
SM	144.8 μs	67.2 μs	161.4 μs	148.7 μs	374.6 μs	- μs	- μs
DM	244.9 μs	623.9 μs	689.7 μs	679.1 μs	961.2 μs	- μs	- μs

Table 4.2: Time for 1 Byte Messages

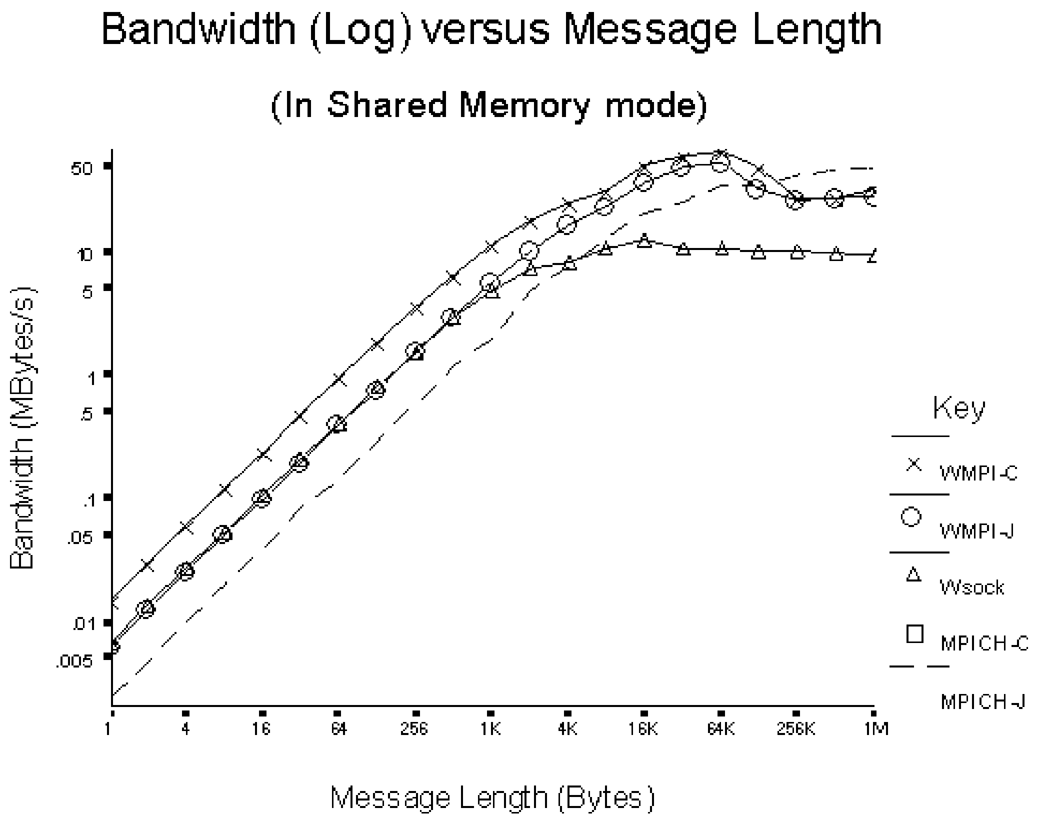

Figure 4.4: PingPong Results in Shared Memory (SM) mode

Under WMPI the peak bandwidth of C is around 65 MBytes/s and `mpiJava` is 54 MBytes/s. The peaks occur at around 64K. Under MPICH the bandwidth is flattening out, but still increasing for C and `mpiJava`, at the 1M. The actual rate measured at this point is about 50 MBytes/s.

Clearly the WMPI C code perform best of those tested. The performance of `mpiJava` in SM under WMPI is good - it exhibits a fairly constant overhead of 95μs up to 2K, thereafter it converges with the C curve. The performance the C code under MPICH is slightly surprising as the NT and Solaris platforms used for these tests had similar specifications. It is assumed that the performance reflects the usage of MPICH rather than a native version of MPI for Solaris. Even so, the MPICH results for `mpiJava` show that it exhibits reasonable performance.

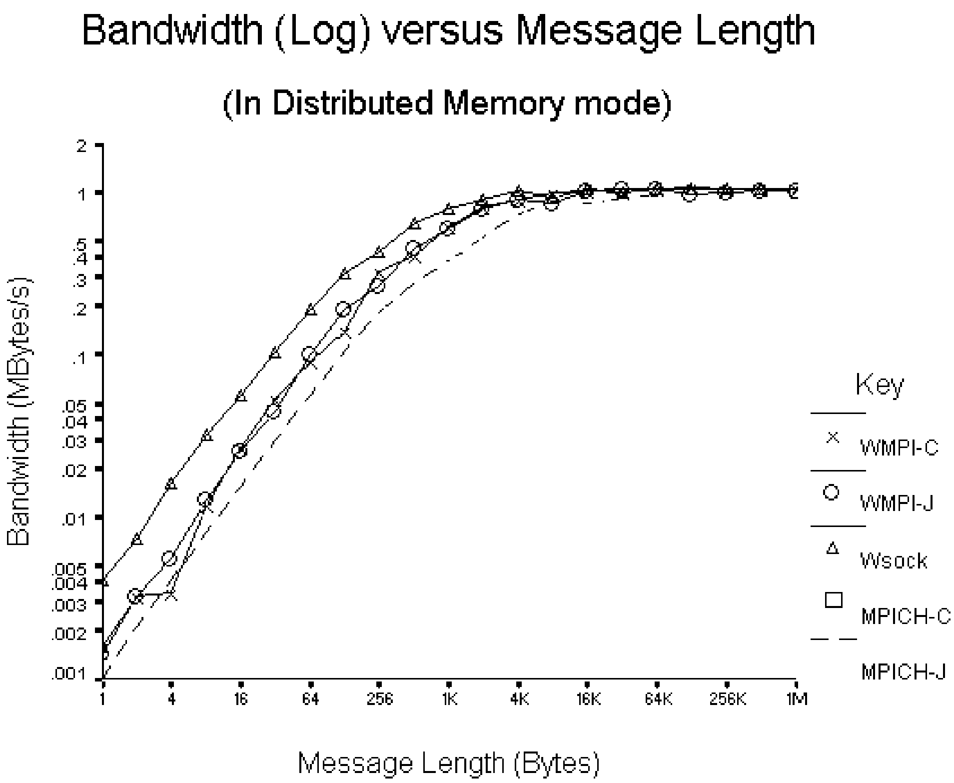

Figure 4.5: PingPong Results in Distributed Memory (DM) mode

4.3.5 Results in Distributed Memory Mode (Figure 4.5)

In DM the differences between the MPI codes is not as pronounced as seen in SM. Under WMPI the C and `mpiJava` codes display very similar performance characteristics throughout the range tested. Under MPICH, there is distinct performance difference between C and `mpiJava`, However the difference is much smaller than in SM and the curves converge at the 4K. All curves peak at about 1 MByte/s, which is about 90% of the maximum attainable on 10 Mbps Ethernet link.

4.3.6 Overall Results Discussion

In both SM and DM modes `mpiJava` adds a fairly constant overhead compared to normal native MPI. In an environment like WMPI, which has been optimized for

NT, the actual overheads of using `mpiJava` are relatively small at around 100ms. Under MPICH the situation is not quite so good, here the use of `mpiJava` introduces an extra overheads of between 250 - 300μs.

It should be noted that these results compare codes running directly under the operating system with those running in the JVM. For example, according to a single 200 MHz PentiumPro will achieve in excess of 62 Mflop/s on a Fortran version of LinPack. A test of the Java LinPack code gave a peak performance of 22 Mflop/s for the same processor running the JVM. The difference in performance will account for much of the additional overhead that `mpiJava` imposes on C MPI codes. From this it can be deduced that the quality and performance of JVM on each platform will have the greatest effect on the usefulness of `mpiJava` for scientific computation.

Chapter 5

Object Serialization for Marshalling Data in the mpiJava

Message buffers have usually been restricted to arrays with elements of primitive type. In this chapter we discuss adoption of the Java object serialization model for marshalling general communication data in MPI-like APIs. This approach is compared with a Java transcription of the standard MPI derived datatype mechanism. We describe an implementation of the *mpiJava* interface to MPI that incorporates automatic object serialization. Benchmark results confirm that current JDK implementations of serialization are not fast enough for high performance messaging applications. Means of solving this problem are discussed, and benchmarks for greatly improved schemes are presented.

Section 5.3 describes an implementation of automatic object serialization in mpiJava. In section 5.4 we discuss benchmarks for this initial implementation. The results confirm that naive use of existing Java serialization technology does not provide the performance needed for high performance message passing environments. Section 5.5 illustrates how various overheads of serialization can be eliminated by customizing the object serialization stream classes. The final draws some conclusion.

5.1 Motivation

Over the past two years several groups have independently developed Java bindings to MPI and Java implementations of MPI subsets. With support of several groups working in the area, the Java Grande Forum drafted an initial proposal for a common MPI-like API for Java [24].

A characteristic feature of MPI is its flexible method for describing message buffers containing mixed primitive fields scattered, possibly non-contiguously, over the local memory of a processor. These buffers are described through special objects called *derived datatypes*—run-time analogues of the user-defined types supported by modern procedural languages. The standard MPI approach does not map very naturally into Java. In previous section 4.1.2 we suggested a Java-compatible restriction of the general MPI derived datatype mechanism, in which all primitive elements of a message buffer have the same type, and they are selected from the elements of a one-dimensional Java array passed as the buffer argument. This approach preserves some of the functionality of the original MPI mechanism—for example the ability to describe strided sections of a one dimensional buffer argument, and to represent a subset of elements selected from the buffer argument by an indirection vector. But it does not allow description of buffers containing elements of mixed primitive types.

This version of the MPI derived datatype mechanism was retained in the initial draft of [24], but its value is not yet certain. A more promising approach may be the addition a new basic datatype to MPI representing a serializable object. The buffer array passed to communication functions is still a one-dimensional array, but as well as allowing arrays with elements of primitive type, the element type is allowed to be `Object`. The serialization paradigm of Java can be adopted to transparently serialize buffer elements at source and unserialize them at destination. An immediate application is to multidimensional arrays. A Java multidimensional array is an array of arrays, and an array is an object. Therefore a multidimensional array is a one-dimensional array of objects and it can be passed directly as a buffer array. The options for representing sections of such an array are limited, but at least one can communicate whole multidimensional arrays without explicitly copying them (though there may be copying inside the implementation).

5.2 Datatypes in an MPI-like API for Java

The MPI standard is explicitly object-based. The C++ binding specified in the MPI 2 standard collects these objects into suitable class hierarchies and defines most of the library functions as class member functions. The Java API proposed in [24] follows this model, and lifts its class hierarchy directly from the C++ binding of MPI.

In our Java version a class `MPI` with only static members acts as a module containing global services, such as initialization of the message-passing layer, and many global constants including a default communicator `COMM_WORLD` It has been pointed out that if multiple MPI threads are allowed in the same Java VM, the default communicator cannot be obtained from a static variable. The final version of the API may change this convention. The communicator class `Comm` is the single most important class in MPI. All communication functions are members of `Comm` or its subclasses. Another class that is relevant for the discussion below is the `Datatype` class. This describes the type of the elements in the message buffers passed to send, receive, and other communication functions. Various basic datatypes are predefined in the package. These mainly correspond to the primitive types of Java, shown in figure 5.1.

The methods corresponding to standard send and receive operations of MPI are members of `Comm` with interfaces

MPI datatype	Java datatype
MPJ.BYTE	`byte`
MPJ.CHAR	`char`
MPJ.SHORT	`short`
MPJ.BOOLEAN	`boolean`
MPJ.INT	`int`
MPJ.LONG	`long`
MPJ.FLOAT	`float`
MPJ.DOUBLE	`double`
MPJ.OBJECT	`Object`

Figure 5.1: Basic datatypes in proposed Java binding

```
void send(Object buf, int offset, int count,
          Datatype datatype, int dst, int tag)


Status recv(Object buf, int offset, int count,
            Datatype datatype, int src, int tag)
```

In both cases the *actual* argument corresponding to `buf` must be a Java array with element type compatible with the `datatype` argument. If the specified type corresponds to a primitive type, the buffer must be a one-dimensional array. Multidimensional arrays can be communicated directly if an object type is specified, because an individual array can be treated as an object. Communication of object types implies some form of serialization and unserialization. This could be the built-in serialization provided in current Java environments, or (as we discuss at length in section 5.5) it could be some specialized serialization tuned for message-passing.

Besides object types the draft Java binding proposal retains a model of MPI derived datatypes. In C or Fortran bindings of MPI, derived datatypes have two roles. One is to allow messages to contain mixed types. The other is to allow noncontiguous data to be transmitted. The first role involves using the `MPI_TYPE_-STRUCT` derived data constructor, which allows one to describe the physical layout of, say, a C *struct* containing mixed types. This will not work in Java, because Java does not expose the low-level layout of its objects. In C or Fortran `MPI_TYPE_STRUCT` also allows one to incorporate displacements computed as differences between absolute addresses, so that parts of a single message can come from separately declared arrays and other variables. Again there is no very natural way to do this in Java. (But effects similar to these uses of `MPI_TYPE_STRUCT` can be achieved by using `MPJ.OBJECT` as the buffer type, and relying on object serialization.)

We conclude that in the Java binding the first role of derived dataypes should probably be abandoned—derived types can only include elements of a single basic type. This leaves description of noncontiguous buffers as the remaining role for derived data types. Every derived data type constructable in the Java binding has a uniquely defined *base type*. This is one of the 9 basic types enumerated above. A *derived datatype* is an object that specifies two things: a base type and a sequence of integer displacements. In contrast to the C and Fortran bindings the displacements can be interpreted in terms of subscripts in the buffer array argument, rather than as byte displacements.

For example the type constructor `indexed` is a member of `Datatype` with interface

```
Datatype indexed(int [] arrayOfBlocklengths,
                 int [] arrayOfDisplacements)
```

This is a binding of the standard MPI operation `MPI_TYPE_INDEXED`. It constructs a new datatype representing replication of the original datatype (to which the method is applied) into a sequence of blocks. Each block can contain a different number of copies and have a different displacement. The base type of the new datatype will be the same as the base type of the original type. If the displacement sequence of the original type was

$$\{disp_0, \ldots, disp_{n-1}\}$$

with extent[1] ex, and `B` is `arrayOfBlocklengths` argument and `D` is `arrayOfDisplacements` argument, the new datatype will have displacement sequence

$$
\begin{aligned}
\{\ & disp_0 + \mathtt{D}[0] \cdot ex, \ldots, disp_{n-1} + \mathtt{D}[0] \cdot ex, \\
& disp_0 + (\mathtt{D}[0] + 1) \cdot ex, \ldots, disp_{n-1} + (\mathtt{D}[0] + 1) \cdot ex, \\
& \ldots, \\
& disp_0 + (\mathtt{D}[0] + \mathtt{B}[0] - 1) \cdot ex, \ldots, \\
& \qquad\qquad disp_{n-1} + (\mathtt{D}[0] + \mathtt{B}[0] - 1) \cdot ex, \\
& \\
& \ldots, \\
& \\
& disp_0 + \mathtt{D}[c-1] \cdot ex, \ldots, disp_{n-1} + \mathtt{D}[c-1] \cdot ex, \\
& disp_0 + (\mathtt{D}[c-1] + 1) \cdot ex, \ldots, \\
& \qquad\qquad disp_{n-1} + (\mathtt{D}[c-1] + 1) \cdot ex, \\
& \ldots, \\
& disp_0 + (\mathtt{D}[c-1] + \mathtt{B}[c-1] - 1) \cdot ex, \ldots, \\
& \qquad\qquad disp_{n-1} + (\mathtt{D}[c-1] + \mathtt{B}[c-1] - 1) \cdot ex\ \}
\end{aligned}
$$

Here, c is the number of blocks.

In Java the derived dataype constructed by `indexed` has a potentially useful role. It allows to send (or receive) messages containing values scattered randomly in some one-dimensional array. The draft proposal incorporates versions of other type constructors from MPI including `MPI_TYPE_VECTOR` for strided sections. We note, though, that the value of providing strided sections is reduced because Java has no natural mapping between elements of its multidimensional arrays and elements of equivalent one-dimensional arrays. This thwarts one common use of strided sections, for representing portions of multidimensional arrays.

[1]The *extent* of a dataype is a measure of the distance between its smallest and largest displacement.

5.3 Adding serialization to the API

In this section we will discuss the other option for representing complex data buffers in the Java API of [24]—introduction of an `MPJ.OBJECT` datatype.

It is natural to assume that the elements of buffers passed to `send` and other output operations are objects whose classes implement the `Serializable` interface. There are at least two ways one may consider communicating object types in the MPI interface

1. Use the standard `ObjectOutputStream` to convert the object buffers to byte vectors, and communicate these byte vectors using the same method as for primitive byte buffers (for example, this might involve a native method call to C MPI functions). At the destination, use the standard `ObjectInputStream` to rebuild the objects.

2. Replace naive use of serialization streams with more specialized code that uses platform-specific knowledge to communicate data fields efficiently. For example, one might modify the standard `writeObject` in such a way that a native method creates an MPI derived datatype structure describing the layout of data in the object, and this buffer descriptor could be passed to a native `MPI_Send` function.

In the second case our implementation is responsible for prepending a suitable type descriptor to the message, so that objects can be reconstructed at the receiving end before data is copied to them.

The first implementation scheme is more straightforward, and this approach will be considered in the remainder of this section. We discuss an implementation based on the *mpiJava* wrappers, combining standard JDK object serialization methods with a JNI interface to native MPI. Benchmark results presented in the next section suggest that something like the second approach (or some suitable combination of the two) deserves serious consideration, hence section 5.5 describes one realization of this scheme.

The original version of mpiJava was a direct Java wrapper for standard MPI. Apart from adopting an object-oriented framework, it added a modest amount of code to the underlying C implementation of MPI. Derived datatype constructors, for example, simply called the datatype constructors of the underlying implementation and returned a Java object containing a representation of the C handle. A `send` operation or a `wait` operation, say, dispatched a single C MPI call. Even exploiting standard JDK object serialization and a native MPI package, uniform support for the `MPJ.OBECT` basic type complicates the wrapper code significantly.

In the new version of the wrapper, every send, receive, or collective communication operation tests if the base type of the datatype argument describing a buffer is `OBJECT`. If not—if the buffer element type is a primitive type—the native MPI operation is called directly, as in the old version. If the buffer is an array of objects, special actions must be taken in the wrapper. If the buffer is a send buffer, the objects must be serialized. We *also* support MPI-like derived datatypes as described in the previous section. On grounds of uniformity, these should be definable with base type `OBJECT`, just as for primitive elements. The message is then some subset of the array of objects passed in the buffer argument, selected according to the displacement sequence of the derived datatype. In the implementation, a method

```
byte [] Object_Serialize(Object   buf,
                         int      offset,
                         int      count,
                         Datatype type)
```

takes the send buffer and descriptor, and returns a byte vector containing the serialized data. At the receiving end a corresponding `Object_deserialize` method is called. Making the Java wrapper responsible for handling derived data types when the base type is `OBJECT` requires additional state in the Java-side `Datatype` class. In particular the Java object explicitly maintains the displacement sequence as an array of integers.

A further set of changes to the implementation arises because the size of the serialized data is not known in advance, and cannot be computed at the receiving end from type information available there. Before the serialized data is sent, the size of the data must be communicated to the receiver, so that a byte receive buffer can be allocated. We send two physical messages—a header containing size information, followed by the data[2]. This, in turn, complicates the implementation of the various `wait` and `test` methods on communication request objects, and the `start` methods on persistent communication requests, and ends up requiring extra fields in the Java `Request` class. Comparable changes are needed in the collective communication wrappers. A `gather` operation, for example, involving object types is implemented as an `MPI_GATHER` operation to collect all message lengths, followed by an `MPI_GATHERV` to collect possibly different-sized data vectors.

5.4 Benchmark results for multidimensional arrays

For the sake of concrete discussion we will make an assumption that, in the kind of *Grande* applications where MPI is likely to be used, some of the most pressing performance issues concern arrays and multidimensional arrays of small objects—

[2]A better protocol would be to eagerly send data for short messages in the header, assuming some fixed-size buffer is preallocated at the receiving end. The two-message protocol would be reserved for long messages. This marginally complicates the implementation but does not essentially change the rest of the discussion, or the benchmark results presented below, since the latter concentrate on the asymptotic case. We are grateful to one of the referees for raising this point.

especially arrays of primitive elements such as `int`s and `float`s. For benchmarks we therefore concentrated on the overheads introduced by object serialization when the objects contain many arrays of primitive elements. Specifically we concentrated on communication of two-dimensional arrays with primitive elements We note that there some debate about whether the Java model of multidimensional arrays is the most appropriate one for high performance computing. There are various proposals for optimized HPC array class libraries [72]. See Section 5.6 for some further discussion. .

The "ping-pong" method was used to time point-to-point communication of an N by N array of primitive elements treated as a one dimensional array of objects, and compare it with communication of an N^2 array without using serialization. As an intermediate case we also timed communication of a 1 by N^2 array treated as a one-dimensional (size 1) array of objects. This allows us to extract an estimate of the overhead to "serialize" an individual primitive element. The code for sending and receiving the various array shapes is given schematically in Figure 5.2.

As a crude timing model for these benchmarks, one can assume that there is a cost $t^{\mathrm{T}}_{\mathrm{ser}}$ to serialize each primitive element of type `T`, an additional cost $t^{\mathrm{vec}}_{\mathrm{ser}}$ to serialize each subarray, similar constants $t^{\mathrm{T}}_{\mathrm{unser}}$ and $t^{\mathrm{vec}}_{\mathrm{unser}}$ for unserialization, and a cost $t^{\mathrm{T}}_{\mathrm{com}}$ to physically tranfser each element of data. Then the total time for benchmarked communications should be

$$t^{\mathtt{T[}N^2\mathtt{]}} = c + t^{\mathrm{T}}_{\mathrm{com}} N^2 \tag{5.1}$$

$$t^{\mathtt{T[1][}N^2\mathtt{]}} = c' + (t^{\mathrm{T}}_{\mathrm{ser}} + t^{\mathrm{T}}_{\mathrm{com}} + t^{\mathrm{T}}_{\mathrm{unser}}) N^2 \tag{5.2}$$

$$\begin{aligned} t^{\mathtt{T[}N\mathtt{][}N\mathtt{]}} &= c'' + (t^{\mathrm{vec}}_{\mathrm{ser}} + t^{\mathrm{vec}}_{\mathrm{unser}}) N + \\ &\quad (t^{\mathrm{T}}_{\mathrm{ser}} + t^{\mathrm{T}}_{\mathrm{com}} + t^{\mathrm{T}}_{\mathrm{unser}}) N^2 \end{aligned} \tag{5.3}$$

These formulae do not attempt to explain the constant initial overhead, don't take into account the extra bytes for type description that serialization introduces into

N^2 float vector

```
float [] buf = new float [N * N] ;   float [] buf = new float [N * N] ;
MPJ.COMM_WORLD.send(buf, 0, N * N,   MPJ.COMM_WORLD.recv(buf, 0, N * N,
                    MPJ.FLOAT,                           MPJ.FLOAT,
                    dst, tag) ;                          src, tag) ;
```

$N \times N$ float array

```
float [] [] buf = new float [N] [N] ; float [] [] buf = new float [N] [] ;
MPJ.COMM_WORLD.send(buf, 0, N,        MPJ.COMM_WORLD.recv(buf, 0, N,
                    MPJ.OBJECT,                           MPJ.OBJECT,
                    dst, tag) ;                           src, tag) ;
```

$1 \times N^2$ float array

```
float [] [] buf = new float [1] [N * N] ; float [] [] buf = new float [1] [] ;
MPJ.COMM_WORLD.send(buf, 0, 1,            MPJ.COMM_WORLD.recv(buf, 0, 1,
                    MPJ.OBJECT,                               MPJ.OBJECT,
                    dst, tag) ;                               src, tag) ;
```

Figure 5.2: Send and receive operations for various array shapes.

the stream, and ignore possible non-linear costs associated with analysing object graphs, etc. Empirically these effects are small for the range of N we consider.

All measurements were performed on a cluster of 2-processor, 200 Mhz UltraSparc nodes connected through a SunATM-155/MMF network. The underlying MPI implementation was Sun MPI 3.0 (part of the Sun HPC package). The JDK was jdk1.2beta4. Shared memory results quoted are obtained by running two processes on the processors of a single node. Non-shared-memory results are obtained by running peer processes in different nodes.

In a series of measurements, element serialization and unserialization timing parameters were estimated by independent benchmarks of the serialization code. The parameters $t^{\text{vec}}_{\text{ser}}$ and $t^{\text{vec}}_{\text{unser}}$ were estimated by plotting the difference between serialization and unserialization times for `T[1][`N^2`]` and `T[`N`][`N`]` Our timing

$t^{\mathrm{byte}}_{\mathrm{ser}} = 0.043\mu\mathrm{s}$	$t^{\mathrm{float}}_{\mathrm{ser}} = 2.1\mu\mathrm{s}$	$t^{\mathrm{vec}}_{\mathrm{ser}} = 100\mu\mathrm{s}$
$t^{\mathrm{byte}}_{\mathrm{unser}} = 0.027\mu\mathrm{s}$	$t^{\mathrm{float}}_{\mathrm{unser}} = 1.4\mu\mathrm{s}$	$t^{\mathrm{vec}}_{\mathrm{unser}} = 53\mu\mathrm{s}$
$t^{\mathrm{byte}}_{\mathrm{com}} = 0.062\mu\mathrm{s}^{\dagger}$	$t^{\mathrm{float}}_{\mathrm{com}} = 0.25\mu\mathrm{s}^{\dagger}$	
$t^{\mathrm{byte}}_{\mathrm{com}} = 0.008\mu\mathrm{s}^{\S}$	$t^{\mathrm{float}}_{\mathrm{com}} = 0.038\mu\mathrm{s}^{\S}$	

Table 5.1: Estimated parameters in serialization and communication timing model. The $t^{\mathrm{T}}_{\mathrm{com}}$ values are respectively for non-shared memory (†) and shared memory (§) implementations of the underlying communication.

model assumed the values of these parameters is independent of the element type. This is only approximately true, and the values quoted in the table and used in the plotted curves are averages. Separately measured values for `byte` arrays were smaller than these averages, and for `int` and `float` arrays they were larger. The raw communication speed was estimated from ping-pong results for $t^{\mathtt{T[}N^2\mathtt{]}}$. Table 5.1 contains the resulting estimates of the various parameters for `byte` and `float` elements.

Figures 5.3 and 5.4 on Page 86 and 87 plot actual measured times from ping-pong benchmarks for the mpiJava sends and receives of arrays with `byte` and `float` elements. In the plots the array extent, N, ranges between 128 and 1024. The measured times for $t^{\mathtt{T[}N^2\mathtt{]}}$, $t^{\mathtt{T[1][}N^2\mathtt{]}}$ and $t^{\mathtt{T[}N\mathtt{][}N\mathtt{]}}$ are compared with the formulae given above (setting the c constants to zero). The agreement is good, so our parametrization is assumed to be realistic in the regime considered.

According to table 5.1 the overhead of Java serialization nearly always dominates other communication costs. In the worst case—floating point numbers—it takes around 2 microseconds to serialize each number and a smaller but comparable time to unserialize. But it only takes a few hundredths of a microsecond to communicate the word through shared memory. Serialization slows communication

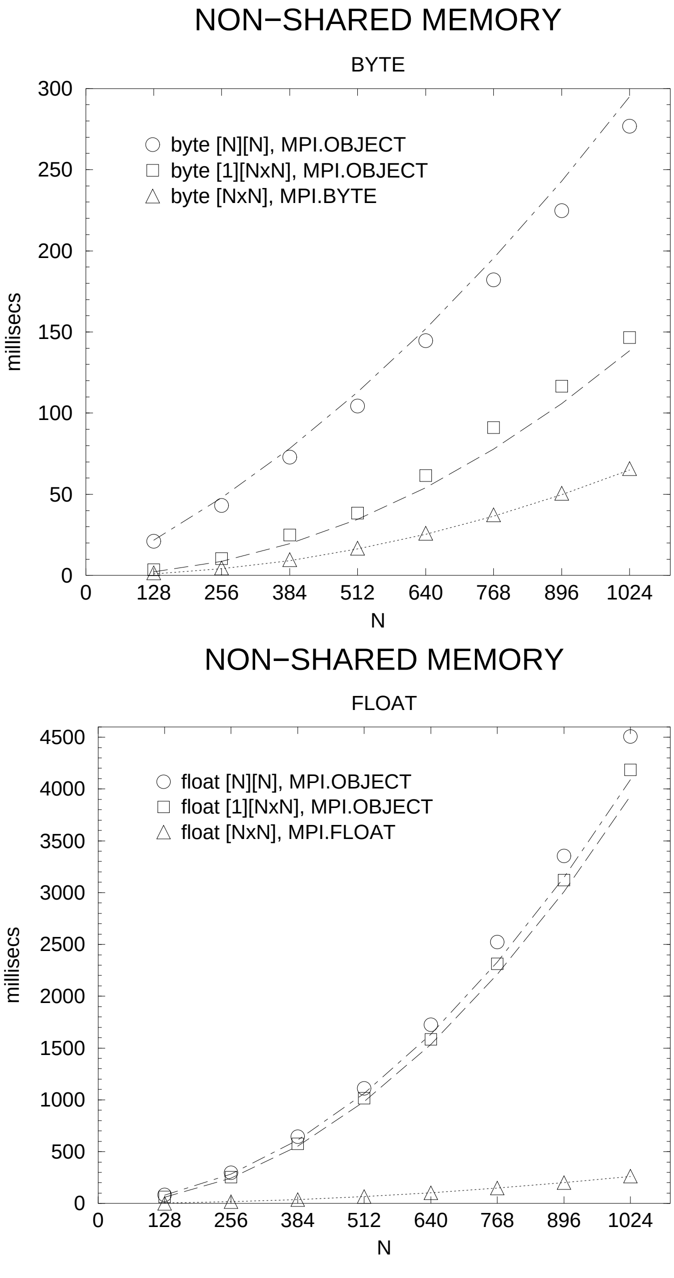

Figure 5.3: Communication times from ping-pong benchmark in non-shared-memory case. The lines represent the model defined by Equations 5.1 to 5.3 in the text, with parameters from Table 5.1.

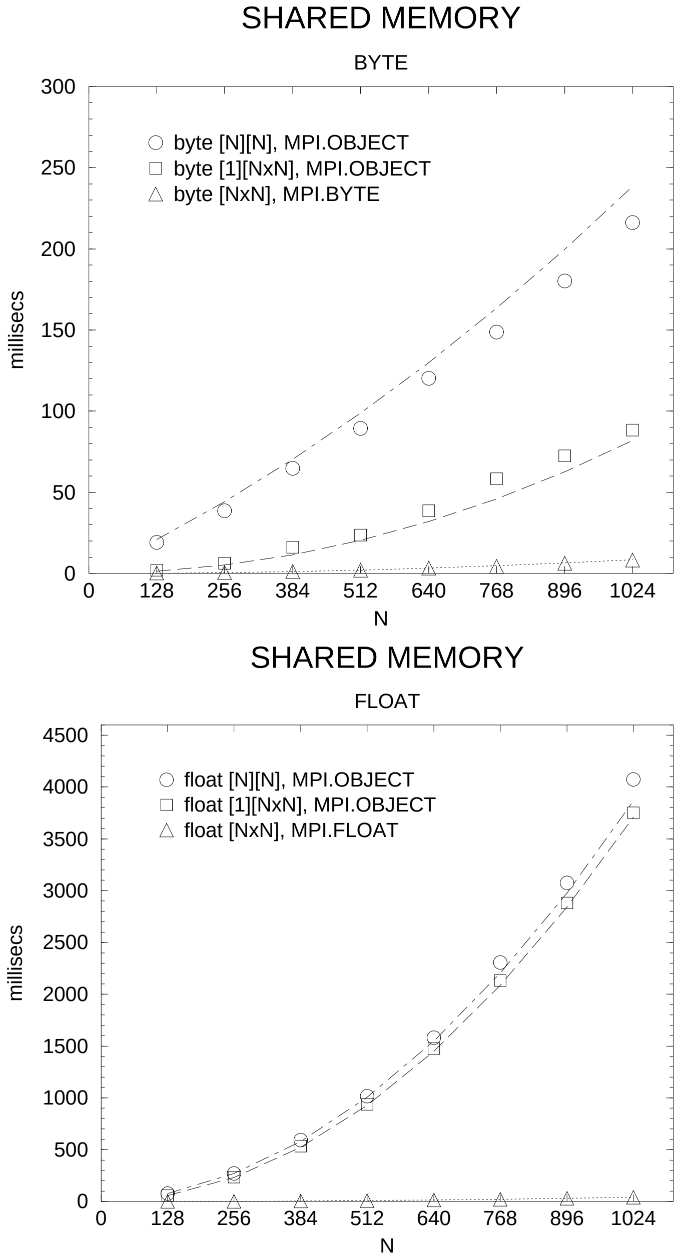

Figure 5.4: Communication times from ping-pong benchmark in shared-memory case. The lines represent the model defined by Equations 5.1 to 5.3 in the text, with parameters from Table 5.1.

by nearly two orders of magnitude. When the underlying communication is over a fast network rather than through shared memory the raw communication time is still only a fraction of a microsecond, and serialization still dominates that time by about one order of magnitude. For `byte` elements serialization costs are smaller, but still larger than the communication costs in the fast network and still much larger than the communication cost through shared memory. Serialization costs for `int` elements are intermediate.

The constant overheads for serializing each subarray, characterized by the parameters $t_{\mathrm{ser}}^{\mathrm{vec}}$ and $t_{\mathrm{unser}}^{\mathrm{vec}}$ are also quite large, although, for the array sizes considered here they only make a dominant contribution for the `byte` arrays, where individual element serialization is relatively fast.

5.5 Reducing serialization overheads for arrays

As seen in Section 2.4.3, the work of [82] and others has established that there is considerable scope to optimize the JDK serialization software. Here we pursue an alternative that is interesting from the point of view of ultimate efficiency in messaging APIs, namely to replace calls to the `writeObject`, `readObject` methods with specialized, MPI-specific, functions. A call to standard `writeObject`, for example, might be replaced with a native method that creates a native MPI derived datatype structure describing the layout of data in the object. This would provide the conceptually straightforward object serialization model at the user level, while retaining the option of fast ("zero-copy") communication strategies inside the implementation.

Implementing this general scheme for every kind of Java object is difficult or impractical because the JVM hides the internal representation of most objects. Less ambitiously, we can attempt to eliminate the serialization and copy overheads for

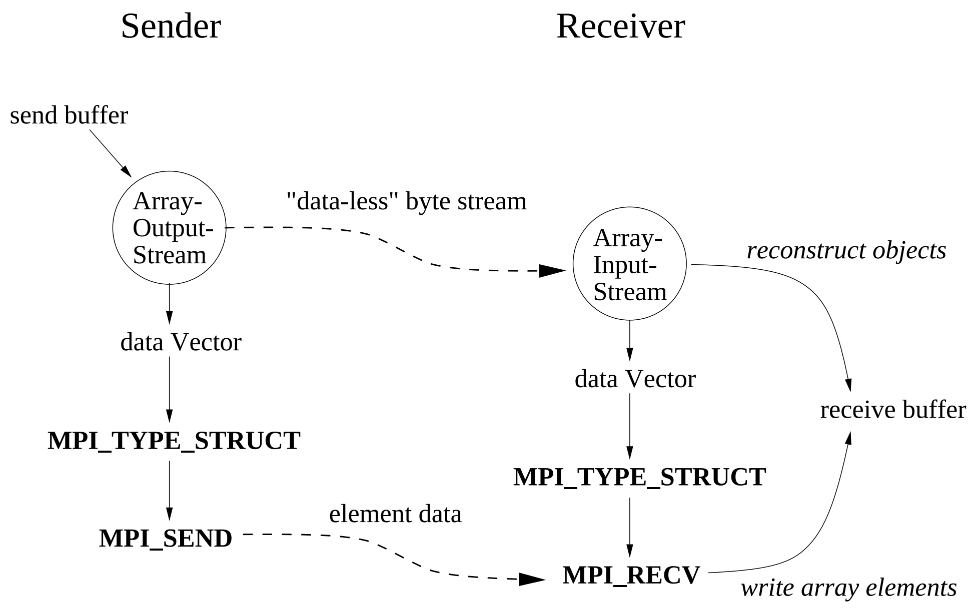

Figure 5.5: Improved protocol for handling arrays of primitive elements.

arrays of primitive elements embedded in the serialization stream. The general idea is to produce specialized versions of `ObjectOutputStream` and `ObjectInputStream` that yield byte streams identical to the standard version *except* that array data is omitted from those streams. The "data-less" byte stream is sent as a header. This allows the objects to be reconstructed at the receiving end. The array data is then sent separately using, say, suitable native `MPI_TYPE_STRUCT` types to send all the array data in one logical communication. In this way the serialization overhead parameters measured in the benchmarks of the previous section can be drastically reduced or eliminated. An implementation of this protocol is illustrated in Figure 5.5.

A customized version of `ObjectOutputStream` called `ArrayOutputStream` behaves in exactly the same way as the original stream except when it encounters an array. When an array is encountered a small object of type `ArrayProxy` is placed in the stream. This encodes the type and size of the array. The array reference itself is placed in a separate container called the "data vector". When serialization is complete, the data-less byte stream is sent to the receiver. A piece of native code unravels the data vector and sets up a native derived type, then the array data is sent. At the receiving end a customized `ArrayInputStream` behaves exactly like an `ObjectInputStream`, except that when it encounters an `ArrayProxy` it allocates an array of the appropriate type and length and places a handle to this array in the reconstructed object graph *and* in a data vector container. When this phase is completed we have an object graph containing uninitialized array elements and a data vector, created as a side effect of unserialization. A native derived data type is constructed from the data vector in the same way as at the sending end, and the data is received into the reconstructed object in a single MPI operation.

Our implementation of `ArrayOutputStream` and `ArrayInputStream` is straightforward. The standard `ObjectOutputStream` provides a method, `replaceObject`, which can be overridden in subclasses. `ObjectInputStream` provides a corresponding `resolveObject` method. Implementation of the customized streams is sketched in Figure 5.6.

Figures 5.7 and 5.8 on Page 92 and 93 show the effect this change of protocol has on the original timings. As expected, eliminating the overheads of element serialization dramatically speeds communication of float arrays (for example) treated as objects, bringing bandwidth close to the raw performance available with `MPJ.FLOAT`.

Each one-dimensional array in the stream needs some separate processing here (associated with calls to `replaceObject`, `resolveObject`, and setting up the native `MPI_TYPE_STRUCT`). Our fairly simple-minded prototype happened to increase the constant overhead of communicating each subarray (parametrized by $t^{\mathrm{vec}}_{\mathrm{ser}}$ and $t^{\mathrm{vec}}_{\mathrm{unser}}$ in the previous section). As mentioned at the end of section 5.4, this overhead typically dominates the time for communicating two-dimensional byte arrays (where the element serialization cost is less extreme), so performance there actually ends up being worse. A more highly tuned implementation could probably reduce

```
class ArrayOutputStream extends ObjectOutputStream {
    Vector dataVector ;

    public Object replaceObject(Object obj) {
        if(obj instanceof int []) {
            dataVector.addElement(obj)
            return new ArrayIntProxy(((int []) obj).length) ;
        }
        ... deal with other primitive array types ...
        else
            return obj
    }
}

class ArrayInputStream extends ObjectInputStream {
    Vector dataVector ;

    public Object resolveObject(Object obj) {
        if(obj instanceof ArrayIntProxy) {
            int dat = new int [((ArrayIntProxy) obj).length] ;
            dataVector.addElement(dat)
            return dat ;
        }
        ... deal with other array proxy types ...
        else
            return obj
    }
}
```

Figure 5.6: Pseudocode for `ArrayOutputStream` and `ArrayInputStream`

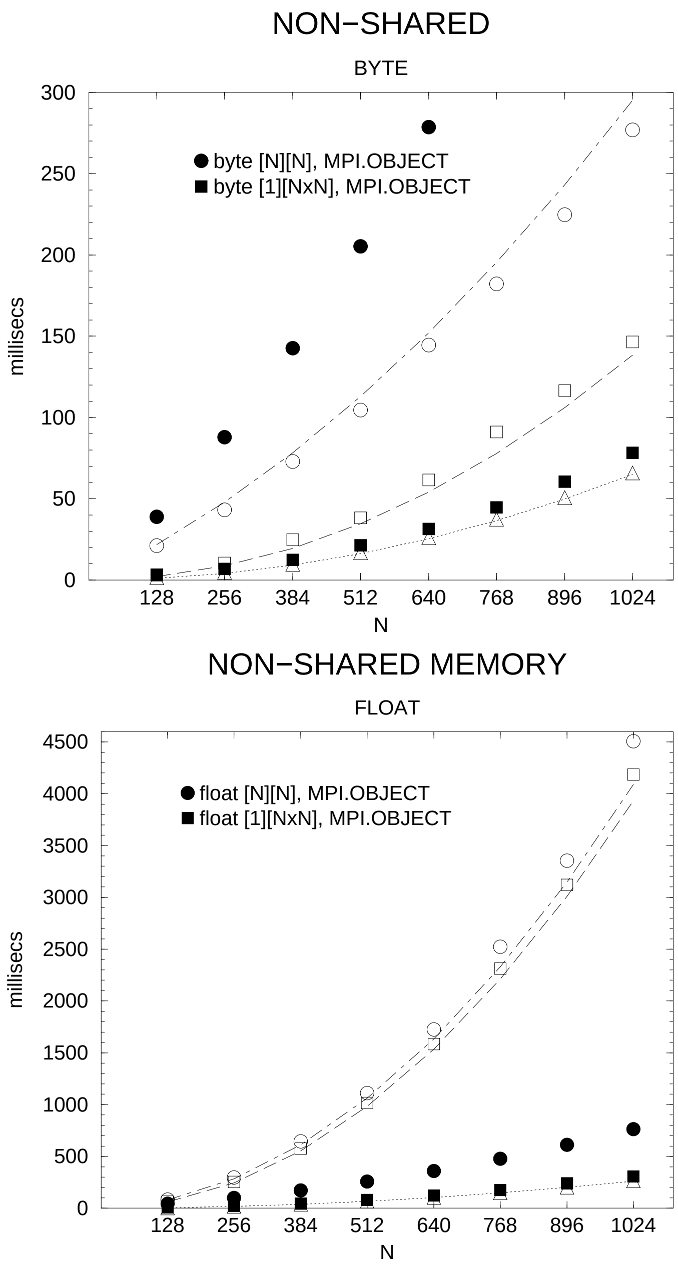

Figure 5.7: Ping-pong timings with primitive array data sent separately (solid points) in distributed memory mode, compared with the unoptimized results from Figure 5.3 (open points). Recall that the goal is to bring times for "object-oriented" sends of arrays down to the "native" send times, most closely approximated by the triangular points.

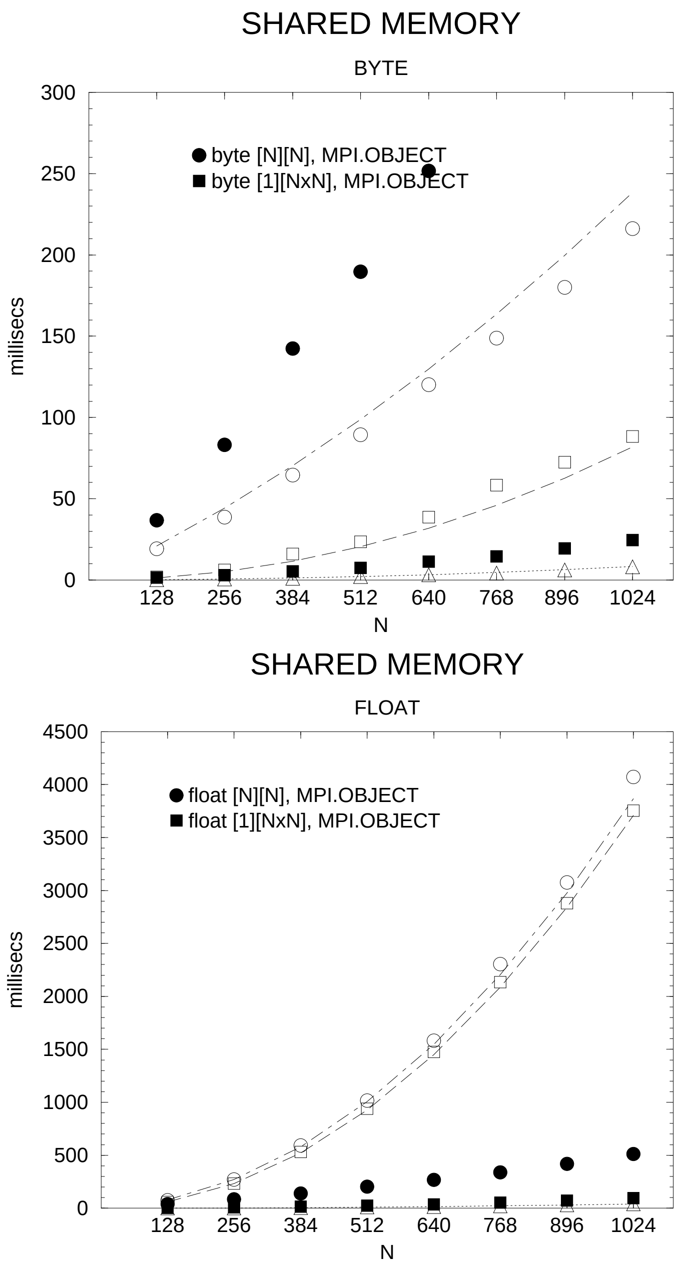

Figure 5.8: Ping-pong timings with primitive array data sent separately (solid points) in shared memory mode, compared with the unoptimized results from Figure 5.4 (open points). Recall that the goal is to bring times for "object-oriented" sends of arrays down to the "native" send times, most closely approximated by the triangular points.

this problem. Alternatively we can go a step further with our protocol, and have the serialization stream object directly replace *two-dimensional* arrays of primitive elements[3]. The benefits of this approach are shown in Figures 5.9 and 5.10 on Page 95 and 96 .

This process could continue almost indefinitely—adding special cases for arrays and other structures considered critical to Grande applications. Currently we do not envisage pushing this approach any further than two-dimensional array proxies. Of course three-dimensional arrays and higher will automaticall benefit from the optimization of their lower-dimensional component arrays. Recognizing a rectangular two-dimensional arrays already adds some unwanted complexity to the serialization process It can also introduce some unexpected behaviour. Our version subtly alters the semantics of serialization, because it does not detect aliasing of rows (either with other rows of the same two-dimensional array, or with one-dimensional primitive arrays elsewhere in the stream). Hence the reconstructed object graph at the receiving end will not reproduce such aliasing. Whether this is a serious problem is unclear.

5.6 Discussion

In Java, the object serialization model for data marshalling has various advantages over the MPI derived type mechanism. It provides much (though not all) of the flexibility of derived types, and is presumably simpler to use. Object serialization provides a natural way to deal with Java multidimensional arrays. Such arrays are likely to be common in scientific programming.

Our initial attempt to add automatic object serialization to our MPI-like API for Java was impaired by poor performance of the serialization code in the current

[3] Defined to be arrays of objects, each element being an array of primitive type of the same type and length.

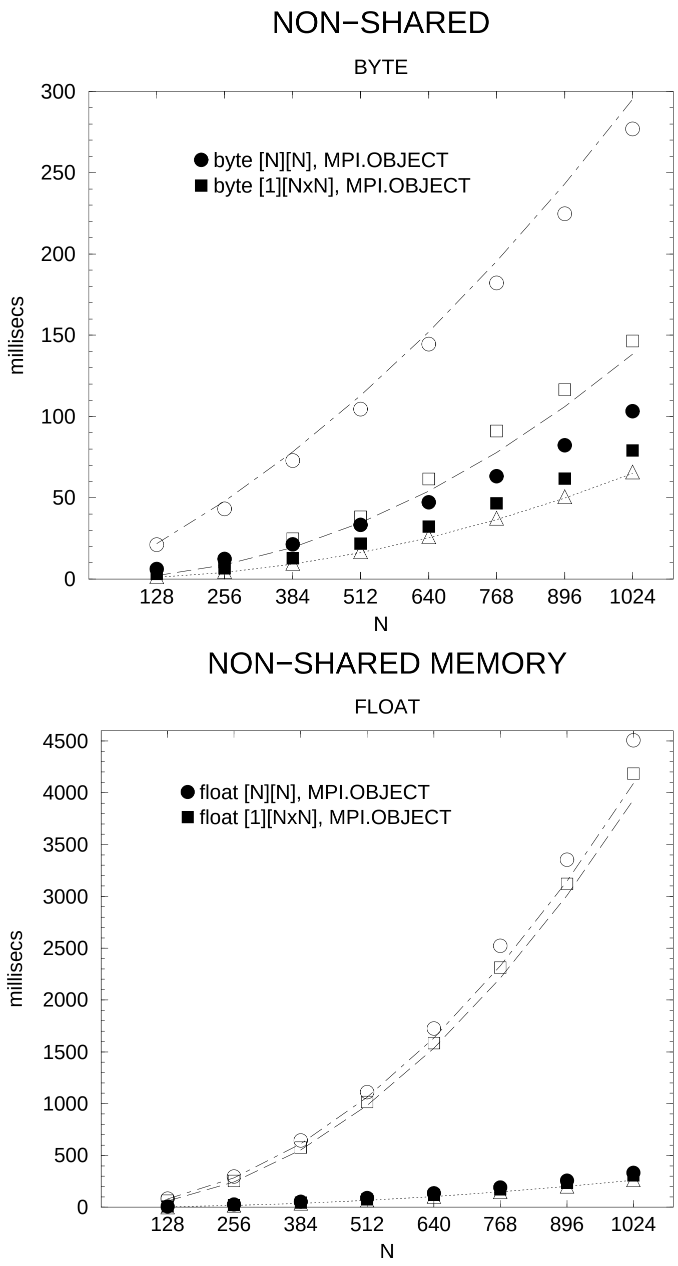

Figure 5.9: Timings allowing *two-dimensional* array proxies in the object stream (solid points) in distributed memory mode, compared with the unoptimized results from Figure 5.3 (open points). Sends of two-dimensional Java arrays (solid circles) are now much closer to the native bandwidth (of which the triangular points are representative).

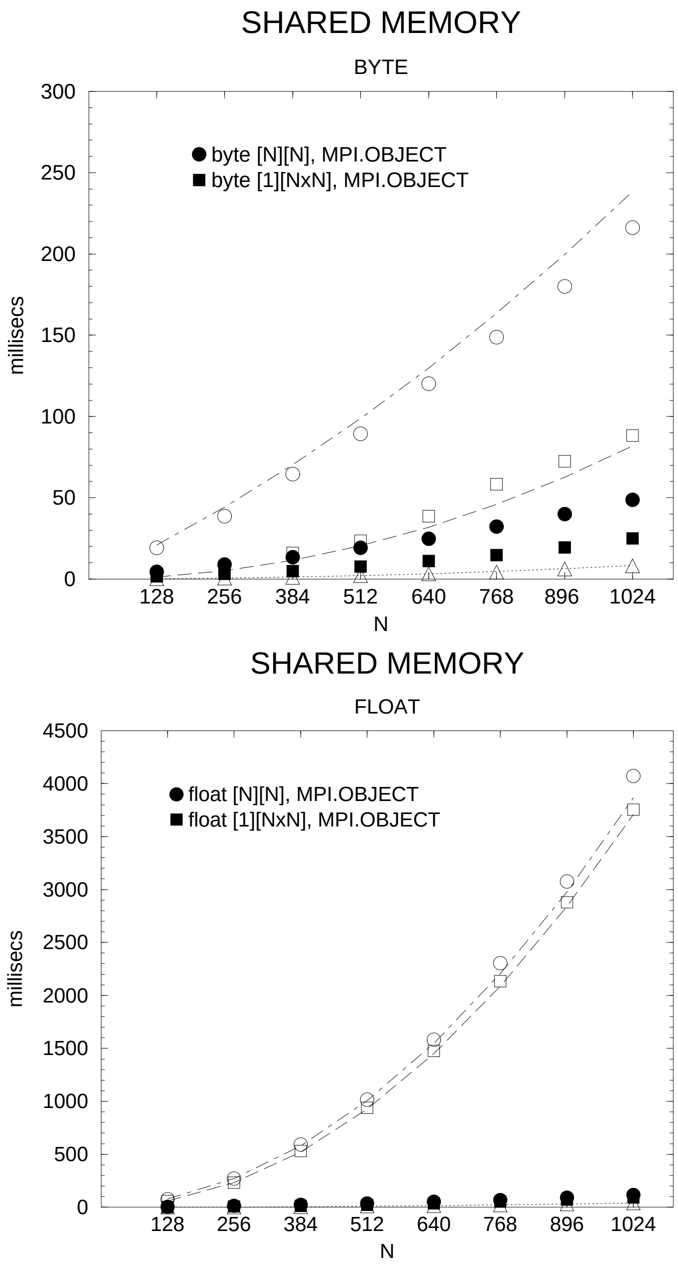

Figure 5.10: Timings allowing *two-dimensional* array proxies in the object stream (solid points) in shared memory mode, compared with the unoptimized results from Figure 5.4 (open points). Sends of two-dimensional Java arrays (solid circles) are now much closer to the native bandwidth (of which the triangular points are representative).

Java Development Kit. Buffers were serialized using standard technology from the JDK. The benchmark results from section 5.4 showed that this implementation introduces very large overheads relative to underlying communication speeds on fast networks and symmetric multiprocessors. Similar problems were reported in the context of RMI implementations in [43]. In the context of fast message-passing environments (not surprisingly) the issue is even more critical. Overall communication performance can easily be downgraded by an order of magnitude or more.

In our benchmarks and tests the organization of primitive elements—their byte-order, in particular—was the same in sender and receiver. This is commonly the case in MPI applications, which are often run on homogenous clusters of computers. Hence it should be possible to send the bytes with no format conversion at all. More generally an MPI-like package can be assumed to know in advance if sender and receiver have different layouts, and need only convert to an external representation in the case that they do. Presuming we are building on an underlying native MPI in the first place, then, a reasonable assumption is that the conversions necessary for, say, communication of float arrays between little-endian and big-endian machines in a heterogenous cluster are dealt with inside the native MPI. This may degrade the effective native bandwidth to a greater or lesser extent, but should not impact the Java wrapper code. In any case, to exploit these features in the native library, we need a way to marshal Java arrays that avoids performing conversions inefficiently in the Java layer.

The standard Java serialization framework allows the programmer to provide optimized serialization and unserialization methods for particular classes, but in scientific programming we are often more concerned with the speed of operations on arrays, and especially arrays of primitive types. The standard Java framework for serialization does not provide a direct way to handle arrays, but in section 5.5 we customized the object streams themselves by suitably defining the `replaceObject`, `resolveObject` methods. Primitive array data was removed from the serialization stream and sent separately using *native* derived datatype mechanisms of the underlying MPI, without explicit conversion or explicit copying. This dramatically reduced the overheads of treating Java arrays uniformly as objects at the API level. Order of magnitude degradations in bandwidth were typically replaced by fractional overheads.

A somewhat different approach was taken by the authors of [74]. Their remote method invocation software, KaRMI, incorporates an extensive reimplemention of the JDK serialization code, to better support their optimized RMI. Their ideas for optimizing serialization can certainly benefit message-based APIs as well, and KaRMI does also reduce copying compared with standard RMI. But we believe they do not immediately support the "zero-copy" strategy we strive for here, whereby large arrays are removed from the serialization stream and dealt with separately by platform-specific software[4]. In our case the platform-specific software was a native MPI binding, but similar strategies could apply to other devices, such as a binding to the new industry standard Virtual Interface Architecture, VIA[5].

Given that the efficiency of object serialization can be improved dramatically—although probably it will always introduce a non-zero overhead—a reasonable question is whether an MPI-like API for Java needs to retain anything like the old derived datatype mechanism of MPI at all?

The MPI mechanism still allows non-contiguous sections of a buffer array to

[4]Our use of the phrase "zero-copy" has been criticized on the basis that a number of existing JVMs *always* copy arrays that are passed through the JNI interface, in which case there is always at least one copy. To our knowledge, there is nothing in the JVM specification that requires such behaviour, and other existing JVMs pin the storage inside the JVM and return a pointer to the actual storage to the native method, rather than copying. But it is true that the phrase zero-copy must be understood modulo the behaviour JNI implementation associated with the JVM and garbage collector that one is using.

[5]We should add that KaRMI can also use specific communication hardware such as VIA for its transport layer, and in principle could even plug in native MPI-routines in this layer. We believe it would nevertheless serialize data first.

be sent directly. Although implementations of MPI derived types, even in the C domain, have often had disappointing performance in the past, we note that VIA provides some low-level support for communicating non-contiguous buffers, and recently there has been interest in producing Java bindings of VIA [28, 95]. So perhaps in the future it will become possible to support derived types quite efficiently in Java. We have emphasized the use of object serialization as a way of dealing with communication of Java multidimensional arrays. Assuming the Java model of multidimensional arrays (as arrays of arrays), we suspect serialization is the most natural way of communicating them. On the other hand there is an active discussion (especially in Numerics Working Group of the Java Grande Forum) about how Fortran-like multidimensional rectangular arrays could best be supported into Java. A reasonable guess is that multidimensional array sections would be represented as strided sections of some standard one-dimensional Java array. In this case the best choice for communicating array sections may come back to using MPI-like derived datatypes similar to `MPI_TYPE_VECTOR`.

In any case—whether or not a version of MPI derived data types survive in Java—the need to support object serialization in a message-passing API seems relatively clear.

Chapter 6

Applications and Performance of mpiJava

We have implemented several SPMD style parallel applications using `mpiJava`. These applications were included in mpiJava software package. In this chapter we discuss two parallel Monte Carlo simulations which can be used for education purpose and experimental performance of mpiJava with a real application.

In first section, we briefly introduce a Java version of parallel random number generator which is used in two Monte Carlo spin model simulations, Ising Model and Potts Model. Then we discuss the `mpiJava` version of Monte Carlo simulation of Ising model implemented with two different algorithms, metropolis and cluster algorithms. This Ising model simulation can be used as a full application Java benchmarks with comparable native MPI-C and MPI-Fortran codes for performance comparison. This is present in the last section. We also present graphical Potts model simulation in Section 6.3. The Potts model is a generalized extension of the Ising model to an arbitrary number, Q, of discrete states, however, it has a much richer phase structure, which makes it a richer graphical visualization.

6.1 Parallel Random Number generator

Random number generators are widely used for simulations in computational science and engineering application. For some Java Grande applications, such Monte Carlo simulations, it is crucial that the random number generator have good randomness properties. This is especially true in large-scale simulations on parallel supercomputers, which consume huge quantities of random numbers, and require parallel algorithms for random number generation. The techniques for testing the quality of parallel random number are presented in [32].

Random number generators use iterative deterministic algorithms for producing for a sequence X_i of pseudo-random numbers that approximate a truly random sequence. Probably the most commonly-used algorithms used for sequential random number generators are Linear Congruential Generators (LCG). LCGs produce a sequence X_i of random integers using the relation

$$X_i = (a * X_{i-1} + b) \mod M, \tag{6.1}$$

where a is known as the multiplier, M is the modulus, and b is an additive constant. If $b = 0$, this termed a Multiplicative Linear Congruential Generators (MLCG). The parameters (a, b, M) must be chosen carefully to ensure a large period and good uniformity and randomness properties. Standard random number generators return a floating point number x_i in the interval [0,1), which can be obtained by dividing X_i by M.

Many different parallel random number generators have been proposed [10, 31], but most of them use the same basic concept, which is to parallelize a sequential generator by taking the elements of the sequence of pseudo-random numbers it generates and distributing them among the processors in some way. One of the method to achieve this goal is to split the sequence into non-overlapping contiguous sections, each of which is generated by a different processor. This also utilizes the fact that this type of generator allows an arbitrary element of the sequence to be calculated. One can therefore simply divide the period of the generator by the number of processors, and jump ahead in the sequence by this amount for each processor. Alternatively, the length of each section of numbers could be chosen much larger than could be used by any processor. If the length of the sections is L, then processor P would generate the sequence

$$X_{PL}, X_{PL+1}, X_{PL+2}, \ldots,$$

This method also requires the ability to jump ahead in the sequence by a given amount.

Another simple method for parallelizing random number generators is for processor P of an N processor machine to generate the sub-sequence

$$X_P, X_{P+N}, X_{P+2N}, \ldots,$$

so that the sequence is spread across processors in the same way as a deck of cards is dealt in turn to players in a card game. This is known as the leapfrog technique, since each processor leap-frogs by N in the sequence. In order to use this method we need to be able to easily jump ahead in the sequence. Linear congruential generators are simple enough that it is possible to specify analytically an arbitrary element in the sequence. By recursively applying equation 6.1, it is easy to show that the element X_{P+N} is given by

$$\begin{aligned} X_{P+N} &= (A * X_P + B) \mod M \qquad (6.2) \\ A &= a^N \\ B &= \sum_{k=0}^{N-1} a^k = (a^N - 1)/(a - 1) \end{aligned}$$

Thus moving through the sequence with stride N, rather than 1, is simply accomplished by specifying a different multiplier and additive constant for the generator.

```
// Parallel MLCG program

class Prand {
  final static int MULT = 1103515245;
  final static int ADD  = 12345;
  final static int MASK = 0x7fffffff;
  final static double twoto31 = 2147483648.0;
  static long AAA,BBB,randx;

  // Initialize random number generator.
  static void randinit(int seed) {
    AAA = 1;
    BBB = 0;
    for (int proc=0;proc<Total_procs;++proc) {
      AAA = (MULT * AAA) & MASK;
      BBB = (MULT * BBB + ADD) & MASK;
      if (proc == My_procnum) randx = (AAA*seed + BBB) & MASK;
    }
  }

  // Return a random double in [0, 1.0).
  static double rand()  {
    double retvalue = randx / twoto31;
    randx = (AAA*randx+BBB)&MASK;
    return(retvalue);
  }
}
```

Figure 6.1: Parallel Multiplicative Linear Congruential Generators (MLCG).

In mpiJava Potts model and Ising model simulations, we use parallel MLCG using leapfrog algorithm as shown in Figure 6.1. The leapfrog method is consequently very easy to implement on parallel machine. The idea is to take a standard linear congruential algorithm, and have every processor compute the *Pth* iterate of that algorithm (where P is the number of processors.) [48]. If the seeds are set up properly (staggered) the processors leapfrog over one another, and it is just as good as having used the basic algorithm on a sequential machine.

6.2 The Monte Carlo Simulations of Spin Model

Parallel computers have been very successfully applied to the Monte Carlo simulation of spin models. Here we present overview of the Potts spin models and then outline two different methods, metropolis and cluster algorithms that known for Monte Carlo simulation of many spin model.

6.2.1 The Potts Model

The Potts model [85, 97] encompasses a number of problems in statistical physics and lattice theory. It generalizes of the Ising model so that each spin can have more than two components, and has been a subject of increasingly intense research interest in recent years. It includes the ice-rule vertex and bond percolation models as special cases.

The Q-state Potts model consists of a lattice of spins σ_i, which can take Q different values, and whose Hamiltonian is

$$H = K \sum_{(i,j)} \delta_{\sigma_i \sigma_j}, \tag{6.3}$$

where the spins take on the values $\sigma_i = 0, 1, 2, ..., Q-1$, the sum is over nearest neighbor pairs on sites on a lattice. For Q=2, this is equivalent to the Ising model. The Potts model is thus a simple extension of the Ising model; however, it has a much richer phase structure, which makes it an important testing ground for new theories and algorithms in the study of critical phenomena.

6.2.2 The Metropolis Algorithm

The Metropolis algorithm [39] was invented by Metropolis et al in 1953 for generating an arbitrary probability distribution. This algorithm has been the most successful

1. Select one spin to flip.
2. Compute the change in energy resulting from this flip.
3. Calculate the probability $P = exp(-\beta * \Delta E)$ for this flip.
4. Generate a random number r between zero and one.
5. If $\Delta E \leq 0$ or $r < P$, flip the spin to one of values $0, 1, ..., Q-1$.
6. Repeat step 1 to 5 over all sites.
7. Visualize the updated lattice configuration.
8. Analyze the resulting configuration.

Figure 6.2: The Main Procedure of Potts Model Simulation using Metropolis Algorithm: One starts with an initial configuration of spins and repeats these procedures.

and influential of all the areas of Monte Carlo Method.

In the Metropolis algorithm, new states or configurations of the system are found by systematically moving through all the lattice sites. A new configuration ϕ' is generated by updating a single variable in the old configuration ϕ and calculating the change in energy

$$\Delta E = E(\phi') - E(\phi). \tag{6.4}$$

If $\Delta E \leq 0$, the change is accepted; if $\Delta E > 0$ the change is accepted with probability $\exp(-\beta * \Delta E)$, where β is inverse temperature. In practice, on a computer, this is done by generating a pseudo-random number r in the interval [0,1) with uniform probability distribution and accepting the change if $r < \exp(-\Delta E)$. These procedures of Metropolis algorithm are shown in Figure 6.2. If the variable is updated by a large amount then the change in energy will be large and the probability of the update being accepted will be small. Conversely, if the variable is changed by

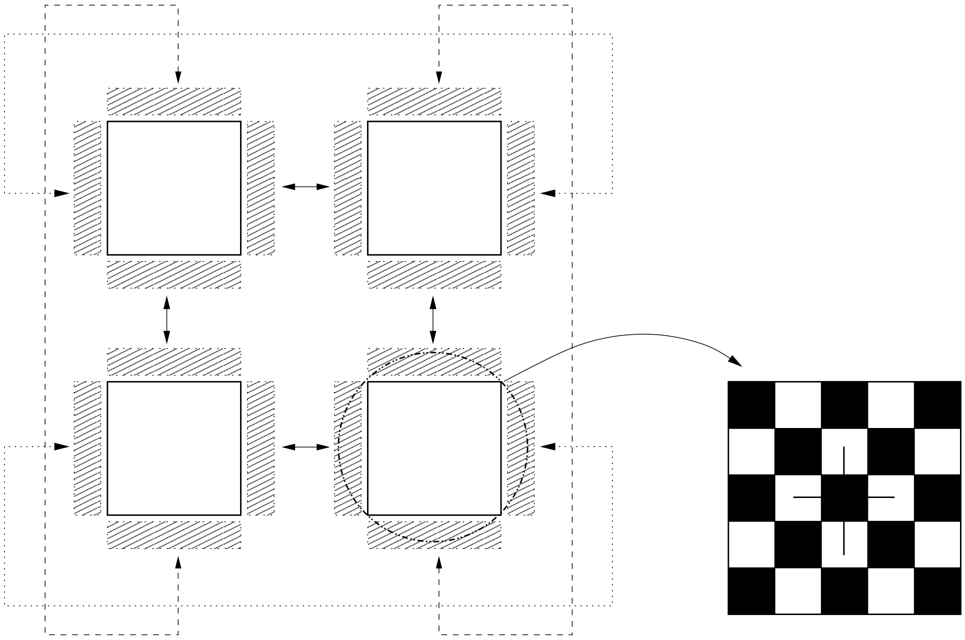

Figure 6.3: Checkerboard Partition and Blocked Communication.

a small amount then the changes will usually be accepted but their small size will make exploration of phase space rather slow.

The Metropolis algorithm for a spin model is well suited to parallelism, since it is regular – so can use standard data decomposition of the lattice onto the processors and get good load balance – and local – the update of a site depends only on its nearest neighbors, so only local communication is required. On SIMD computers, we can not update a site and its neighbors at the same time, since they are dependent each other, and we would violate detailed balance. The detailed balance is a sufficient condition for equilibrium probability distribution and guarantee a valid Monte Carlo procedure. Therefore each site must updated separately. However we can partition the lattice into a checkerboard of red and black sites, and update all the black sited in parallel(at once), since they are all independent, and then followed by all red sites in parallel. This method is well known as *red/black* or *checkerboard* update.

On MIMD computers, we do a checkerboard partitioning of the lattice to do the data communication as large blocks of spins rather than single spins, by passing all the edge data to neighboring processors after each red or black update. This checkerboard partitioning and blocked communication is illustrated in Figure 6.3. This blocked communication greatly reduces the latency time for communication, thereby improving efficiency. It is also possible to overlap communications with computation, by computing edge values first, and sending them while the interior points are being computed.

6.2.3 The Swendsen-Wang Cluster Algorithm

The previous Metropolis algorithm perform poorly because the updates are *local*, that is, one spin at a time is updated. Cluster algorithms use a clever way of finding large clusters of sites which can be updated at once.

Cluster algorithms were proposed by Swendsen and Wang [88] in 1987, for Potts spin models. They have since been generalized to other models, and other algorithms. In the Swendsen-Wang algorithm, clusters of spins are created by introducing bonds between neighboring spins with probability

$$P(\sigma_i, \sigma_j) = \delta_{\sigma_i \sigma_j}(1 - \exp^{-\beta}) \tag{6.5}$$

if the two spins are the same, and zero if they are not. All such clusters are generated and then updated by choosing a random new spin value for each cluster and assigning it to all the spins in that cluster.

A slightly different cluster algorithm has been proposed by Wolff [**?**]. In the Swendsen-Wang algorithm, we generated many clusters and then flip these clusters. In Wolff algorithm, a spin is chosen at random and a single cluster constructed around it, using same bond probabilities as for the Swendsen-Wang algorithm. All the spins in this cluster are then flipped, i.e, changed to a random new spin different

1. Go through each four nearest neighbor connection of the lattice, set up a bond between the two neighboring sites i and j with probability $p = 1 - \exp^{-\beta}$ only if the spins are the same, otherwise zero.

2. Identify clusters as a set of sites connected by bonds. Two sites are said to be in the same cluster if there is a connected path of bonds joining them. Every site has to belong to one of the clusters.

3. After the clusters are created, each cluster is assigned a new Ising spin chosen with equal probability between +1 and -1.

4. Repeat step 1 to 3 for the next iteration.

Figure 6.4: The Main Procedure of Sequential Swendsen-Wang Algorithm.

from the old one. These parallel Wolff cluster algorithms were developed in several different languages including C code for the CM-5 using CMMD [89] which was presented in [12].

Cluster algorithms have in common the problem of identifying and labeling the connected clusters of spins. This is very similar to an important problem in image processing, that of identifying and labeling the connected components in a binary or multi-colored image composed of an array of pixels. The only real difference is that in the spin model case, neighboring sites of the same spin have a certain *probability* of being in the same cluster, while for neighboring pixels of the same color that probability is one. Unfortunately this is a large enough difference so that some algorithms which work in image analysis will not work, or require substantial changes, for spin models. In our Ising model, we have used a commonly used cluster identification algorithm invented by Hoshen and Kopelman.

6.2.3.1 Hoshen and Kopelman

Hoshen and Kopelman [61] developed a sequential cluster identification algorithm which gives each cluster a unique label and counts the number of sites it contains. Each site i belonging to cluster α is assigned a cluster label m_t^α. The cluster α may initially be assigned several different cluster labels. These are given as a set of natural numbers $\{m_1^\alpha, m_2^\alpha, \ldots, m_s^\alpha, \ldots, m_t^\alpha, \ldots\}$. In this set only one number is regarded as the *proper* cluster label which we designate as m_s^α. This is the smallest number in the set.

A connection between the label m_t^α at any site, and the proper cluster cluster label m_s^α, is provided by an array N as $\{N(m_1^\alpha), N(m_2^\alpha), \ldots, N(m_s^\alpha), \ldots, N(m_t^\alpha), \ldots\}$. This is constructed so that $N(m_s^\alpha)$ is the only positive integer member of the set and denotes the number of sites in the cluster, while the remaining $N(m_t^\alpha)$ provide the links between the $N(m_t^\alpha)$ and the proper cluster label $N(m_s^\alpha)$. To be specific, if a site with label m_t^α is bonded to a site which has the proper cluster label m_s^α then

$$N(m_t^\alpha) = -m_s^\alpha \tag{6.6}$$

However, if a site with label m_p^α is not directly bonded to a site with the proper cluster label, but is connected to a site with label m_t^α then

$$N(m_p^\alpha) = -m_t^\alpha \tag{6.7}$$

Therefor to get the proper cluster label for this site we have to go through m_t^α using 6.7 then 6.6. Similarly for higher-order indirections, we just need to iterate $-m \leftarrow N(m)$ until we reach a positive value of $N(m)$, which means that m is the proper cluster label and $N(m)$ the current number of sites in the cluster. Fortunately in most cases this hierarchy consists to one or two levels(one or two equations) only [61].

In practice the algorithm works as follows. We sweep through the sites of the lattice looking at neighbors in the negative directions(there are d of them for a d-dimensional lattice). If site i has no connected neighbors(or none of its neighbors have been labeled) then it is assigned a new label $S_i = m_t^\alpha$, and $N(m_t^\alpha) = 1$. If there is only one connected neighbor, at site n with label $S_n = m_r^\alpha$, say, then i gets the same label as $n : S_i = m_r^\alpha$, and $N(m_t^\alpha) = N(m_t^\alpha)+1$. The good feature of this cluster labeling technique becomes apparent when site i links two or more previously labeled cluster fragments into a single cluster, that is, when it is bonded to two or more of its neighbors. No site belonging to any of these cluster fragments is relabeled (so that once a site is labeled it retains this label throughout the labeling process)—instead the readjustments occur within the $N(m_t^\alpha)$. The number of readjusted $N(m_t^\alpha)$'s for a site is equal to the number of coalescing cluster fragments at that site. Let us assume that the connected sites belong to clusters $\alpha, \beta, \gamma, \ldots$ which have proper cluster labels $m_s^\alpha, m_s^\beta, m_s^\gamma, \ldots$ with m_s^α being the smallest. These clusters coalesce at i to from a single larger cluster so we set label $S_i = m_s^\alpha$ and readjust

$$
\begin{aligned}
N(m_s^\alpha) &= N(m_s^\alpha) + N(m_s^\beta) + N(m_s^\gamma) + \ldots + 1 \\
N(m_s^\beta) &= -m_s^\alpha \\
N(m_s^\gamma) &= -m_s^\alpha \\
&\ldots
\end{aligned}
\tag{6.8}
$$

It is this continual readjustment which keeps the hierarchy of indirections small. As a final stage in the algorithm we can pass through the lattice a second time setting the label at each site to be the proper cluster label m_s^α. This makes it easier to pick out the clusters for updating. An illustration of this algorithm for the 4x4 lattice used in Figure **??**.

6.2.3.2 Parallel Cluster Algorithm

A parallel cluster algorithm involves distributing the lattice onto an array of processors using the usual domain decomposition. Clearly a sequential algorithm can be used to label the clusters on each processors, but we also need a procedure for converting these labels to their correct *global* values. We need to be able to tell many processors, which may be any distance apart, that some of their clusters are actually the same. Thus we need to be able to agree on which of many different local labels for a given cluster should be assigned to be the global cluster label, and to pass this label to all the processors containing a part of that cluster. We will discuss one of the methods called *self-labeling* for tackling this problem.

Self-labeling We shall refer to this algorithm as *self-labeling*, since each site figures out which cluster it is in by itself from local information. We begin by assigning each site i a unique cluster label S_i. In practice this is simply chosen as the position of that site in the lattice. At each step of the algorithm, in parallel, every site looks in turn at each of its neighbors in the positive directions. If it is bonded to a neighboring site n which has a different cluster label S_n, then both S_i and S_n are set to the minimum of the two. This is continued until nothing changes, by which time all the clusters will have been labeled with the minimum initial label of all the sites in the cluster. Note that to check termination of the algorithm involves each processors sending a termination flag(finished or nor finished) to every other processor after each step, which can become very costly for large processor array.

We can improve this method by using faster sequential algorithm, such as Hoshen and Kopelman mentioned in previous section, to label the clusters in the sub-lattice on each processor, and then just use self-labeling on the sites at the edges of each processor to eventually arrive at the global cluster labels. These are

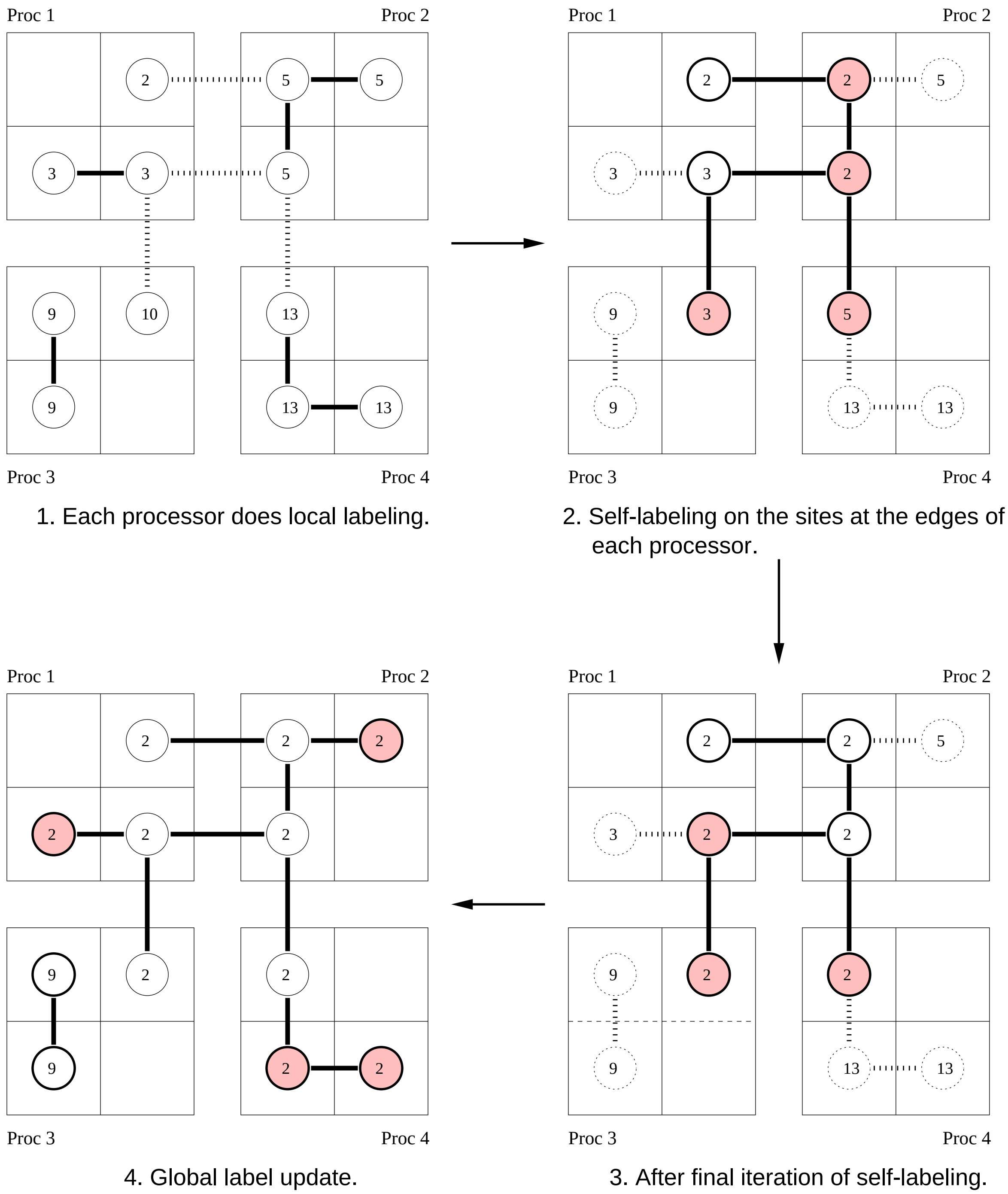

Figure 6.5: MIMD Component Labeling. The bonds are shown as the thick lines.

illustrated in Figure 6.5. The number of steps required to do the self-labeling will depend on the largest cluster, which at the phase transition will generally span the entire lattice. The number of steps self-labeling steps will therefore be of the order of the maximum distance between processors, which for a square array of P processors is just $2\sqrt{P}$. Hence the amount of communication(and calculation) involved in doing the self-labeling, which is proportional to the number of iterations times the perimeter of the sub-lattice, goes like L for an $L \times L$ lattice, whereas the time taken on each processor to do the local cluster labeling goes like the area of the sub-lattice, which is L^2/P. Therefore as long as L is substantially greater than the number of processors we can expect to obtain a reasonable speedup.

6.3 Graphical Simulation

`mpiJava` application that performs a Potts Model Monte Carlo simulation using parallel Metropolis algorithms in two dimensions running on four processors is shown in Figure 6.6. One of the good feature of `mpiJava` is that users have full access to the Java AWT and other Java graphical packages to support GUI and visualize graphical output of application. Visualization of the collected data is a critical element in providing developers or educators with the needed insight into the system under study and education. In this Potts model simulation, the different colors represent the Q different spin states on a lattice. All the options of the simulation are customizable through a GUI. Processor *P0* handles all GUI input and visualization of output data. When users run simulation by clicking start button, input values are broadcasted to other processors and then after each processor generate a new configuration, processor *P0* gathers all updated data to plot the output in order to visualize the results. Measurements of standard quantities such as energy and magnetization are easily done in parallel by just calculate the quantities locally, sum

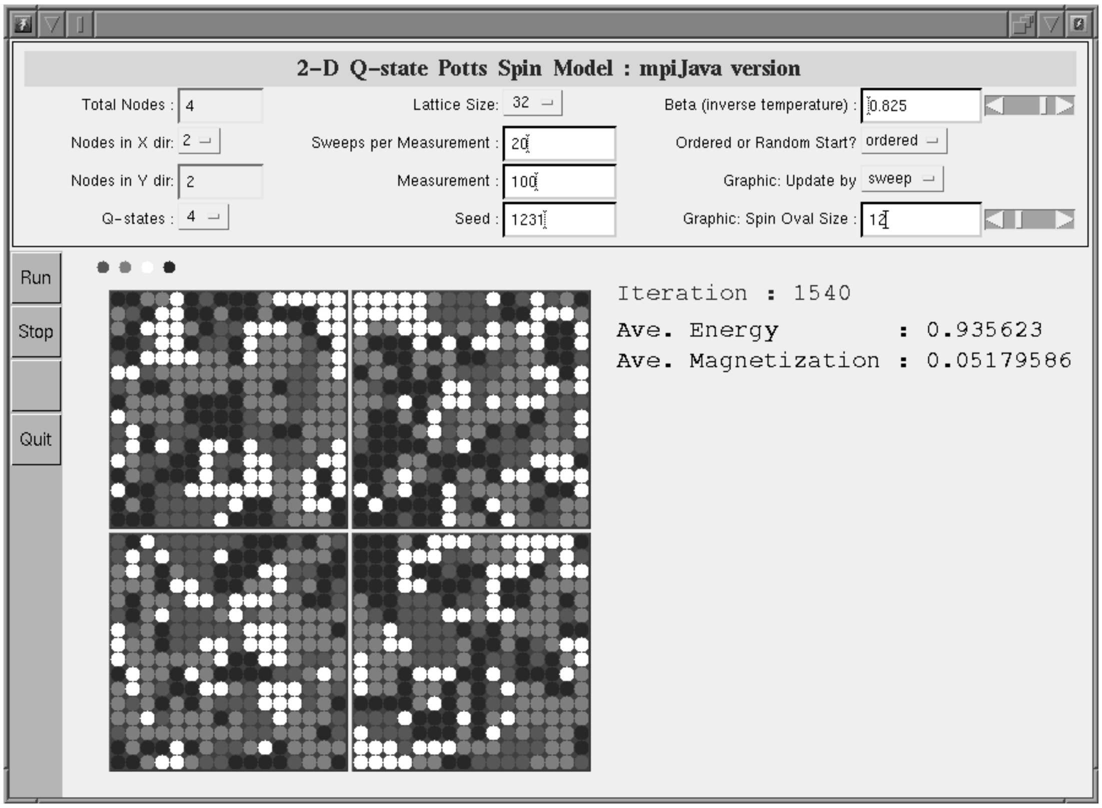

Figure 6.6: The Potts Model Monte Carlo Simulation.

over sites in each processor, and then combine the partial sums on each processor using MPI.Allreduce.

Users can also choose data distribution in row, column or block wise through the number of nodes in X direction options after the number of running processors is given at the starting time by execution command as follow;

```
prunjava 4 Potts
```

The `prunjava` script is a wrapper for the MPICH `mpirun` script. The first argument is the number of processors on which the program will be executed. The number of nodes in Y direction is set automatically by total number of running nodes and the given number of nodes in X direction. A list of available host computers may be given in an MPICH-style 'machines' file in the local directory.

From this simulation, one is able to determine the average magnetization, the average energy, and the specific heat of the system and numerous other quantities which are experimentally observable.

6.4 Ising Model Evaluation

In this section we evaluate the performance of applications written in mpiJava. We have developed both sequential and parallel message passing programs to compare the performance of mpiJava using Monte Carlo simulation of the two dimensional Ising model. The sequential program were written in C, F77 and Java and message passing programs were written in MPI-C, MPI-F77 and mpiJava. This Ising model test can also be used for testing the quality of parallel random number generators [32].

The two different methods, Metropolis and Swendsen-Wang cluster algorithms are used with a standard block domain decomposition. As in Potts model, the red/black updating scheme is used in parallel Metropolis Ising model simulation [48].

The system environment was as follows:

- CPU: Four dual processor (UltraSparc 200 MHz) Solaris workstations with 256 MBytes of DRAM. Each of the two processors/system has 1MB cache.
- OS: Solaris 2.5.1
- Network: 10BaseT Ethernet and SunATM-155/MFiber
- Java VM: Sun's JIT
- Java Compiler: Sun JDK1.2.2
- MPI packages: MPICH1.2.0 and Sun HPC MPI 2.0

For comparison, we have completed experiments for three different versions of out programming environment—sequential codes, parallel codes with MPICH and Sun HPC MPI. For the sequential F77 and C codes, we use Sun WorkShop FORTRAN 77 5.0 and C 5.0 compilers and Sun JDK1.2.2 for sequential Java code. For the parallel mpiJava codes, we use MPICH 1.2.0 and Sun HPC MPI 2.0 on 1, 2, 4 and 8 nodes. The MPICH 1.2.0 was compiled with `-comm=shared` communication option so that to permit the use of multiple communication of both shared memory and TCP/IP, since each of four our Solaris machines has dual processors. Sun HPC MPI uses Sun ATM network which is much faster in communication than p4 device of MPICH. For better performance, all sequential and parallel Fortran, C and Java codes were compiled using `-O` optimization option and all sequential Java and mpiJava codes executed with JIT enabled.

We performed Monte Carlo Ising simulations on simple 2-D lattices with linear size of L=32 to 1024 and periodic boundary conditions. To measure the total execution time, we performed 20 iterations to thermalize the system and at least 2000 to Monte Carlo sweeps. The inverse temperature β was taken to be the critical inverse temperature $\beta_c = \log(1+\sqrt{2})/2 \approx 0.4406868$ for the $2-d$ Ising model. Timings were measured using `MPI_Wtime` for parallel and shell built-in `time` command for sequential codes. We have repeated the benchmarks several times when there was little network activity and on quiet machines.

The Metropolis timing results for sequential and parallel tests for different lattice sizes are shown in Figure 6.7 through 6.12 and Swendsen-Wang in Figure 6.13 through 6.18. The results demonstrate as follows;

- **MPICH vs sunHPC**

  In both Metropolis and Swendsen-Wang(SW), for small lattice size $L \leq 64$ on node $N \geq 2$, the performance of the mpiJava benchmarks is better with Sun HPC MPI than MPICH due to faster communication device. On the other hand, for the larger latice size $L \geq 512$, the performance difference is much smaller between sunHPC and MPICH, since the main overhead is in the computations rather than communications as the size of sub-lattice increases.

- **mpiJava vs serial codes**

  The speedup obtained for a variety of lattice sizes using the mpiJava, as compared with sequential Java, are shown in Figure 6.19 and 6.20. On the largest lattice size plotted(1024×1024), mpiJava with sunHPC provides a speedup up to 2.4(Metropolis) and 4.6(SW) over sequential F77 and, 2.1(Metropolis) and 3.7(SW) over sequential C.

- **mpiJava vs MPI F77 and C**

  It is not surprising that mpiJava has lower performance than MPI F77 or C since the performance of Java floating point arithmetic is slower than the performance of native C or Fortran code [69]. In addition, mpiJava uses wrapper routines to interface between the Java and the C routines that introduce additional overhead. The sequential and parallel Java/F77 and Java/C run time ratios of Metropolis and Swendsen-Wang are shown in Table 6.1 and 6.2 respectively. It shows that the performance difference between mpiJava and MPI F77 or C decreases as the number of processors increases. For Swendsen-Wang benchmarks, the MPI F77 and C codes are only about twice as fast as mpiJava version for the largest lattice size $L = 1024$.

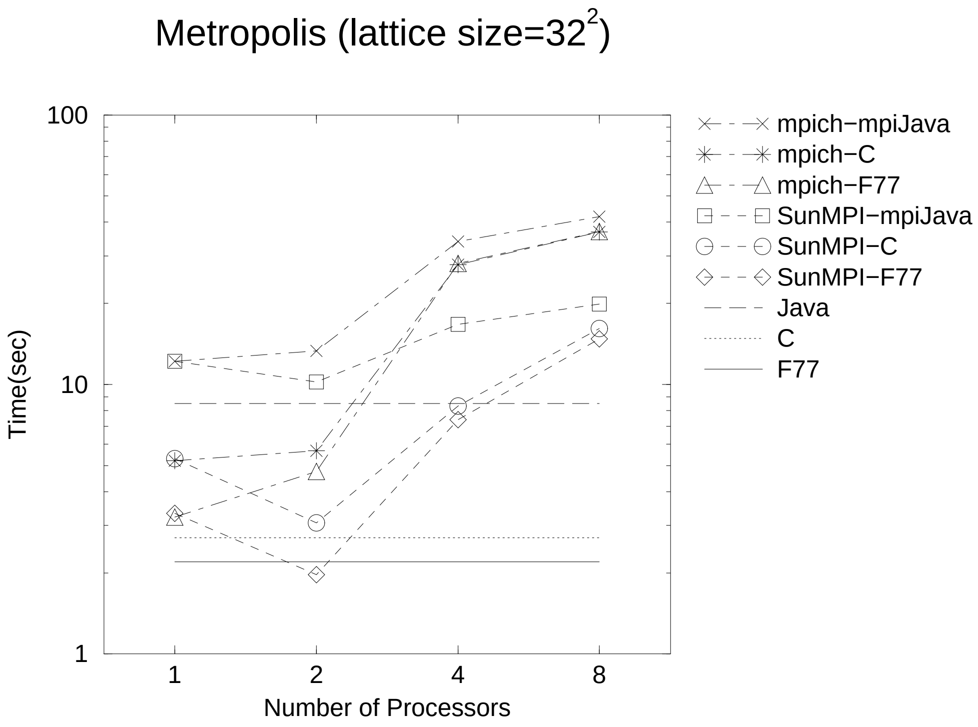

Figure 6.7: Metropolis Performance with Lattice Size 32^2.

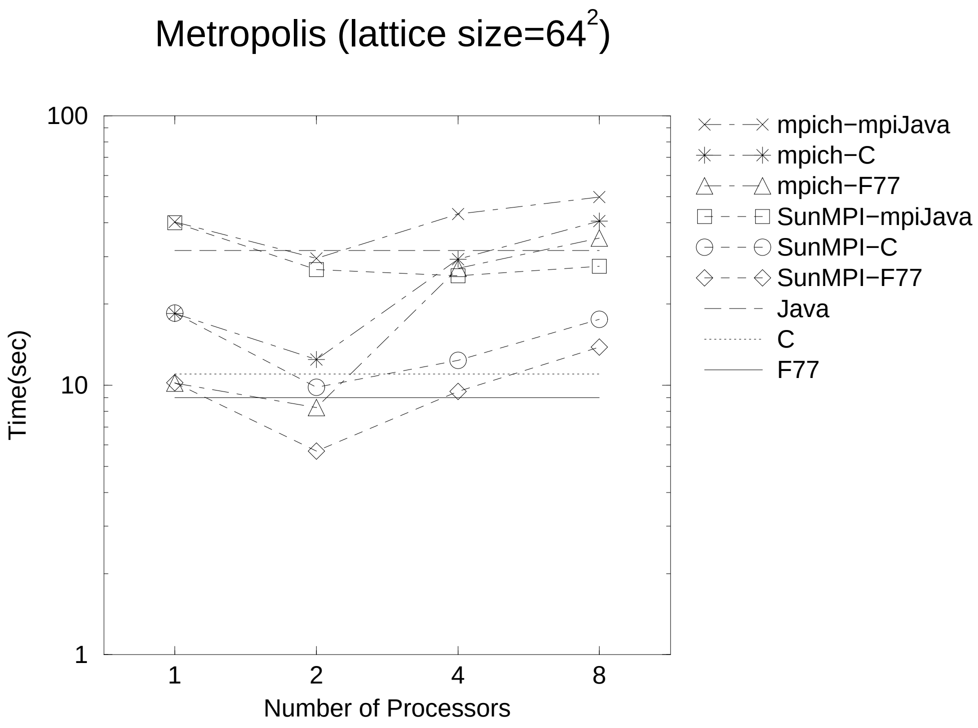

Figure 6.8: Metropolis Performance with Lattice Size 64^2.

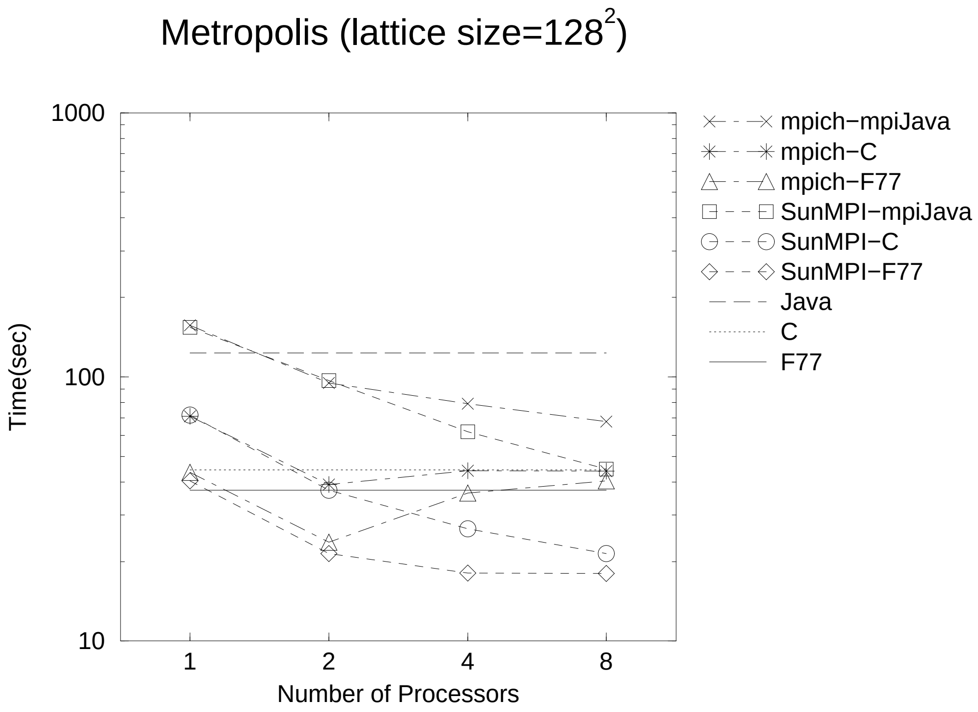

Figure 6.9: Metropolis Performance with Lattice Size 128^2.

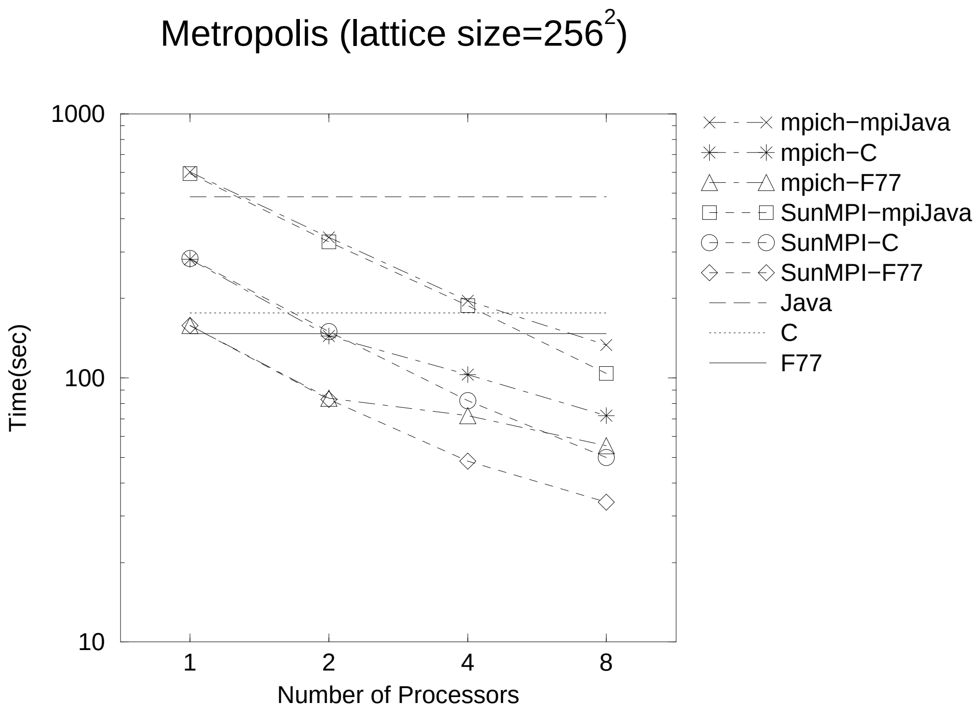

Figure 6.10: Metropolis Performance with Lattice Size 256^2.

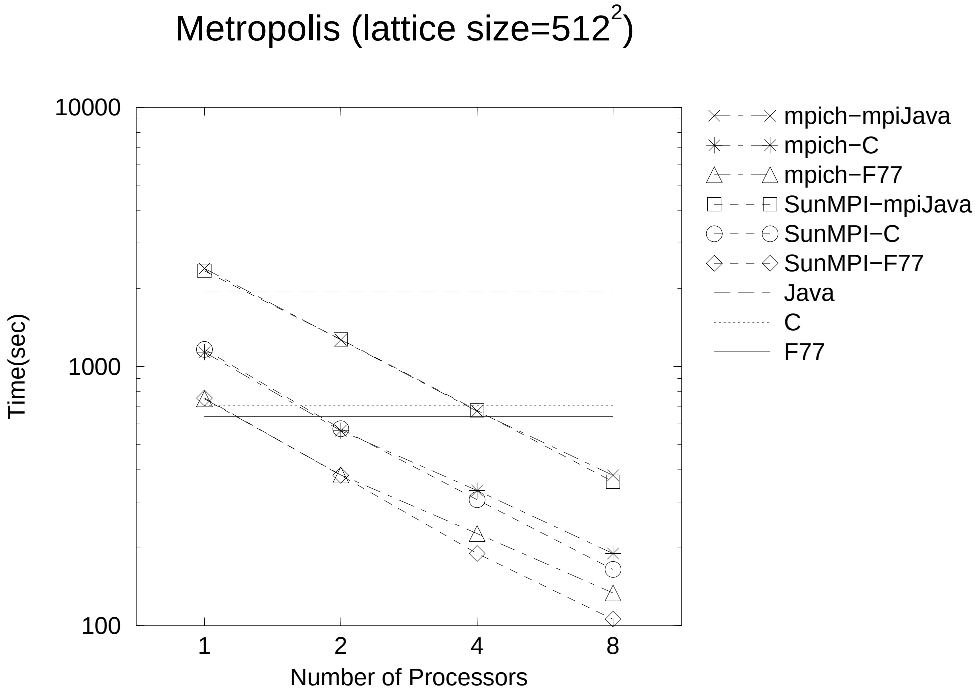

Figure 6.11: Metropolis Performance with Lattice Size 512^2.

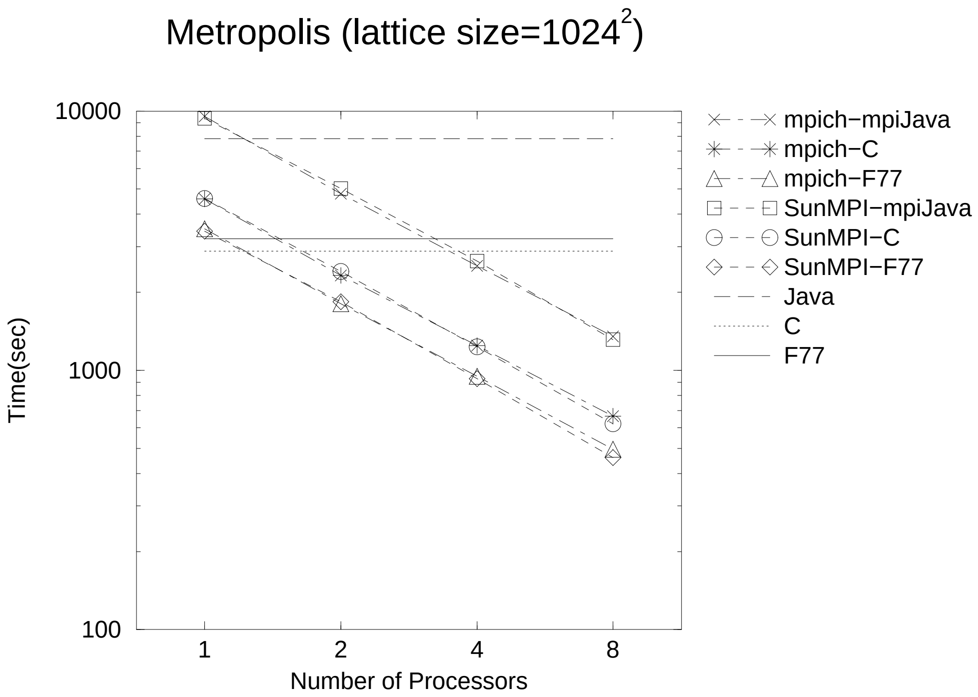

Figure 6.12: Metropolis Performance with Lattice Size 1024^2.

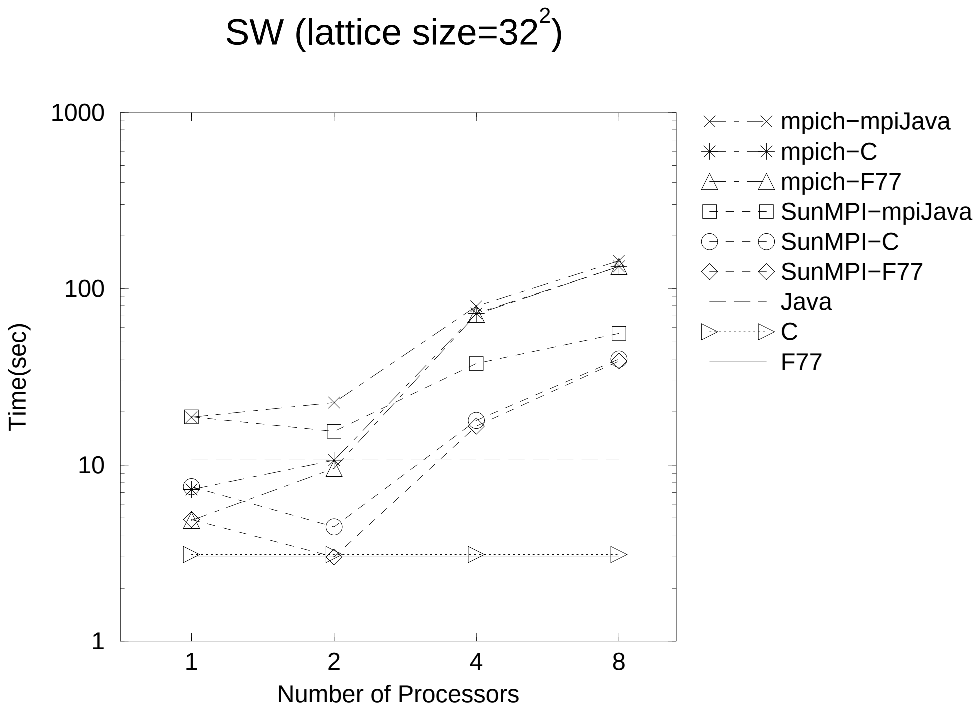

Figure 6.13: Swendsen-Wang Performance with Lattice Size 32^2.

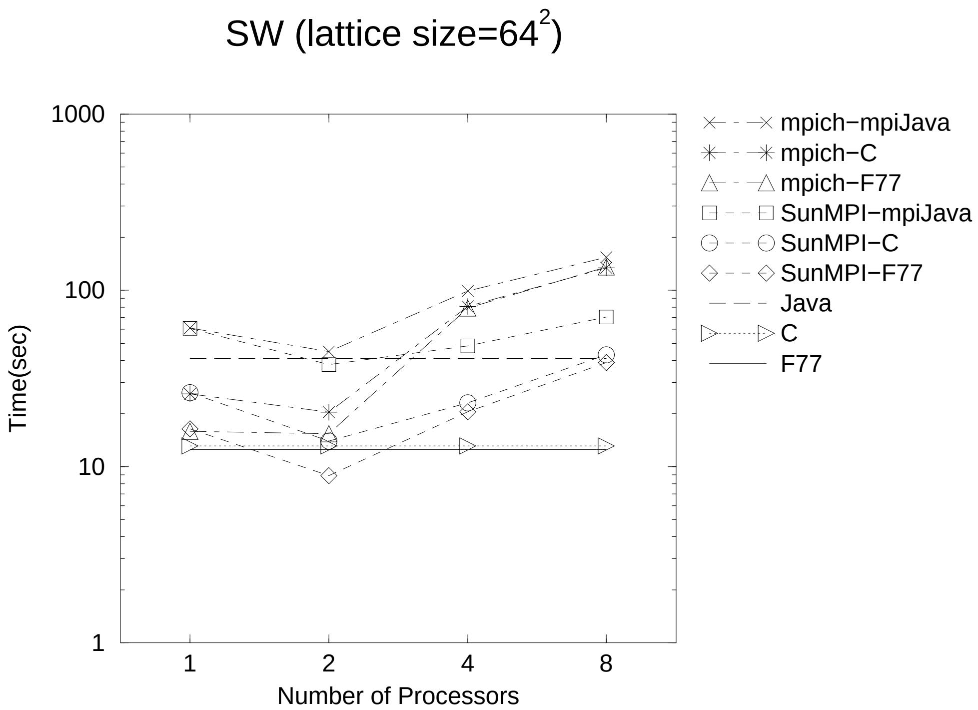

Figure 6.14: Swendsen-Wang Performance with Lattice Size 64^2.

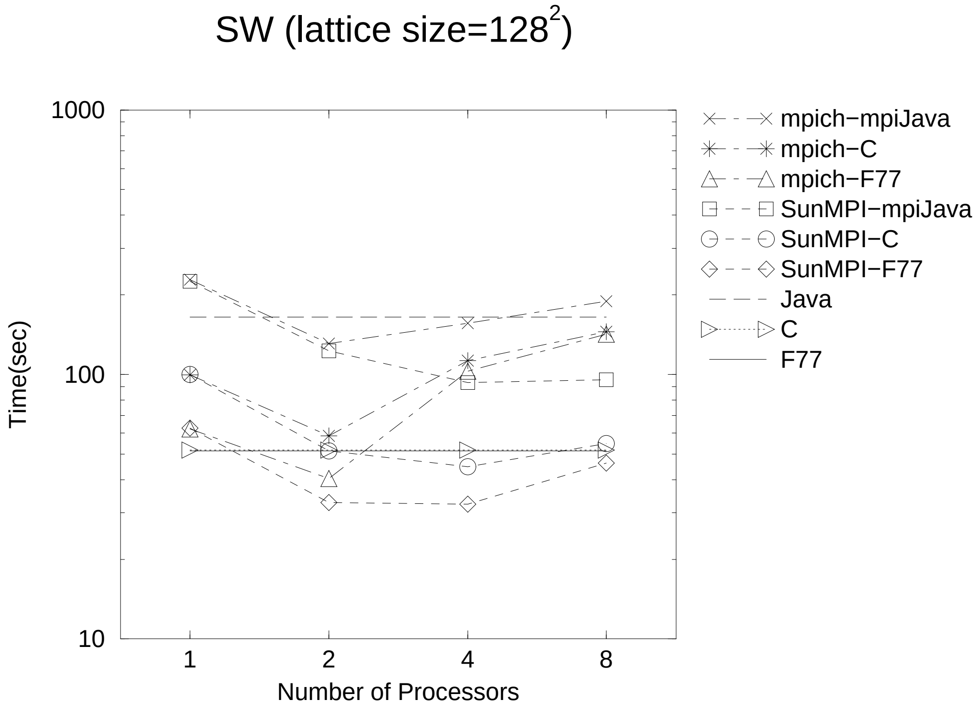

Figure 6.15: Swendsen-Wang Performance with Lattice Size 128^2.

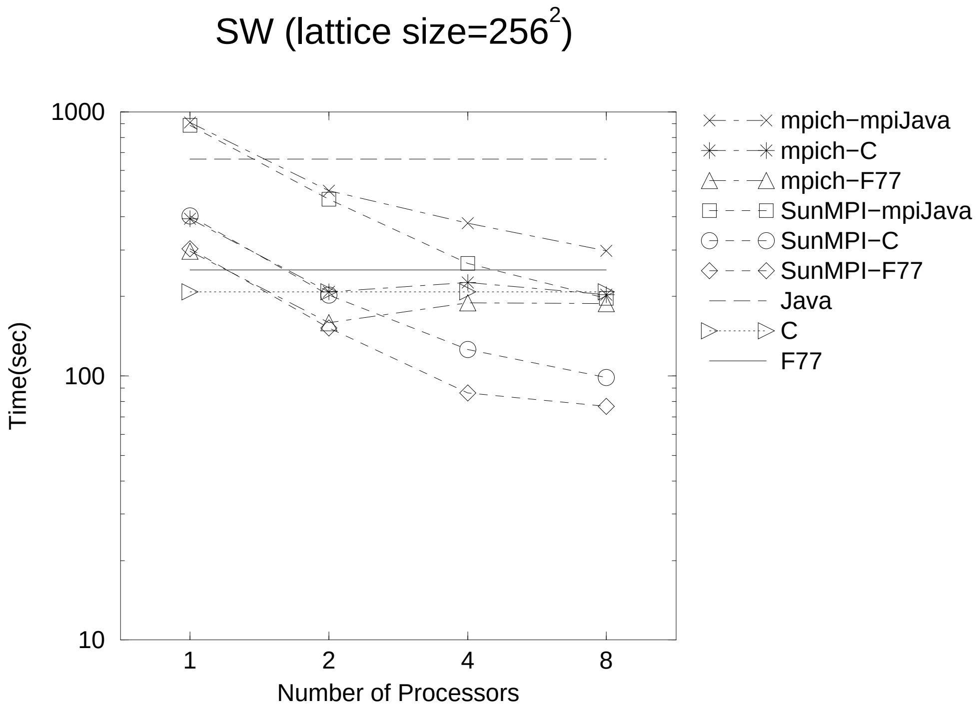

Figure 6.16: Swendsen-Wang Performance with Lattice Size 256^2.

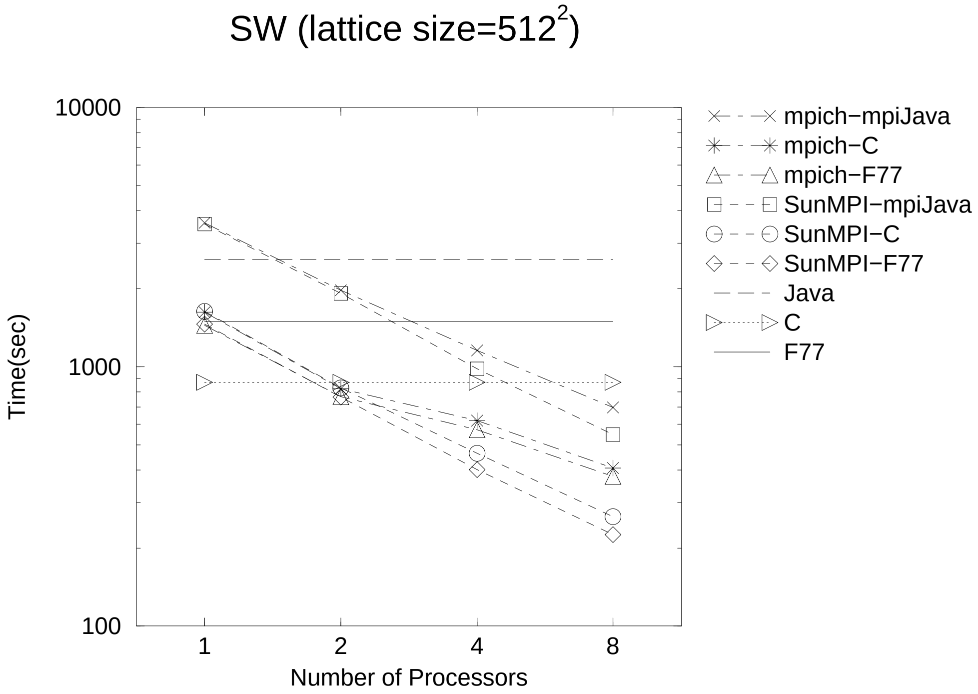

Figure 6.17: Swendsen-Wang Performance with Lattice Size 512^2.

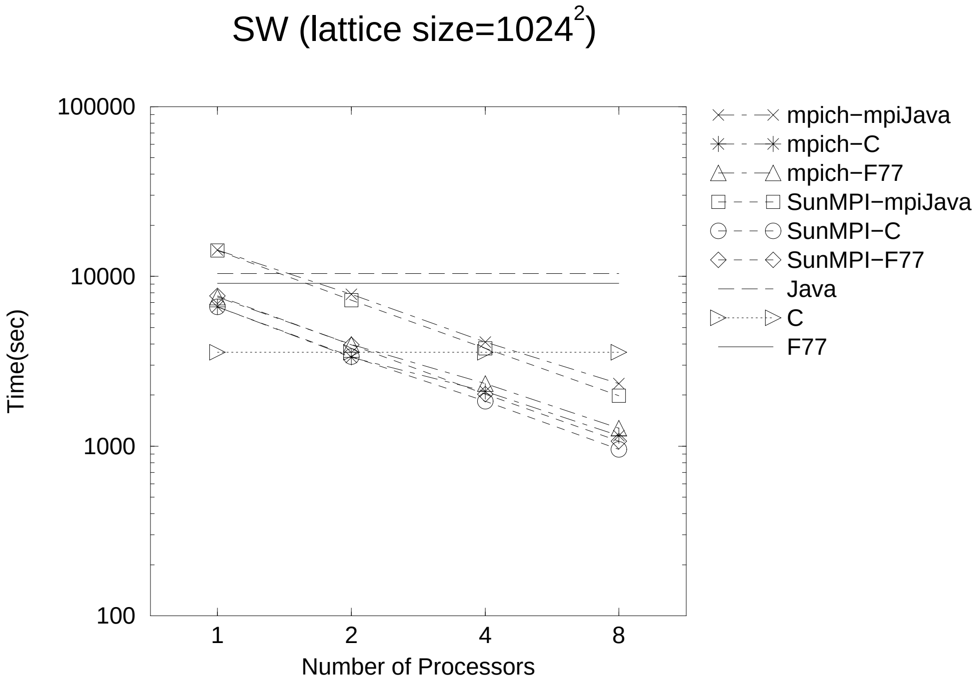

Figure 6.18: Swendsen-Wang Performance with Lattice Size 1024^2.

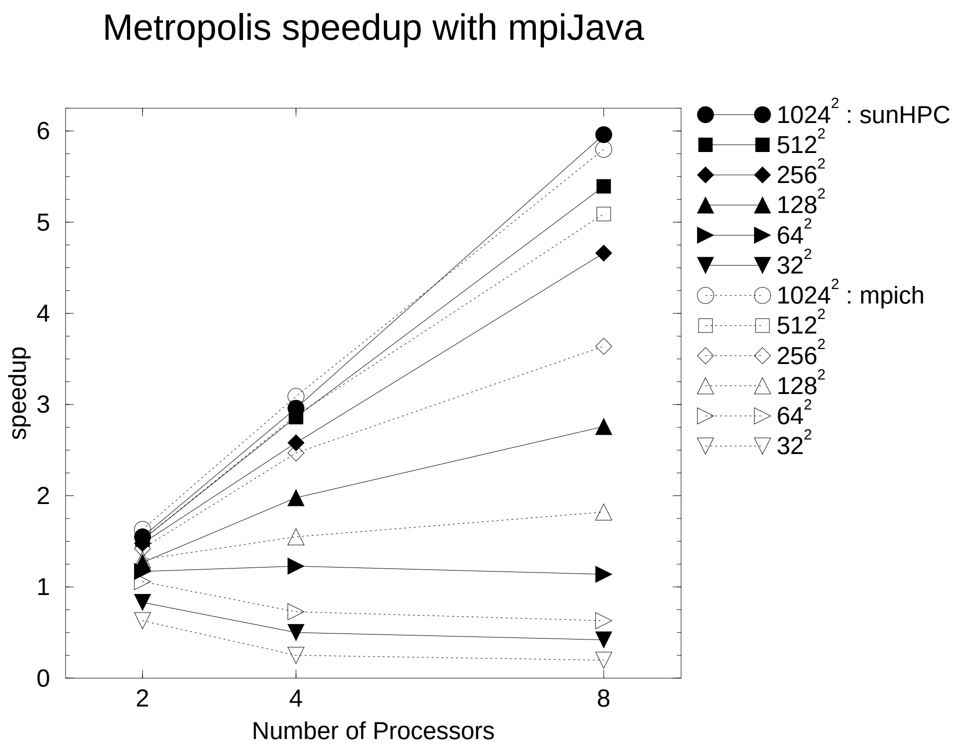

Figure 6.19: Speedup of Metropolis by using mpiJava as compared with serial Java.

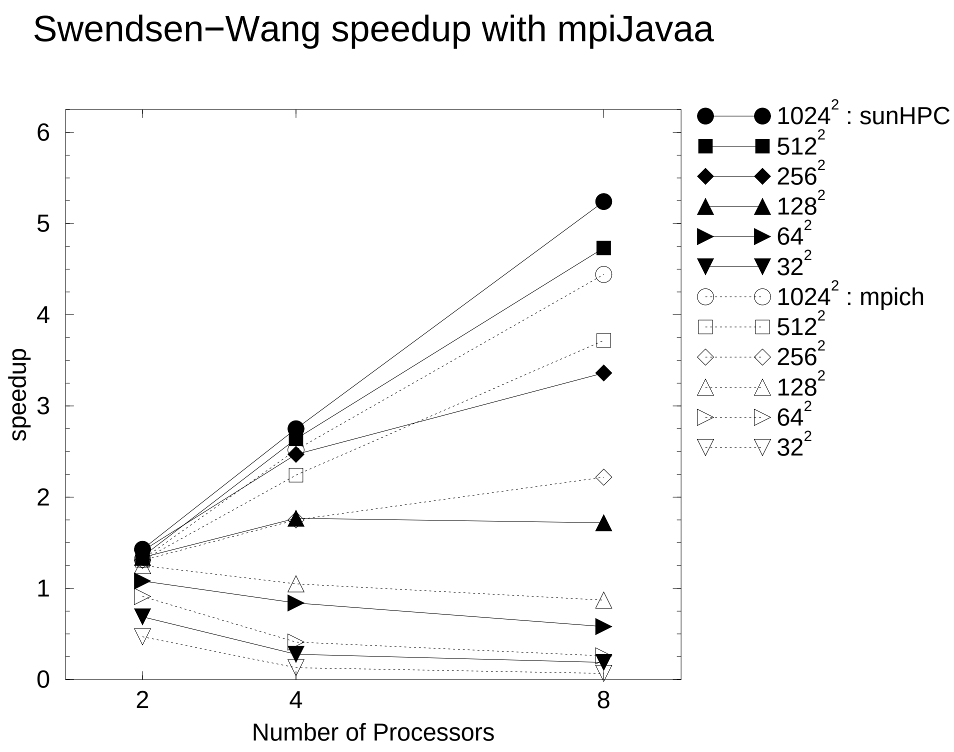

Figure 6.20: Speedup of Swendsen-Wang by using mpiJava as compared with serial Java.

	sunHPC		MPICH		Sequential	
Nodes	Java/F77	Java/C	Java/F77	Java/C	Java/F77	Java/C
$L = 32^2$						
2	5.17	3.33	2.80	2.34	3.86	3.14
4	2.25	2.00	1.20	1.21		
8	1.34	1.23	1.13	1.14		
$L = 64^2$						
2	4.71	2.74	3.57	2.37	3.51	2.87
4	2.69	2.06	1.59	1.47		
8	1.99	1.57	1.42	1.22		
$L = 128^2$						
2	4.52	2.60	4.00	2.42	3.33	2.74
4	3.43	2.33	2.18	1.79		
8	2.48	2.08	1.67	1.54		
$L = 256^2$						
2	3.94	2.18	4.05	2.36	3.30	2.75
4	3.88	2.29	2.72	1.90		
8	3.07	2.08	2.40	1.84		
$L = 512^2$						
2	3.33	2.20	3.31	2.23	3.01	2.73
4	3.56	2.21	2.96	2.02		
8	3.38	2.17	2.83	2.00		
$L = 1024^2$						
2	2.73	2.08	2.66	2.07	2.43	2.71
4	2.84	2.14	2.67	2.03		
8	2.85	2.11	2.71	2.02		

Table 6.1: Metropolis...

	sunHPC		MPICH		Sequential	
Nodes	Java/F77	Java/C	Java/F77	Java/C	Java/F77	Java/C
$L = 32^2$						
2	5.16	3.49	2.37	2.13	3.60	3.48
4	2.27	2.10	1.11	1.10		
8	1.43	1.39	1.08	1.08		
$L = 64^2$						
2	4.27	2.71	2.92	2.21	3.29	3.13
4	2.37	2.11	1.24	1.22		
8	1.80	1.64	1.14	1.15		
$L = 128^2$						
2	3.75	2.39	3.24	2.24	3.21	3.18
4	2.88	2.08	1.51	1.38		
8	2.07	1.74	1.33	1.30		
$L = 256^2$						
2	3.07	2.31	3.16	2.41	2.63	3.18
4	3.10	2.12	2.00	1.67		
8	2.57	1.99	1.59	1.47		
$L = 512^2$						
2	2.52	2.32	2.57	2.39	1.73	2.98
4	2.45	2.12	2.03	1.86		
8	2.44	2.08	1.84	1.71		
$L = 1024^2$						
2	1.83	2.15	1.98	2.35	1.14	2.90
4	1.87	2.04	1.76	1.96		
8	1.85	2.07	1.84	2.03		

Table 6.2: SW.....

Chapter 7

Conclusion

7.1 Conclusion

We have explored the practicality of doing parallel computing in Java, and of providing Java interfaces to High Performance Computing software. For various reasons, the success of this exercise was not a foregone conclusion. Java sits on a virtual machine model significantly different to the hardware-oriented model that C or Fortran exploit directly. Java discourages or prevents direct access to the some of the fundamental resources of the underlying hardware (most extremely, its memory).

Our earliest experiments in this direction involved working entirely within Java, building new software on top of the communication facilities of the standard API. The work in Chapters 4 and 5 involved creating a Java interface to an existing HPC package. Which is the better strategy? In the long term Java may become a major implementation language for large software packages like MPI. It certainly has advantages in respect of portability that could simplify implementations dramatically. In the immediate term recoding these packages does not appear so attractive. Java wrappers to existing software look more sensible. On a cautionary note, our experience with MPI suggests that interfacing Java to non-trivial communication packages may be less easy than it sounds. Nevertheless, we intend to create a Java interface to an existing run-time library for data parallel computation.

So is Java, as it stands, a good language for High Performance Computing?

It still has to be demonstrated that Java can be compiled to code of efficiency comparable with C or Fortran. Many avenues are being followed simultaneously towards a *higher performance* Java.

In many respects Java is undoubtedly a better language than Fortran. It is object-oriented to the core and highly dynamic, and there is every reason to suppose that such features will be as valuable in scientific computing as in any other programming discipline. But to displace established scientific programming languages Java will probably have to acquire some of the facilities taken for granted in those languages.

HPJava

mpiJava

- `mpiJava` provides a fully functional and efficient Java interface to MPI.
- Our performance tests have shown that, in terms of communications speeds, WMPI on NT out performs MPICH on Solaris.
- When used for distributed computing the current implementation of `mpiJava` does not impose a huge overhead on-top of the native MPI interface.
- As interfacing Java to MPI is not always trivial, in earlier implementation we often saw low-level conflicts between the Java runtime and the interrupt mechanisms used in the MPI implementations. These new *native* thread feature in JDK 1.2 has eliminated the interrupt problem that we encountered

with earlier releases of the JDK. `mpiJava` is now stable on UNIX and Linux platforms using MPICH and JDK 1.2 .

- Our performance tests indicate that much of the additional latency that `mpiJava` imposes is due to the relatively poor performance of the JVM rather than the impact of messages traversing additional software layers.
- The syntax of `mpiJava` is easy to understand and use, thus making it relatively simple for programmers with either a Java or Scientific background to take up.
- We believe that `mpiJava` will also provide a popular means for teaching students the fundamentals of parallel programming with MPI.

7.2 Contributions

The major contributions of this research is the proposing the potential of Java as a language for scientific parallel programming, and then providing Java interfaces to High Performance Computing software.

- We have designed and proposed a high-level structured SPMD programming style, *HPJava*. *HPJava* is a programming language extended from Java to support parallel programming, especially (but not exclusively) data parallel programming on message passing and distributed memory systems, from multi-processor systems to workstation clusters. We propose introducing some of the characteristic ideas of HPF—specifically its distributed array model and array intrinsic functions and libraries—into a basically SPMD programming model.
- We have developed an object-oriented Java interface to MPI, *mpiJava*. We have discussed the design of the *mpiJava* API and the issues associated with

its development. In addition we have presented the steps necessary to port *mpiJava* onto a range of operating systems, including Windows NT, Linux and Solaris. The *mpiJava* package have been released 2nd version and currently more than 400 people download from all around the world. The releases of mpiJava include complete source, makefiles, configuration scripts, compiled libraries for WMPI, release test codes (based on the IBM MPI test suite), example applications, javadoc documentation, and installation and usage notes. The software is publically available from:

`http://www.npac.syr.edu/projects/pcrc/mpiJava`

- We have developed the the Java object serialization model for marshalling general communication data in MPI-like APIs. This approach is compared with a Java transcription of the standard MPI derived datatype mechanism. We describe an implementation of the *mpiJava* interface to MPI that incorporates automatic object serialization. Benchmark results confirm that current JDK implementations of serialization are not fast enough for high performance messaging applications. Means of solving this problem are discussed, and benchmarks for greatly improved schemes are presented.

- Finally, we have developed *mpiJava* version of Monte Carlo spin models using a parallel Java random number genrator. We also have evaluated these with comparable native parallel MPI-C and MPI-Fortran and sequencial C, F77 and Java codes for performance comparison. In addition we have presented graphical parallel Potts model simulation with GUI using Java AWT.

7.3 Future Work

HPJava...

mpiJava We plan to continue to improve `mpiJava` and add-in the functionality that has been proposed in MPI-2[44]. We intend to 'port' `mpiJava` to a multitude of new MPI environments, including LAM[21] and Globus[46]. We are also planning a *pure*-Java MPI environment which does not rely on native MPI services.

MPJ...

Bibliography

[1] URL: http://www.isye.gatech.edu/chmsr/JavaPVM/.

[2] URL: http://ccc.cs.byu.edu/OnlineDocs/docs/mpij/MPIJ.html.

[3] IBM Java Linux. URL: http://www.alphaworks.ibm.com/tech/linuxjvm,.

[4] Java Grande Forum. URL: http://www.javagrande.org.

[5] Java Linux. URL: http://www.blackdown.org/java-linux.html.

[6] Linux mandrake. URL: http://www.linux-mandrake.com.

[7] PingPong Benchmarks. URL: http://www.dcs.port.ac.uk/˜ mab/TOPIC/.

[8] Redhat linux. URL: http://www.redhat.com/.

[9] Wmpi. URL: http://dsg.dei.uc.pt/w32mpi/.

[10] S.L. Anderson. Random Number Generators on Vector Supercomputers and Other Advanced Architectures. *SIAM Rev*, 32:221, 1990.

[11] Susan Atlas, Subhankar Banerjee, Julian C. Cummings, Paul J. Hinker, M. Srikant, John V. W. Reynders, and Mary Dell Tholburn. POOMA: A high performance distributed simulation environment for scientific applications. In *Supercomputing '95*, 1995.

[12] Seungjo Bae, Sung-Hoon Ko, and Paul D. Coddington. Parallel Wolff Cluster Algorithms. *International Journal of Modern Physics C*, 6(2):197–210, 1995.

[13] Mark Baker and Bryan Carpenter. Initial thoughts on structure of an MPJ reference implementation. In *JGMPWG meeting*. NPAC, Syracuse University, October 1999.

[14] Mark A. Baker and Geoffrey C. Fox. MPI on NT: A Preliminary Evaluation of the Available Environments. Parallel and Distributed Computing, Springer Verlag, Heifelberg, Germany, April 1998. 12th IPPS and 9th SPDP, LNCS, Jose Rolim (Ed.).

[15] Shahzad A. Bhatti, Phillip M. Dickens, and George K. Thiruvathukal. Java on Networks of Workstations(JavaNOW): A Parallel Computing Framework Inspired by Linda, Actors, and the Message Passing Interface(MPI). URL: http://www.jhpc.org/projects.htm, http://www.csam.iit.edu/~pmd/pubs.html.

[16] Aart J. C. Bik and Dennis B. Gannon. Automatically exploiting implicit parallelism in Java. *Concurrency: Practice and Experience*, 9(6):579–619, June 1997.

[17] Aart J. C. Bik, Juan E. Villacis, and Dennis B. Gannon. javar: A prototype Java restructuring compiler. *Concurrency: Practice and Experience*, 9(11):1181–1191, November 1997. URL: http://www.extreme.indiana.edu/~ajcbik/JAVAR/.

[18] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK User's Guide*. SIAM, 1997. URL: http://www.netlib.org/scalapack/scalapack_home.html.

[19] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, 1987.

[20] Fabian Breg, Shridhar Diwan, Juan Villacis, Jayashree Balasubramanian, Esra Akman, and Dennis Gannon. Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++. *Concurrency: Practice and Experience*, 10(11-13):941–956, sep–nov 1998. URL: http://www.extreme.indiana.edu/hpjava/nexusrmi/.

[21] G. Burns, R. Daoud, and J. Vaigl. LAM: An open cluster environment for MPI. In *Supercomputing Symposium*, Toronto, Canada, June 1994. URL: http://www.mpi.nd.edu/lam/.

[22] R. Butler and E. Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20:547–564, April 1994.

[23] Ralph Butler and Ewing Lusk. Monitors, messages and cluster: The p4 parallel programming system. *Parallel Computing*, 20:547–564, April 1994.

[24] Bryan Carpenter, Vladimir Getov, Glenn Judd, Tony Skjellum, and Geoffrey Fox. MPI for Java: Position document and graft API specification. Technical Report JGF-TR-3, Java Grande Forum, November 1998. URL: http://www.javagrande.org/.

[25] Bryan Carpenter, Guansong Zhang, and Yuhong Wen. NPAC PCRC Runtime Kernel Definition. Technical Report CRPC-TR97726, Center for Research Parallel Computation, NPAC, Syracuse University, 1997. URL: http://www.npac.syr.edu/projects/pcrc/kernel.html.

[26] D. B. Carpenter. Adlib: A distributed array library to support HPF translation. In *Presented at the 5th International Workshop on Compilers for Parallel Computers*, 1995. URL: http://www.npac.syr.edu/users/dbc/Adlib.

[27] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object-oriented programming notation. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, page 24. MIT Press, 1993.

[28] Chi-Chao Chang and Thorsten von Eiken. Interfacing Java to the Virtual Interface Architecture. In *ACM 1999 Java Grande Conference*. ACM Press, 1999.

[29] A. Choudhary, G. Fox, S. Ranka, S. Hiranandani, K. Kennedy, C. Koelbel, and J. Salts. Software support for irregular and loosely synchronous problems. *Computing Systems in Engineering*, 3:43–52, 1992.

[30] Bernd O. Christiansen, Peter Cappello, Mihai F. Ionescu, Michael O. Neary, Klaus E. Schauser, and Daniel Wu. Javelin: Internet-based parallel computing using Java. *Concurrency: Practice and Experience*, 9(11):1139–1160, November 1997. URL: http://www.cs.ucsb.edu/research/javelin/.

[31] Paul D. Coddington. Random Number Generators for Parallel Computers. *The NHSE Review, 1996 Volume, Second Issuse*, 1996. URL: http://nhse.cs.rice.edu/NHSEreview/, http://www.crpc.rice.edu/NHSEreview/RNG/.

[32] Paul D. Coddington and Sung-Hoon Ko. Techniques for Empirical Testing of Parallel Random Number Generators. In *Proc. of the 12th ACM Int. Conf. on Supercomputing*, Melbourne Australia, July 1998.

[33] Parallel Compiler Runtime Consortium. Common runtime support for high-performance parallel language. In *Supercomputing '93*. IEEE Computer Society Press, 1993.

[34] Parallel Compiler Runtime Consortium. HPCC and Java—a report by the Parallel Compiler Runtime Consortium. Technical report, 1996. URL: http://www.npac.syr.edu/users/gcf/hpjava3.html.

[35] R. Das, M. Uysal, J.H. Salz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994. URL: http://www.cs.umd.edu/projects/hpsl/Chaos.htm.

[36] Kivanc Dincer. jmpi and Performance Instrumentation Analysis and Visualization Tool for jmpi. In *First UK workshop on Java for High Performance Network Computing, Europar '98*, September 1998. URL: http://www.baskent.edu.tr/~kdincer/.

[37] Kivanc Dincer. A Ubiquitous Message Passing Interface Implementation in Java: jmpi. In *Proceedings of IPPS/SPDP 1999(13th International Parallel Processing Symposium and Distributed Processing)*, San Juan, Puerto Rico, April 1999.

[38] J.J. Dongarra, R. Pozo, and D.W. Walker. An object oriented design for high performance linear algebra on distributed memory architectures. In *Object Oriented Numerics Conference*, 1993.

[39] N. Metropolis *et al. J. Chem. Phys*, 21:1087, 1953.

[40] Adam J. Ferrari. JPVM: Network Parallel Computing in Java. In *ACM*

1998 Workshop on Java for High-Performance Network Computing, Palo Alto, February 1998. URL: http://www.cs.virginia.edu/~ajf2j/jpvm.html.

[41] Stephen J. Fink and Scott B. Baden. The KeLP User's Guide. University of California, San Diego, La Jolla, CA, March 1996. URL: http://www-cse.ucsd.edu/groups/hpcl/scg/kelp.html.

[42] High Performanc Fortran Forum. *High Performance Fortran language Specification.* Scientific Programming, special issue 2. 1993.

[43] Java Grande Forum. Java Grande Forum report: Making Java work for high-end computing. Technical Report JGF-TR-1, Java Grande Forum, November 1998. URL: http://www.javagrande.org/.

[44] MPI Forum. *MPI-2: Extensions to the Message-Passing Interface.* URL: http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[45] I. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 26(1):24, 1995.

[46] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997. URL: http://www.globus.org.

[47] Ian Foster, George K. Thiruvathukal, and Steven Tuecke. Technologies for Ubiquitous Supercomputing: A Java Interface to the Nexus Communication System. *Concurrency: Practice and Experience*, 9(6):465–475, June 1997. URL: http://www.globus.org/nexus/nexusjava.html.

[48] G. Fox, M. Johnson, G. Lyzenga, S. Otto, and J. Salmon amd D. Walker. *Solving Problems on Concurrent Processors*, volume I. Prentice Hall, Englewood Cliffs, N.J., 1988.

[49] Geoffrey C. Fox. Java for Computational Science and Engineering–Simulation and Modelling. *Concurrency: Practice and Experience*, 9(6):413–414, June 1997.

[50] Geoffrey C. Fox and Wojtek Furmanski. Computing on the Web: new approaches to parallel processing–petaop and exaop performance in the year 2007. Technical report, 1997. URL: http://www.npac.syr.edu/users/gcf/petastuff/petaweb/.

[51] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine–A User's Guide and Tutorial for Networked Parallel Computing.* Scientific and Engineering. MIT Press, 1994.

[52] editor Geoffrey C. Fox. Java for Computational Science and Engineering–Simulation and Modelling II. *Concurrency: Practice and Experience*, 9(11):1001–1002, November 1997.

[53] editor Geoffrey C. Fox. ACM 1998 Workshop on Java for high-performance network computing. *Concurrency: Practice and Experience*, 10(11-13):821–824, September 1998.

[54] Vladmimir Getov, Susan Flynn Hummel, and Sava Mintchev. High-performance parallel programming in Java. *Concurrency: Practice and Experience*, 10(11–13):863–872, September 1998.

[55] Rob Gordon. *essential JNI: Java Native Interface.* Prentice Hall PTR, Upper Saddle River, NJ 07458, 1998.

[56] James Gosling and Henry McGilton. The Java Language Environment: A White Paper. *JavaSoft, Sun Microsystems, Inc*, 1996. URL: http://www.javasoft.com/doc.

[57] Paul A. Gray and Vaidy S. Sunderam. IceT: distributed computing and Java. *Concurrency: Practice and Experience*, 9(11):1161–1167, November 1997. URL: http://www.mathcs.emory.edu/icet/.

[58] A.S. Grimshaw. An introduction to parallel object-oriented programming with Mentat. Technical Report 91 07, University of Virginia, 1991.

[59] William Gropp, Ewing Lusk, and Anthony Skjellum. Using mpi. The MIT Press, MIT, Cambridge, Massachusetts, 1994.

[60] Tomasz Haupt, Erol Akarsu, and Geoffrey Fox. WebFlow: a Framework for Web Based Metacomputing. In *High-Performance Computing and Networking '99*, Amsterdam, April 1999.

[61] J. Hoshen and R. Kopelman. *Phys. Rev.*, B14:3438, 1976.

[62] George Crawford III, Yoginder Dandass, and Anthony Skjellum. The JMPI Commercial Message Passing Environment and Specification: Requirements, Design, Motivations, Strategies, and Target Users. Technical report, MPI Software Technology, Inc, 1998.

[63] Glenn Judd, Mark Clement, and Quinn Snell. DOGMA: Distributed Object Group Management Architecture. In *ACM 1998 Workshop on Java for High-performance Network Computing*, volume 10(11-13), Palo Alto, February 1998. URL: http://ccc.cs.byu.edu/DOGMA/.

[64] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steel, and M.E. Zosel. *The High Performance Fortran Handbook*. MIT press, 1994.

[65] Vijaykumar Krishnaswamy, Dan Walther, Sumeer Bhola, Ethendranath Bommaiah, George Riley, and Brad Topol. Efficient implementation of Java Re-

mote Method Invocatio(RMI). In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and System(COOTS'98)*, 1998.

[66] Sheng Liang. *The Java Native Interface*. Addison Wesley, 1999.

[67] Inmos Ltd, editor. *occam 2 Reference Manual*. Prentice-Hall International, 1988.

[68] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, and Aske Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *ACM SIGPLAN symposium on principles and practice of parallel programming(PPoPP'99)*, pages 173–182, May 1999. URL: http://www.cs.vu.nl/manta/.

[69] J.A. Mathew, P.D. Coddington, and K.A. Hawick. Analysis and Development of Java Grande Benchmarks. In *ACM 1999 Java Grande Conference*, San Francisco, California, June 1999. URL: http://www.cs.ucsb.edu/conferences/java99.

[70] John Merlin, Bryan Carpenter, and Tony Hey. shpf: a subset High Performance Fortran compilation system. *Fortran Journal*, pages 2–6, March 1996.

[71] Message Passing Interface Forum, University of Tenessee, Knoxville, TN. *MPI: A Message-Passing Interface Standard*, June 1995. URL: http://www.mcs.anl.gov/mpi.

[72] S.P. Midkiff, J.E. Moreira, and M. Snir. Optimizing array reference checking in Java programs. *IBM Systems Journal*, 37(3):409, 1998.

[73] Sava Mintchev and Valdimir Getov. Towards Portable Message Passing in Java: Binding MPI. In *Proceedings of EuroPVM-MPI'97*, pages 135–142, Kraków, Poland, November 1997. Springer LNCS 1332.

[74] Christian Nester, Michael Philippsen, and Bernahard Haumacher. A more efficient RMI for Java. In *Proc. ACM 1999 Java Grande Conference*, pages 152–159, San Francisco, June 1999.

[75] J. Nieplocha, R.J. Harrison, and R.J. Littlefield. The Global Array: Non-uniform-memory-access programming model for high-performance computers. *The Journal of Supercomputing*, 10:197–220, 1996. URL: http://www.emsl.pnl.gov:2080/docs/global/.

[76] R.W. Numrich and J.L. Steidel. A simple parallel extension to Fortran 90. *SIAM News*, page 30, 1997.

[77] Martin Odersky and Michael Philippsen. Espresso grinder, 1996. URL: http://ipd.ira.uka.de/~espresso.

[78] Martin Odersky and Philip Walder. Pizza into Java: Translating theory into practice. In *24th ACM Symposium on Principles of Programming Languages*, January 1997. URL: http://ipd.ira.uka.de/~pizza.

[79] Manish Parashar and J.C. Browne. Systems engineering for high performance computing software: The HDDA/DAGH infrastructure for implementation of parallel structured adaptive mesh. In *Structured Adaptive Mesh Refinement Grid Methods*, IMA Volumes in Mathmatics and its Applications. Springer-Verlag. URL: http://www.cs.utexas.edu/users/dagh/.

[80] R. Parsons and D. Quinlan. A++/P++ array classes for architecture independent finite difference calculations. In *Object Oriented Numerics Conference*, 1994.

[81] Michael Philippsen and Bernhard Haumacher. Bandwidth, Latency, and other

Problems of RMI and Serialization. In *Second Java Grande Forum Meeting*, Palo Alto, May 1998. URL: http://www.javagrande.org/meetings.htm.

[82] Michael Philippsen and Bernhard Haumacher. More Efficient Object Serialization. In *Parallel and Distributed Processing*, LNCS 1586, pages 718–732. International Workshop on Java for Parallel and Distributed Computing, San Juan, Puerto Rico, April 1999.

[83] Michael Philippsen and Matthias Zenger. JavaParty-Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997. URL: http://wwwipd.ira.uka.de/JavaParty/.

[84] The Portland Group, Inc., 9150 SW Pioneer Court, Suite H, Wilsonville, Oregon. *PGHPF Users's Guide.*

[85] R.B. Potts. *Camb. Phil. Soc.*, 48:106, 1952.

[86] Rajeev R. Raje, Joseph I. William, and Michael Boyles. An Asynchronous Remote Method Invocation(ARMI) Mechanism for Java. *Concurrency: Practice and Experience*, 9(11):1207–1211, November 1997.

[87] Lawrence Snyder. A ZPL programming guide. Technical report, University of Washington, May 1997. URL: http://www.cs.washington.edu/research/projects/zpl/.

[88] Robert H. Swendsen and Jian-Sheng Wang. Non-universal critical dynamics in monte carlo simulations. *Physical Review Letters*, 58:86, 1987.

[89] Thinking Machines Corporation, Cambridge, MA. *CMMD Reference Manual*, 1993.

[90] George K. Thiruvathukal, Lovely S. Thomas, and Andy T. Korczynski. Reflective remote method invocation. In *Concurrency: Practice and Experience*, volume 10(11-13), pages 911–926, September-November 1998. URL: http://www.jhpc.org/projects.html.

[91] Rob van Nieuwpoort, Jason Maassen, Henri E. Bal, Thilo Kielmann, and Ronald Veldema. Wide-area parallel computing in Java. In *ACM 1999 Java Grande Conference*, pages 8–14, San Francisco, California, June 1999. URL: http://www.cs.vu.nl/manta/.

[92] Kees van Reeuwijk, Arjan J.C van Gemund, and Henk J. Sips. Spar: a programming language for semi-automatic compilation of parallel programs. *Concurrency: Practice and Experience*, 9(11):1193–1205, November 1997. URL: http://www.pds.twi.tudelf.nl/~reeuwijk/projects/javaws_poster.html.

[93] Matt Welsh. Ninja. URL: http://www.cs.berkeley.edu/~mdw/proj/ninja/.

[94] Matt Welsh. NinjaRMI. URL: http://www.cs.berkeley.edu/~mdw/proj/ninja/ninjarmi.html.

[95] Matt Welsh. Using Java to make servers scream. In *Invited talk at ACM 1999 Java Grande Conference*, San Francisco, June 1999.

[96] Gregory V. Wilson and Paul Lu. *Parallel Programming using C++*. MIT, 1996.

[97] F. Y. Wu. The Potts model. *Reviews of Modern Physics*, 54(1):235–268, January 1982.

[98] Narendar Yalamanchilli and William Cohen. Communication Performance of Java based Parallel Virtual Machines. In *ACM 1998 Workshop on Java for high-Performance Network Computing*, Palo Alto, February 1998.

[99] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A High-Performance Java Dialect. In *ACM 1998 Workshop on Java for High-Performance network computing*, February 1998. URL: http://www.cs.berkeley.edu/Research/Projects/titanium/.

[100] Guansong Zhang, Bryan Carpenter, Geoffrey Fox, Xiaoming Li, Xinying Li, and Yuhong Wen. PCRC-based HPF compilation. In *10th International Workshop on Languages and Compilers for Parallel Computing*, 1997.

Vita

NAME OF AUTHOR: SUNG-HOON KO

This dissertation was typeset with LaTeX 2ε[1] by the author.

[1] LaTeX 2ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin.