Java for High Performance SPMD and Message Passing Programming

SUNG-HOON KO

Syracuse University, 2000

Abstract of Dissertation

We present designs for Java interfaces to High Performance Computing softwares, and research implementation issues associated with their development.

We investigate various issues and options for parallel programming in Java—data parallelism and message passing libraries. We discuss motivations for introducing HPJava, an HPspmd programming model. As part of the larger environment of HPJava, we have designed and developed mpiJava, an object-oriented Java interface to MPI. This has been implemented by "native methods" wrappers around some pre-existing MPI implementation. We describe the design of the mpiJava API and the issues associated with its development. We present and then discuss performance measurements made of communications bandwidth and latency to compare mpiJava with C and Fortran bindings of MPI. We discuss adoption of the Java object serialization model in mpiJava 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 evaluate overheads introduced by object serialization in mpiJava, then present optimized methods that reduce serialization overheads. We present graphical mpiJava Potts model simulation and evaluate the performance of Ising model simulation using Metropolis and Swendsen-Wang cluster algorithms. Benchmark results are compared with native parallel and sequential codes.

We have publically released mpiJava on a Web site. This includes complete source, makefiles, configuration scripts, compiled libraries for WMPI, test codes (based on the IBM MPI test suite), example applications, javadoc documentation, and installation and usage notes.

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

Acknowledgments

I would like to thank my advisor, Professor Geoffrey C. Fox, for his advice and encouragement throughout my research work. Without his valuable suggestions and continuous support this thesis could not have been completed.

This dissertation would not have been possible without the cooperation of several researchers and colleagues at Northeast Parallel Architecture Center (NPAC). I would like to extend my deepest thanks to Dr. Bryan Carpenter, for his guidance throughout my research work. His excellent suggestions and useful comments greatly helped me in preparing the final revision. It has been a great pleasure to work with him. I would also like to thank Dr. Paul Coddington, from whom I have learned much about parallel computing including random number generators and Monte Carol simulations. I am grateful to Mark Baker who guided and helped me port mpiJava onto Windows NT. I would like to thank all previous members of the HPJava group at NPAC such as Guansong Zhang, Xinying Li, Hanku Lee, and my officemate Sangboem Lim who cooperated with me in implementing mpiJava. Very Special thanks go to Sangyoon Oh for his hospitality during visit in Florida for the final revision of thesis. Many thanks go to Ilkyun Ra, Cheol-Min Hwang, Dongmin Kim, Sang-Gue Oh, Jungkee Kim, Kang-Seok Kim and all my friends for their concern and help during my long journey to my degree.

Finally, I would like to thank all my family. I owe a lot to my parents and my wife's parents. I could not have completed my degree without their support and advice. I appreciate the great support and patience from my wife Jiyeon and my son Yubin.

SUNG-HOON KO

Syracuse University

May 18, 2000

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 [52]

> "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."

and in fact recent development have emphasized server-side applications over applets.

Several of the concerns of the Java development 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 [15, 19, 73, 39, 66].

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 [48, 47].

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 [3, 24, 33, 38, 54, 101]

- 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. An interesting series of papers from IBM [76, 77, 105], confirmed that the current generation of Java virtual machines have rather poor performance on Fortran-like, array-intensive computations, but went on to demonstrate how to apply aggressive optimizations in Java compilers to obtain performance competitive with Fortran. In a recent paper [78] they described a case study involving a data mining application. Using the experimental IBM HPCJ Java compiler they reported obtaining over 90% of the performance of Fortran.

- 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 [69].

1.1 Issues

1.1.1 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 more interested in truly *distributed 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. 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 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.

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 heterogeneous networks. Conventionally, it was often necessary to install and compile an application on each target host before starting the task remotely.

1.1.2 Communication in Java

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

The Java 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 2.2.1, 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 [69, 43] or higher-level library interfaces [73].

- 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, 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.

1.2 Research Objectives

The main goal of the research presented in this dissertation is to investigate issues pertinent to high performance programming in Java, design Java interfaces to High Performance Computing softwares, and research implementation issues associated with their development. The dissertation has the following objectives:

- Investigate various issues and options for parallel programming in Java—data parallelism and message passing libraries. These include discussing motivations for introducing HPJava, an HPspmd programming model. It allows programs to combine data parallel code and SPMD library calls directly. We present sample code that directly calls to MPI from within the data parallel program.

- Develop mpiJava, an object-oriented Java interface to MPI. This could be implemented by "native methods" wrappers around some pre-existing MPI implementation. However, interfacing Java to MPI is not always trivial. The new *native* thread feature in JDK 1.2 might be a solution to eliminate conflicts between the Java runtime and the interrupt mechanisms used in the MPI implementations. Also, the syntax of mpiJava should be easy to understand and use, thus making it relatively simple for programmers with either a Java or Scientific background to take up.

- Incorporate automatic object serialization in mpiJava. In order to support derived datatypes or handle a multidimensional array, the Java object serialization model for marshalling general communication data in MPI-like APIs should be introduced, because the standard MPI approach describing user-defined types does not map very naturally into Java. In addition, we should evaluate overheads introduced by object serialization in mpiJava, then consider optimized methods that reduce serialization overheads.

- Develop real applications written in mpiJava and evaluate their performance. It is also interesting to compare the results of benchmarks written in Java with equivalent C or Fortran code. Applications include graphical visualization parallel Potts model simulation with GUI using Java AWT.

- Publically release mpiJava on a Web site. This includes complete source, makefiles, configuration scripts, compiled libraries for WMPI, test codes (based on the IBM MPI test suite), example applications, javadoc documentation, and installation and usage notes.

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 processes. The Java Grande Forum is a group looking at general issues for large-scale computing in Java. We also discuss Java for scientific parallel programming issues and look at work related to Java

Grande in Chapter 2. Chapter 3 introduce the design of our HPJava framework and its core features. HPJava is a programming language extended from Java to support SPMD parallel programming style. 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 MPI. Chapter 6 presents applications and analyzes the performance of Ising model simulation implemented using mpiJava, and compared with Fortran and C versions. Finally, Chapter 7 contains conclusion and the directions for future work. An earlier version of some of the material in this dissertation has been presented previously in the publications listed in Publications section on page 137.

1.4 Acknowledgements

This work was supported in part by the National Science Foundation Division of Advanced Computational Infrastructure and Research, contract number 9872125.

Chapter 2

Java for Parallel Programming

In this chapter we give a general overview of the activities and processes of the Java Grande Forum [61], a group looking at general issues for large-scale computing in Java, and then outline possible ways to handle inter-process communication. Finally we review some of parallel computing environments and approaches for Java.

The Java Grande Forum is addressing many of the issues that arise in using Java for High Performance Computing and Grande Applications. It is a coalition of industrial and academic partners. It has been organized with the goal of established consensus and recommendations on possible enhancements to the Java language and Java standards for large-scale ("Grande") applications. During the last few years it has held a series of ACM-supported workshops and conferences that have helped stimulate research on Java compilers and systems for Grande applications.

2.1 Java for Scientific Computing

The attractiveness of Java for scientific computing is being encouraged by bodies like Java Grande [61]. The first meeting of the Java Grande Forum was held at ACM 1998 Workshop on Java for High-Performance Network Computing 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 a large scale nature, potentially requiring any combination of computers, networks, I/O, and memory. Examples may include, 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, public dissemination and requests for comments on its recommendations. Two major working groups were initially defined: the Numerics Working Group and the Applications and Concurrency Working Group.

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

- floating-point arithmetic,
- complex arithmetic,
- multidimensional arrays,
- lightweight classes, and
- operator overloading.

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

The second major working group is Applications and Concurrency group. The activity of this group is particularly relevant to this dissertation. The goal of the 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 that can be taken to improve the language and environment. An important focus has been on the Java-specific frameworks designed to support concurrent and high-performance computing methodology.

Focusses of interest in the group include:

- critical JDK issues,
- benchmarking JVM performance,
- seamless (desktop) access to high-performance computers (computing portals), and
- other parallel and distributed computing issues.

Critical JDK issues are highest priority issues, mostly related to performance of RMI and object serialization. Other parallel and distributed computing issues include the availability of MPI-like message passing facilities for programming networks of computers.

Some time after initial Java Grande working groups were defined, a Message-Passing Working Group was formed in response to the independent development by several groups of Java APIs for MPI-like and PVM-like systems. An immediate goal was to discuss a common API for MPI-like Java libraries. An initial draft for a common API specification was distributed at Supercomputing '98 [22]. Since then the working group has met in San Francisco and Syracuse, and a Birds of a Feather meeting was held at Supercomputing '99. Minutes of meetings were published on the *java-mpi* mailing list and are available at `http://mailer.csit.fsu.edu/mailman/listinfo/java-mpi/`. To avoid confusion with standards published by the original MPI Forum (which is not presently convening) the nascent API is now called *MPJ*. Work reported in this dissertation has been significant input to this Message-Passing Working Group.

Information about the Java Grande Applications and Concurrency Working Group activities can be found at `http://www.javagrande.org`.

2.2 Communication in Parallel Programming

Communication is essential to do parallel programming in distributed systems. A distributed memory system consists of processors and their local memories connected by an interconnection network. Since there is no global memory, it is necessary to move data from one local memory to another, ultimately by means of message-passing. This may done through *send/receive* API calls which must be written into the application program or by some higher-level software.

In this section we discuss three different communication methods in Java: socket, RMI (Remote Method Invocation), and message passing libraries. The first approach is to use the socket interface in the standard Java API. The second is to work through Java's RMI. Finally the third is to work through a Java interface to the message-passing standard, MPI [73]. Message-passing remains one of the most effective and widely used communication paradigms in parallel computing.

2.2.1 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 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.

Carpenter et al [20] present a schematic outline of a distributed Conway's Game of Life program using Java socket interface. In this socket version example, they observed that

- Initialization of socket communication is a complex procedure and clearly it should not be coded anew for each application program.
- In their Life example the messages were contiguous byte vectors that could be transmitted efficiently through the *read* and *write* methods of the Java socket API. But 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, even with the simplified Java socket API, they concluded

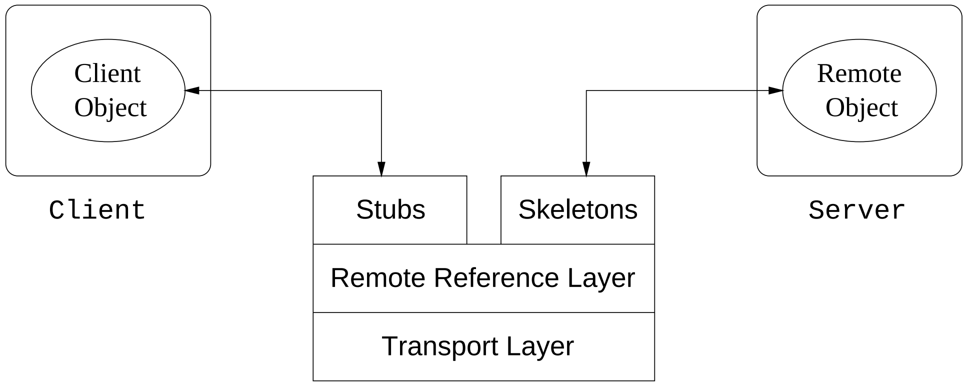


Figure 2.1: RMI Architecture.

that direct socket programming will probably remain unattractive to scientific parallel programmers, even with the simplified Java socket API.

2.2.2 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 a 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. An overview of the RMI model is shown in figure 2.1.

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*. 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 requests, monitoring them and listening for incoming calls.

High performance distributed computing can be done with Java RMI, but this has a few drawbacks since RMI is designed for client-server programming in Web based systems over slow networks. In parallel programming fast RMI with low latency and high bandwidth would be required. A better serialization would be needed, since Java's object serialization often takes 25%-50% of the time [87] needed for a remote invocation.

2.2.3 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 that support programs written in Java. Current message-passing in Java can be classified into two categories. The first category consists of Java based implementations that use Java as wrapper for existing message-passing libraries 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 systems where both Java and corresponding message-passing library are supported. The second category of Java based implementations use only pure Java libraries. Typically, in this case the underlying communication libraries are Java's RMI or sockets. The pure Java implementation obtains a high degree of code portability among platforms but may have lower performance. It is expected that the performance problem will be reduced with just-in-time (JIT) compilation and other compiler technologies.

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

2.2.3.1 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 machines. 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 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. *JavaPVM* (or jPVM) [62] 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 methods, it limits cross-platform portability. *JPVM* [36] is a pure Java implementation of PVM. Therefore unlike JavaPVM, it is better matched to 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 [106].

2.2.3.2 Message-Passing Interface (MPI) in Java

MPI [73, 55] 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 topologies and user defined data types. The first phase of MPI (MPI 1) was focussed 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 MPI interfaces for Java.

- An approach comparable to the work described in this dissertation to creating full Java MPI interfaces has been taken by University of Westminster [75, 50]. 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.

- *MPIJ* [79] 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 marshalling of primitive Java types. This technique allows MPIJ to achieve communication speeds comparable to native MPI implementations.

- *jmpi* [31, 32] is a pure Java implementation of MPI built on top of JPVM. Due to the additional wrapper layer to JPVM routines, its performance is expected to be poor compared to JavaPVM.

- *JMPI* [60] is a commercial effort underway at MPI Software Technology, Inc. to develop a message-passing framework and parallel support environment for Java. It aims to build a pure Java version of the MPI-2 standard specialized for commercial applications.

- Finally, *MPJ* [7] is the Java Grande Forum's MPI-like message-passing API proposed for both Java wrappers and a pure-Java versions. The aim is to provide an implementation of MPJ that is maximally portable.

2.3 Parallel Computing Environments for Java

In this section we review some of computing *environments* for Java that are under development. Currently several projects are under way that achieve increased bandwidth, reduced latency, and better reliability within workstation clusters using Java.

2.3.1 JavaParty

The *JavaParty* [88] 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

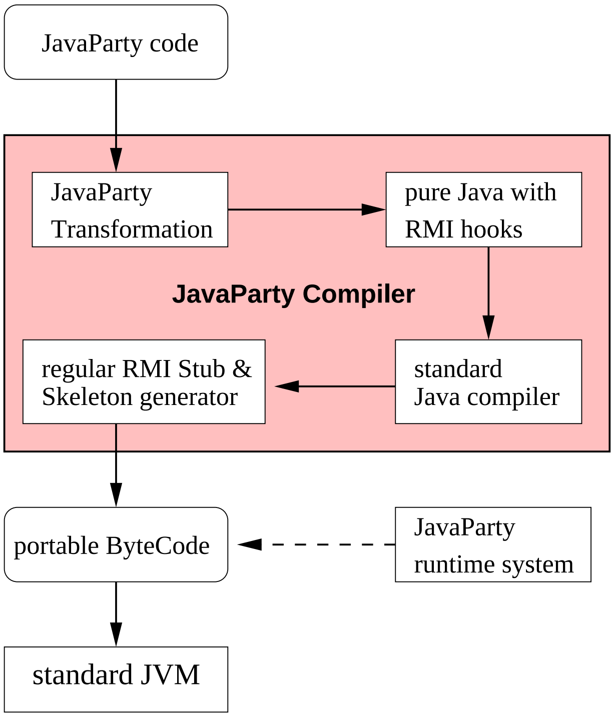


Figure 2.2: The JavaParty system.

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 by a pre-processing phase added to the Java compilers, EspressoGrinder [83] and Pizza [84]. As shown in the figure 2.2, JavaParty code is transformed into regular Java code plus RMI hooks. The resulting RMI portions are then fed into RMI compiler (stub and skeleton generator).

For example, the following program fragment

```
remote class B {
    T x = I;            // instance variable
    T foo(V z) { P }    // method
}
```

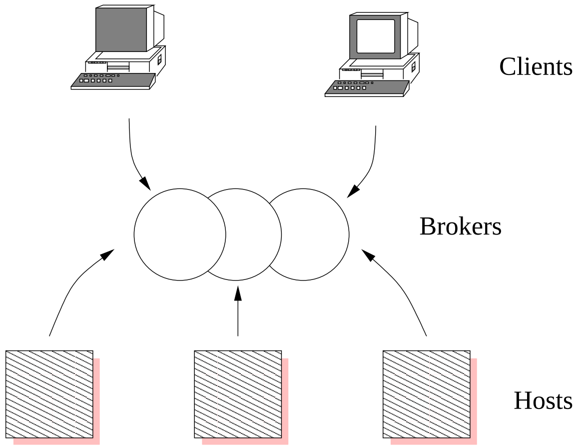


Figure 2.3: The Javelin Architecture.

may generate

```
interface B_intf extends RemoteIntf {...}
class B_impl extends RemoteInstance implements B_intf {
    T x = I;           // instance variable
    public T foo(V z) throws RemoteException {  // method
        toRemote(P)
    }
}
```

JavaParty is important contribution to research on Java-based parallel computing. Compared to approach in this dissertation, we expect to suffer from some unavoidable overhead from the RMI mechanism, relative to direct message-passing. Also there is relatively little evidence that the remote procedure call approach is actually convenient for SPMD programming.

2.3.2 Javelin

Javelin [27] from the University of California, Santa Barbara, is an Internet-based global computing infrastructures that supports Java. In Javelin, there are three kinds of 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 2.3 illustrates the 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.

The design is based on widely used components: Web browsers and the portable Java. By 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. 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.

It is good for very loosley-coupled parallel applications ("task parallelism") but it is less appropriate for the more tightly-coupled SPMD programming made possible by our approach.

2.3.3 Efficient RMI and Object Serialization

One of the goals of Java Grande Forum has been 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 project at the University of Karlsruhe, Germany, described in section 2.3.1, a more efficient RMI and object serialization has been studied. Philippsen et al [86] describe detailed analysis of RMI and object serialization and it's performance problems. They also suggest modifications to improve RMI and object serialization. Based on these suggestions, they implemented more efficient version of RMI and object serialization called KaRMI and UKA-serialization. *KaRMI* [80] 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 factories or port numbers. *UKA-serialization* [87] uses explicit marshalling/unmarshalling 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 *MantaRMI* [70], based on native compilation for better optimization. Manta [96] is compiler-based Java system and uses a native compiler that generates executable code rather than byte code. The 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. A 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 [67] 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. em Reflective RMI (RRMI) [95] 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 asynchronous remote method invocations.

Raje et al implement *ARMI* (Asynchronous RMI) [90] which adds an asynchronous remote method to RMI. It achieves asynchronous communication by replacing the RMI client-side stub, adding a thread mechanism.

NexusRMI [16] implements Java RMI on top of NexusJava [45], which is a Java interface to the Nexus communication system developed at Argonne National Laboratory. NexusRMI also has support for object serialization of all object types except for exception objects.

At the University of California, Berkeley, *NinjaRMI* [99] has been developed to support Ninja project [98]. NinjaRMI has some features that are not found in Sun's RMI including multiple communication protocols.

2.3.4 Other Systems

A number of approaches have been proposed for parallel computing using the Java language. In this section, we briefly descibe some of the ongoing projects.

Java/DSM [108] is heterogeneous computing system which implements a JVM on top of the Tread-Marks DSM (distributed shared memory) system [64]. In a 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.

JavaNow [11] 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.

Titanium [107] is a Java based language and system for high-performance parallel computing. It has additional features over Java like immutable classes, multi-dimensional 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. It does not provide any special support for distributed arrays, and the programming style is quite different to HPJava. Because the Titanium compiler translates Titanium into C, users do not have full access to the AWT or certain other Java libraries directly from Titanium.

Spar [97] is a Java-based language for array-parallel programming. 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. 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.

javar [12] 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.

The *WebFlow* [56] of Northeast Parallel Architectures Center, Syracuse University, is a particular programming paradigm implemented over a virtual Web accessible metacomputer, providing a dataflow-programming model. A 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 need not 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 [53] is a 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 goals of this metacomputing-based approach include better utilization of larger resource bases, as well as an enhanced collaborative environment for high performance computing. Among the challenges introduced by this idea are the numerous security issues introduced by code-upload capabilities for non-privileged uses.

In the next chapter we will review in detail our own HPJava parallel programming environment. This will provide a setting for the main original content of the dissertation.

Chapter 3

HPJava: A Java Environment for SPMD Programming

The Java message-passing library *mpiJava* discussed in the remaining of this dissertation is part of a larger environment called *HPJava*.

Early in the last decade, two groups formed for developing standards for parallel programming. The first, the High Performance Fortran Forum was formed by many leading industrial and academic groups in 1992. They established a language standard called High Performance Fortran (HPF) that allows programmers to easily write data-parallel programs. About a year later, the second group, the Message Passing Interface Forum, with participation from over 40 organizations, collected ideas from many previous message-passing systems and put them into a standard called MPI.

HPF provide a simple programming model in which all processors execute a single, logical thread of control that performs high-level operations on distributed arrays. In contrast, MPI provide multiple, logical threads of control that excute their own tasks in their own processors. HPF is compiler-based and MPI is library-based.

MPI has been quite successful and widely used for programming parallel system. On the other hand, HPF has been less successful, and many companies have abandoned their HPF projects. One of the failings is that HPF is not easily compatible with SPMD programming using MPI or many other similar libraries

Unlike HPF, our HPJava environment is based on an *HPspmd* model that is designed to facilitate direct calls to established SPMD libraries, including message-passing libraries, for parallel programming. To increase the flexibility of the system, and make it more attractive to programmers accustomed to the direct message-passing style, an interface to MPI is clearly essential.

The main emphasis is of this dissertation is mpiJava. mpiJava itself does not depend on any special language extension. For completeness, however, this chapter will review the language model of the full HPJava environment.

3.1 Motivation

Data parallel programming languages have always held a special position in the high-performance computing world. The basic implementation issues related to this paradigm are well understood. However, the choice of high-level programming environment, particularly for modern MIMD architectures, remains uncertain. In 1993, the High Performance Fortran Forum published the first standardized definition of a language for data parallel programming [57, 66]. In the intervening period considerable progress has been made in HPF compiler technology, and the HPF language definition has been extended and revised in response to demands of compiler-writers and end-users [40]. Yet it seems to be the case that most programmers developing parallel applications—or environments for parallel application development—do *not* code in HPF. The slow uptake of HPF can be attributed in part to immaturity in the current generation of compilers. But it seems likely that many programmers are actually more comfortable with the Single Program Multiple Data (SPMD) programming style, perhaps because the effect of executing an SPMD program is more controllable, and the process of tuning for efficiency is more intuitive.

SPMD programming has been very successful. There are countless applications written in the most basic SPMD style, using direct message-passing through MPI or similar low-level packages. Many higher-level parallel programming environments and libraries assume the SPMD style as their basic model. Examples include ScaLAPACK [14], PetSc [9], DAGH [85], Kelp [37], the Global Array Toolkit [81] and NWChem [10]. While there remains a prejudice that HPF is best suited for problems with very regular data structures and regular data access patterns, SPMD frameworks like DAGH and Kelp have been designed to deal directly with irregularly distributed data, and other libraries like CHAOS/PARTI [30] and Global Arrays support unstructured access to distributed arrays.

These successes aside, the library-based SPMD approach to data-parallel programming certainly lacks the uniformity and elegance of HPF. All the environments referred to above have some idea of a distributed array, but they all describe those arrays differently. Compared with HPF, creating distributed arrays and accessing their local and remote elements is clumsy and error-prone. Because the arrays are managed entirely in libraries, the compiler offers little support and no safety net of compile-time checking.

This chapter discusses a class of programming languages that borrow certain ideas, various run-time technologies, and some compilation techniques from HPF, but relinquish some of its basic tenets. In particular they forgo the principles that the programmer should write in a language with (logically) a single global thread of control, and that the compiler should determine automatically which processor executes individual computations in a program, then automatically insert communications if an individual computation involves accesses is to non-local array elements.

If these assumptions are removed from the HPF model, does anything useful remain? What will be retained is an explicitly MIMD (SPMD) programming model complemented by syntax for representing distributed arrays, and syntax for expressing that certain computations are localized to certain processors, including syntax for a distributed form of the parallel loop. The claim is that these features are adequate to make calls to various data-parallel libraries, including application-oriented libraries and high-level libraries for communication, about as convenient as, say, making a call to an array transformational intrinsic function in Fortran 90. Besides their advantages as a framework for library usage, the resulting programming languages can conveniently express various practical data-parallel algorithms. The resulting framework may also have better prospects for dealing effectively with *irregular* problems than is the case for HPF.

3.2 HPspmd Language Extensions

HPspmd system aims to provide a flexible hybrid of the data parallel and low-level SPMD paradigms. To this end HPF-like distributed arrays appear as language primitives. But a design decision is made that all access to *non-local* array elements should go through library functions—either calls to a collective communication library, or simply `get` and `put` functions for access to remote blocks of a distributed array. Clearly this decision 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.

For the newcomer to HPF, one of the advantages of that language lies in the fact that the effect of a particular operation is logically identical to its effect in the corresponding sequential program. Assuming programmers understand conventional Fortran, it is very easy for them to understand the behavior of a program at the level of what values are held in program variables, and the final results of procedures and programs. Unfortunately, the ease of understanding this "value semantics" of a program is counterbalanced by the difficulty in knowing exactly how the compiler translated the program. Understanding the *performance* of an HPF program may require that the programmer have rather detailed knowledge of how arrays are distributed over processor memories, and what strategy the compiler adopts for distributing computations.

The language model we discuss has a special relationship to the HPF model, but the HPF-style semantic equivalence between the data-parallel program and a sequential program is abandoned in favor of a simple equivalence between the data-parallel program and an MIMD (SPMD) program. Because understanding an SPMD program is presumably more difficult than understanding a sequential program, our language may be slightly harder to learn and use than HPF. But understanding performance of programs should be much easier.

The distributed arrays of an HPspmd language should be kept strictly separate from ordinary arrays. They are a different kind of object, not type-compatible with ordinary arrays. A property of the languages we describe is that if a section of program text *looks like* program text from the unenhanced base language (Fortran 90 or Java, for example), it is translated exactly as for the base language—as local sequential code. Only statements involving the extended syntax are treated specially. This makes preprocessor-based implementation of the new languages straightforward, allows sequential library code to be called directly, and gives programmers good control over the generated code—they can be confident no unexpected overhead have been introduced into code that looked like ordinary Fortran, for example.

Our HPspmd system adopts a distributed array model semantically equivalent to to the HPF data model in terms of how elements are stored, the options for distribution and alignment, and facilities for describing *regular sections* of arrays. Distributed arrays may be subscripted with global subscripts, as in HPF. But an array element reference must not imply access to a value held on a different processor. To simplify the task of the programmer, who must be sure accessed elements are held locally, the languages can add *distributed control* constructs. These play a role something like the `ON HOME` directives of HPF 2.0 and earlier data parallel languages [65]. One special control construct—a distributed parallel loop—facilitates traversal of locally held elements from a group of aligned arrays.

The HPJava is a Java instantiation language of this HPspmd language model. A brief review of the syntax extensions is given in section 3.4. In [21] we have outlined possible syntax extensions to Fortran to provide similar semantics to HPJava.

3.3 Integration of High-Level and Low-Level Libraries

Libraries are at the heart of our HPspmd model. From one point of view, the language extensions are simply a framework for invoking libraries that operate on distributed arrays. Hence an essential component of the ongoing work is definition of a series of bindings from HPspmd languages to established SPMD libraries and environments. Because the language model is explicitly SPMD, such bindings are a more straightforward proposition than in HPF, where one typically has to pass some extrinsic interface barrier before invoking SPMD-style functions.

We can group the existing SPMD libraries for data parallel programming into three categories. In the first category we have libraries like ScaLAPACK [14] and PetSc [9] where the primary focus is similar to conventional numerical libraries—providing implementations of standard matrix algorithms (say) but operating on elements in regularly distributed arrays. We assume that designing HPspmd interfaces to this kind of package will be relatively straightforward. ScaLAPACK for example, provides linear algebra routines for distributed-memory computers. These routines operate on distributed arrays—specifically, distributed matrices. The distribution formats supported are restricted to two-dimensional block-cyclic distribution for dense matrices and one-dimensional block distribution for narrow-band matrices. Since both these distribution formats are supported by HPspmd, using ScaLAPACK routines from the HPspmd framework should present no fundamental difficulties.

In a second category we place libraries conceived primarily as underlying support for general parallel programs with regular distributed arrays. They emphasize high-level communication primitives for particular styles of programming, rather than specific numerical algorithms. These libraries include compiler runtime libraries like Multiblock Parti [1] and Adlib [109], and application-level libraries like the Global Array toolkit [81]. Adlib is a runtime library that was designed to support HPF translation. It provides communication primitives similar to Multiblock PARTI, plus the Fortran 90 transformational intrinsics for arithmetic on distributed arrays. The Global Array (GA) toolkit, developed at Pacific Northwest National Lab, provides an efficient and portable "shared-memory" programming interface for distributed-memory computers. Each process in a MIMD parallel program can asynchronously access logical blocks of distributed arrays, without need for explicit cooperation by other processes ("one-sided communication"). Besides providing a more tractable interface for creation of multidimensional distributed arrays, our syntax extensions should provide a more convenient interface to primitives like `ga_get`, which copies a patch of a global array to a local array.

Regular problems (such as the linear algebra examples in section 3.4) are an important subset of parallel applications, but of course they are far from exclusive. Many important problems involve data structures too irregular to represent purely through HPF-style distributed arrays. Our third category of libraries therefore includes libraries designed to support irregular problems. These include CHAOS [30] and DAGH [85]. We anticipate that irregular problems will still benefit from regular data-parallel language extensions—at some level they usually resort to representations involving regular arrays. But lower level SPMD programming, facilitated by specialized class libraries, is likely to take a more important role. For an HPspmd binding of the CHAOS/PARTI library, for example, the simplest assumption is that the preprocessing phases yield new arrays. Indirection arrays may well be left as HPspmd distributed arrays; data arrays may be reduced to ordinary Java arrays holding local elements. Parallel loops of an executor phase can then be expressed using *overall* constructs. More advanced schemes may incorporate irregular maps into generalized array descriptors [40, 34, 26] and require extensions to the baseline HPspmd language model.

3.4 The HPJava Language

HPJava [23] is an instance of the HPsmpd language model. HPJava extends its base language, Java, by adding some predefined classes and some additional syntax for dealing with distributed arrays.

As explained in the previous section, the underlying distributed array model is equivalent to the HPF array model. Array mapping is described in terms of a slightly different set of basic concepts. *Process group* objects generalize the processor arrangements of HPF. *Distributed range* objects are used instead HPF templates. A distributed range is comparable with a single dimension of an HPF template. These substitutions are a change of parameterization only. Groups and ranges fit better with our distributed control constructs.

Figure 3.1 is a simple example of an HPJava program. It illustrates creation of distributed arrays, and access to their elements. The class `Procs2` is a standard library class derived from the special base class `Group`. It represents a two-dimensional

```
Procs2 p = new Procs2(P, P) ;
on(p) {
  Range x = new BlockRange(M, p.dim(0)) ;
  Range y = new BlockRange(N, p.dim(1)) ;

  float [[,]] a = new float [[x, y]], b = new float [[x, y]],
              c = new float [[x, y]] ;

  ... initialize values in 'a', 'b'

  overall(i = x for :)
    overall(j = y for :)
      c [i, j] = a [i, j] + b [i, j] ;
}
```

Figure 3.1: A parallel matrix addition.

grid of processes. Similarly the distributed range class `BlockRange` is a library class derived from the special class `Range`; it denotes a range of subscripts distributed with BLOCK distribution format over a specific process dimension. Process dimensions associated with a grid are returned by the `dim()` inquiry. The `on(p)` construct is a new control construct specifying that the enclosed actions are performed only by processes in group `p`.

The variables `a`, `b` and `c` are all distributed array objects. The type signature of an r-dimensional distributed array involves double brackets surrounding r comma-separated slots. The constructors specify that these all have ranges `x` and `y`—they are all `M` by `N` arrays, block-distributed over `p`.

A second new control construct, *overall*, implements a distributed parallel loop. The constructs here iterates over all locations (selected by the degenerate interval " `:` ") of ranges `x` and `y`. The symbols `i` and `j` scoped by these constructs are called *distributed indices*. In HPF, a distributed array element is referenced using integer

```
Procs2 p = new Procs2(P, P) ;
on(p) {
  Range x = new ExtBlockRange(N, p.dim(0), 1, 1) ;
  Range y = new ExtBlockRange(N, p.dim(1), 1, 1) ;

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

  ... some code to initialise 'u'

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

    Adlib.writeHalo(u) ;

    overall(i = x for 1 : N - 2)
      overall(j = y for 1 + (i` + iter) % 2 : N - 2 : 2)
        u [i, j] = 0.25 * (u [i - 1, j] + u [i + 1, j] +
                           u [i, j - 1] + u [i, j + 1]) ;
  }
}
```

Figure 3.2: Red-black iteration.

subscripts, like an ordinary array. In HPJava, with a couple of exceptions noted below, the subscripts in element references must be distributed indices, and these must be bound by an overall construct to a location in the range associated with the array dimension. This rather drastic restriction is a principal means of ensuring that referenced array elements are held locally.

The general policy is relaxed slightly to simplify coding of stencil updates. A subscript can be a *shifted location*. Usually this is only legal if the subscripted array is declared with suitable *ghost regions* [49]. Figure 3.2 illustrates the use of the library class `ExtBlockRange` to create arrays with ghost extensions (in this case, extensions of width 1 on either side of the locally held "physical" segment). The communication library routine `Adlib.writeHalo` updates the ghost region. This

```
Procs1 p = new Procs1(P) ;
on(p) {
  Range x = new BlockRange(N, p.dim(0)) ;

  float [[,*]] a = new float [[x, N]], c = new float [[x, N]] ;
  float [[*,]] b = new float [[N, x]], tmp = new float [[N, x]] ;

  ... initialize ‘a’, ‘b’

  for(int s = 0 ; s < N ; s++) {

    overall(i = x for :) {

      float sum = 0 ;
      for(int j = 0 ; j < N ; j++)
        sum += a [i, j] * b [j, i] ;

      c [i, (i‘ + s) % N] = sum ;
    }

    // cyclically shift ‘b’ (by amount 1 in x dim)...

    Adlib.cshift(tmp, b, 1, 1) ;
    HPspmd.copy(b, tmp) ;
  }
}
```

Figure 3.3: A pipelined matrix multiplication program.

example also illustrates application of a postfix back-quote operator to a bound location. The expression `i`` (read "i-primed") yields the integer global loop index.

Distributed arrays can be defined with some sequential dimensions. The sequential attribute of an array dimension is flagged by an asterisk in the type signature. As illustrated in Figure 3.3, element reference subscripts in sequential dimensions can be ordinary integer expressions.

The short examples here have already covered much of the special syntax of HPJava. Other significant extensions allow Fortran-90-like *sections* of distributed arrays. This, in turn, forces us to define certain *subranges* and *subgroups*. Arrays constructed directly using subgroups and subranges can reproduce all the alignment options of HPF. In any case, the language itself is relatively simple. Complexities associated with varied and irregular patterns of communication are dealt with in libraries. These can implement many richer operations than the `writeHalo` and `cshift` functions of the examples.

3.5 MPI Programming in HPJava

To increase the flexibility of the system, and make it more attractive to programmers accustomed to the direct message-passing style, an interface to MPI is clearly desirable. In HPF, with its global-thread-of-control model, a proper interface to the underlying message-passing platform is only practical through the *extrinsic procedure* mechanism. Unlike HPF, the HPJava environment is based on the HPspmd model which is designed to facilitate direct access to SPMD library interfaces. Different processors can either simultaneously work on data with the globally subscripted arrays, or independently execute complex procedures on their own local data. The conversion between these phase is relatively seamless.

We use a simple example extracted from an N-body algorithm to illustrate the

```
Procs1 p = new Procs1(P) ;
on(p) {
  Range x = new BlockRange(N, p.dim(0)) ;

  float [[]] f = new float [[x]], a = new float [[x]],
             b = new float [[x]] ;

  // initialize
  overall(i = x for :) {
    f[i] = 0 ;
    b[i] = a[i] ;
  }

  for(int s = 0 ; s < N ; s++) {

    overall(i = x for :)
      f[i] += force(a[i], b[i]) ;

    // cyclically shift ‘b’ (by amount 1 in x dim)...

    Adlib.cshift(tmp, b, 1, 0) ;
    HPspmd.copy(b, tmp) ;

  }
}
```

Figure 3.4: HPJava data parallel version of the N-body.

usage of the HPJava binding for MPI. Figure 3.4 is a simplified pure data parallel version of force computation in a N-body program. There are three distributed arrays in the program, `a`, `b` and `f`. Distributed array `a` has fixed copy of the particle positions and `b` has circulating copy of the positions

Now suppose we wanted to combine features of this HPJava version with an MPI version, while partially keeping the convenient data-parallel array syntax. Figure 3.5 shows HPJava/MPI version of N-body. Process group arrangements and initialization for distributed arrays are handled by HPJava, while the shift-operation is done by calling the mpiJava `MPI.Sendrecv_replace` point-to-point communication routine between processors. The local variables `a_block`, `b_block` and `f_block` in the program are not distributed arrays, but they are assigned different values by distributed arrays `a`, `b` and `f` according to their position in the process grid. A inquiry function `dat()` returns a sequential Java array containing the locally held elements of the distributed array. The array size of local variables for each processor is $B = N/P$, where N is total particle size and P is number of processors. These local arrays run serially on each node, but concurrently across all the nodes.

Comparing this HPJava/MPI version with the data-parallel version, notice that both programs send the same amount of total data, but the HPJava/MPI version does P shifts of whole blocks of size B for sending N data, while the pure data-parallel version uses a `cshift` call that does one circular shift for every step for sending N data. This adds extra communication start-up overheads. In addition, HPJava version requires more copying operations (N times) than the mpiJava version does (P times), where typically $P \ll N$.

This example leaves non-trivial issues unresolved—in general what is the mapping from distributed-array elements to local-data-segment elements? What is the mapping between HPJava process groups and MPI groups? The complete specification of HPJava will address these issues; eventually a better integrated message-passing API may be desirable.

3.6 Discussion

In this chapter we discussed motivations for introducing an HPspmd programming model: a SPMD framework for using libraries based on distributed arrays. It adopts the model of distributed arrays standardized by the HPF Forum, but relinquishes

```
Procs1 p = new Procs1(P) ;
on(p) {
  Range x = new BlockRange(N, p.dim(0)) ;

  float [[]] f = new float [[x]], a = new float [[x]],
             b = new float [[x]] ;

  // initialize
  overall(i = x for :) {
    f[i] = 0 ;
    b[i] = a[i] ;
  }

  // variables containing local array elements of distributed arrays

  float [] a_block = a.dat() ;
  float [] b_block = b.dat() ;
  float [] f_block = f.dat() ;

  for(int s = 0 ; s < P ; s++) {

    for(int i = 0 ; i < B ; i++)  // B : local block size
      for(int j = 0 ; j < B ; j++)
        f_block[i] += force(a_block[i], b_block[j]) ;


    // cyclically shift ‘b_block’ (by amount B in x dim)...

    MPI.COMM_WORLD.Sendrecv_replace(b_block, 0, B, MPI.FLOAT,
                                    right, 99, left, 99) ;

  }

}
```

Figure 3.5: Data parallel version of the "N-body" example with MPI communications.

the high-level single-threaded model of the HPF language. This makes compilers or translators for the HPspmd-extended languages a relatively straightforward proposition. As a concrete example, we described the specific syntax of HPJava.

Two recent languages that have some similarities to our HPspmd languages are F-- and ZPL. F-- [82] is an extended Fortran dialect for SPMD programming. The approach is different to the one proposed here. 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 our basic philosophy is to provide communication through separate libraries. ZPL [92] is a array parallel programming language for scientific computations. It has a construct for performing computations over a *region*, or set of indices, quite similar to our *overall* construct. Communication is more explicit than HPF, but not as explicit as in the language discussed in this article.

At the time of writing the HPJava translator is partially operational. All examples given in this chapter can be compiled and run. Ongoing work will complete the functionality, and add some optimization for the generated code. The language definition calls for full compile-time or runtime checking of the constraints on locality of reference. The translator will be enhanced to add these.

In the next chapter we will further discuss the mpiJava library briefly mentioned in the previous section.

Chapter 4

mpiJava: An Object-Oriented Java Interface to MPI

In this chapter, we turn to the main subject of this dissertation, *mpiJava*—a Java interface to the widely used Message Passing Interface (MPI).

In the first part of this chapter, we discuss the design of the mpiJava API and issues associated with its development. In the second part of this chapter, we briefly outline the steps necessary to "port" mpiJava onto a range of operating systems, including Windows NT, Linux and Solaris. In the final part of this chapter, we present and then discuss some performance measurements made of communications bandwidth and latency to compare mpiJava on these systems.

4.1 Motivation

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 a typical implicit means might be 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 [73]. 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) [68, 51]. The software was originally publically available from:

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

It has moved to

`http://aspen.csit.fsu.edu/pss/HPJava/mpiJava.html`

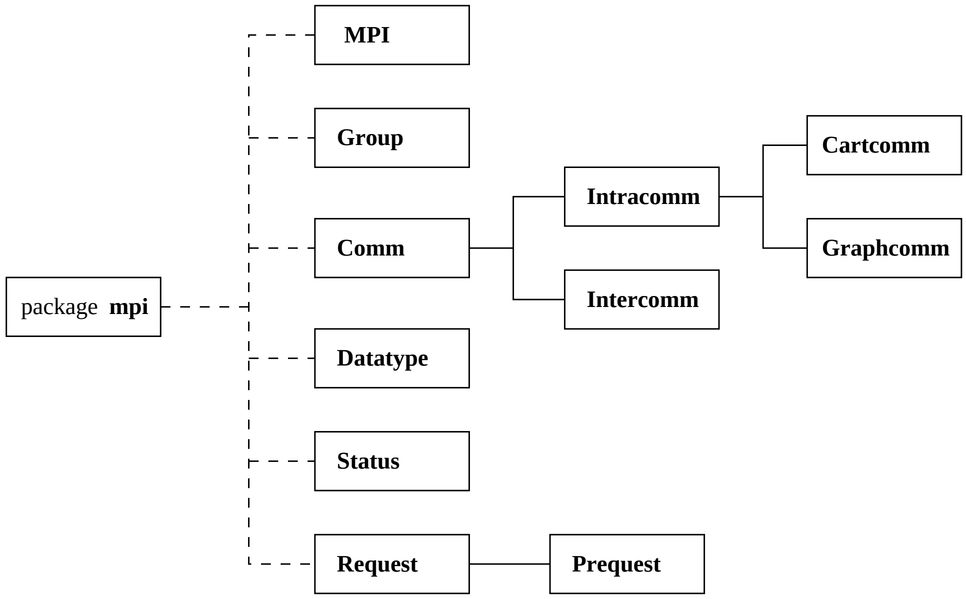


Figure 4.1: Principal classes of mpiJava

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.

4.2 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

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.OBJECT	`Object`

Table 4.1: Basic datatypes of mpiJava

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 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 `MPI.OBJECT` is a new predefined datatype for derived data types.

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
```

```
// 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)

```
                    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. 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.

4.2.1 Language Binding

A simple "HelloWorld" mpiJava program is shown in figure 4.2. The mpiJava API is modelled 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 this section we discuss the issues and special features of Java binding. As a result of the following changes mpiJava argument lists are often more concise than the corresponding C or C++ argument lists.

4.2.1.1 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.

4.2.1.2 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`.

4.2.1.3 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 4.2.2.

4.2.1.4 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.*

4.2.1.5 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.

4.2.1.6 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.)

4.2.1.7 Multiple Return Values.

A few functions in the MPI interface return multiple values, even after the error code is eliminated. 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 typically return an object reference which is `null` if the operation fails. Occasionally extra internal fields are added to an existing MPI class to hold extra results—for example the `Status` class has an extra field initialized by functions like `Waitany` to hold the *index* value. Rarely none of these methods work and we resort to defining auxiliary classes to hold multiple results from a particular function.

4.2.1.8 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.

4.2.1.9 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 [51] 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 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.

4.2.2 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.

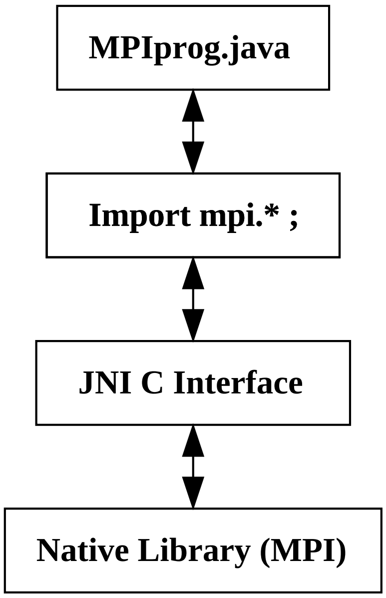


Figure 4.3: Software Layers

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

To port mpiJava onto a new platform, generally two steps are needed.

- 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("mpijava")` call in the main source file `MPI.java`.

4.3.1 UNIX Platforms

The development and testing of mpiJava was undertaken on various Sun and SGI UNIX platforms using MPICH. Interfacing Java to MPI is not always trivial. In earlier implementations 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. As a by-product this 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 .

4.3.2 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.

4.3.2.1 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 [8]. We chose WMPI [102] 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 [18] device standard. 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.

To create a release of mpiJava for WMPI, the mpiJava JNI C interface was compiled into a Win32 Dynamic Link Library (`mpiJava.dll`), then the Java MPI interface was compiled into class libraries, and finally a JNI interface to WMPI was created.

4.3.2.2 Linux

With previous version of JDK1.1.7 for Linux from Blackdown [13], we had problems similar to those encountered during our early attempts to create the interface on Solaris, mentioned in section 4.3.1. Sun 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 or later. It is also tested sucessfully with Sun's JDK1.2.2 for Linux. The IBM alpha-version 1.1.6 of JDK [59] for Linux was released in 1999. IBM JDK1.1.6 uses *native* threads as the underlying implementation. Blackdown and Sun JDK1.2 supports both *green* and *native* threads.

The most recent release of the Blackdown JDK for Linux is 1.2.2-RC4 (Release Candidate 4) that requires at least a glibc-2.1.2 based system with a recent 2.2.xx kernel.

We have tested mpiJava with Mandrake 6.0 [71], Redhat 6.2 [91] Linux. These Linux packages incorporate both glibc version 2.1.x and Linux kernel version 2.2.x.

4.3.3 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"—or as multiple processes running on separate machines—"distributed memory mode". Under WMPI and Solaris-MPICH all the codes ran in both modes without alterations.

4.4 Simple Communications Performance Measurement

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.

4.4.1 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` and `MPI_Recv`. PingPong provides information about latency of `MPI_Send` and `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 [6]. 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 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.

4.4.2 Message Startup Latencies

In Table 4.2 we show the transmission time in microseconds (μs) to send a 1 byte message in each of the environments tested. In shared memory mode (SM) the mpiJava wrapper adds an extra 94μs (140%) and 226μs (152%) compared to WMPI and MPICH C respectively. In distributed memory mode (DM) the mpiJava wrapper adds an extra 66μs (11%) and 282μs (42%) compared to WMPI and MPICH

	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

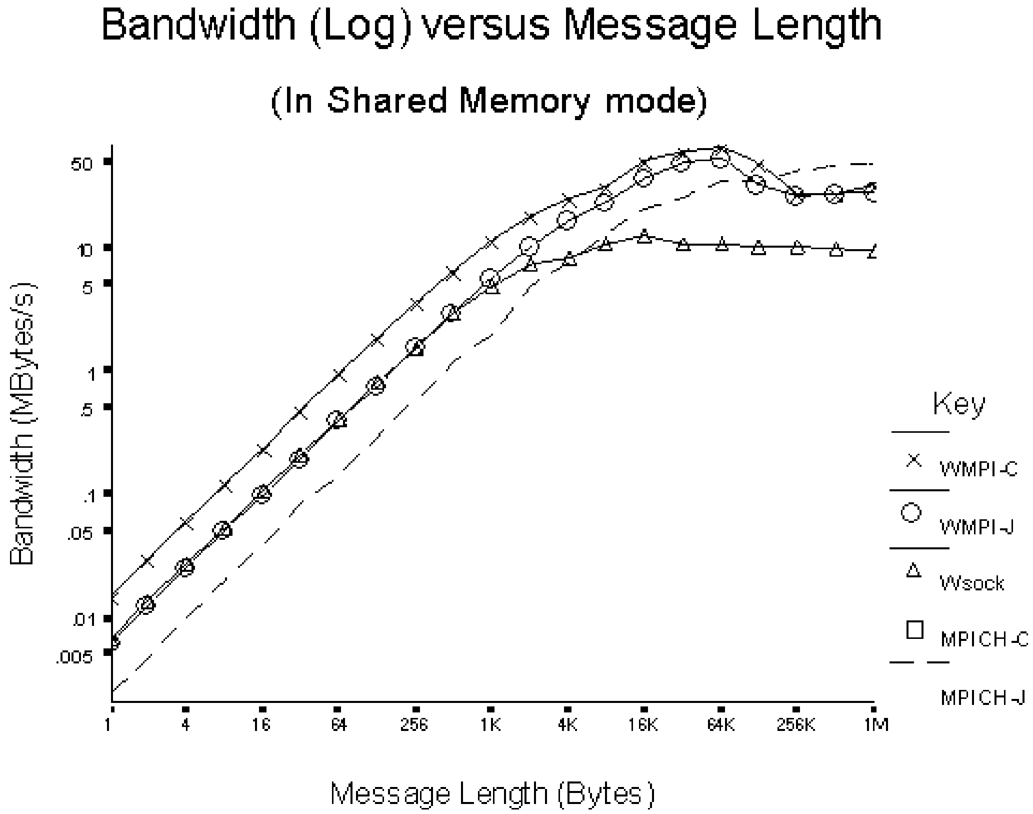


Figure 4.4: PingPong Results in Shared Memory (SM) mode

C respectively. The "Wsock" figures are those for a WinSock implementation of PingPong benchmark using TCP.

4.4.3 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.

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

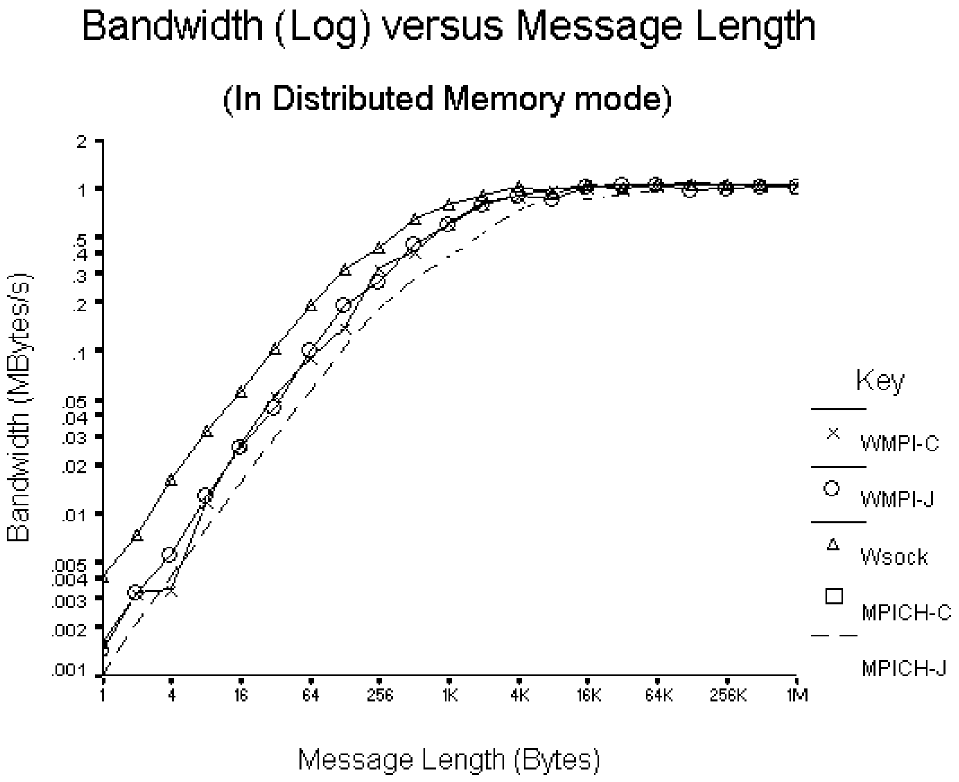


Figure 4.5: PingPong Results in Distributed Memory (DM) mode

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.

4.4.4 Results in Distributed Memory Mode (Figure 4.5)

In distributed memory mode the differences between the MPI codes is not as pronounced as seen in shared memory mode. 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.

4.5 Discussion

We have discussed the design and development of mpiJava—a pure Java interface to MPI. Our performance tests have shown that mpiJava should fulfill the needs of MPI programmers not only in terms of functionality but also in terms of good performance when compared to similar C MPI programs.

In both shared memory and distributed memory modes mpiJava adds a fairly constant overhead compared to normal native MPI. In an environment like WMPI, which has been optimized for Windows 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. The quality and performance of JVM on each platform will have the greatest effect on the usefulness of mpiJava for scientific computation.

We also suggested required environment to make mpiJava run on different platforms including Linux. On these platforms, several parallel applications including Potts model simulation using Java AWT for GUI have been implemented. These applications will be covered in Chapter 6 with their performance evaluation.

Chapter 5

Object Serialization for Marshalling Data in mpiJava

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 section draws some conclusion.

5.1 Motivation

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 the previous section 4.2.2 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 [22], 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 [22] 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 4.1 in section 4.2.

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

```
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 `arrayOfDisplace-`

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

`ments` 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.

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 [22]—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 fields 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.

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 [74]. 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 5.1.

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

$$
\begin{aligned}
t^{\mathtt{T[}N^2\mathtt{]}} &= c + t^{\mathrm{T}}_{\mathrm{com}} N^2 && (5.1)\\
t^{\mathtt{T[1][}N^2\mathtt{]}} &= c' + (t^{\mathrm{T}}_{\mathrm{ser}} + t^{\mathrm{T}}_{\mathrm{com}} + t^{\mathrm{T}}_{\mathrm{unser}}) N^2 && (5.2)\\
t^{\mathtt{T[}N\mathtt{][}N\mathtt{]}} &= c'' + (t^{\mathrm{vec}}_{\mathrm{ser}} + t^{\mathrm{vec}}_{\mathrm{unser}}) N + \\
&\quad (t^{\mathrm{T}}_{\mathrm{ser}} + t^{\mathrm{T}}_{\mathrm{com}} + t^{\mathrm{T}}_{\mathrm{unser}}) N^2 && (5.3)
\end{aligned}
$$

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 5.1: 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

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

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

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 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 $t^{\texttt{T[}N^2\texttt{]}}$. Table 5.1 contains the resulting estimates of the various parameters for `byte` and `float` elements.

Figures 5.2 and 5.3 on Page 74 and 75 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 $t^{\texttt{T[}N^2\texttt{]}}$, $t^{\texttt{T[1][}N^2\texttt{]}}$ and $t^{\texttt{T[}N\texttt{][}N\texttt{]}}$ 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 dom-

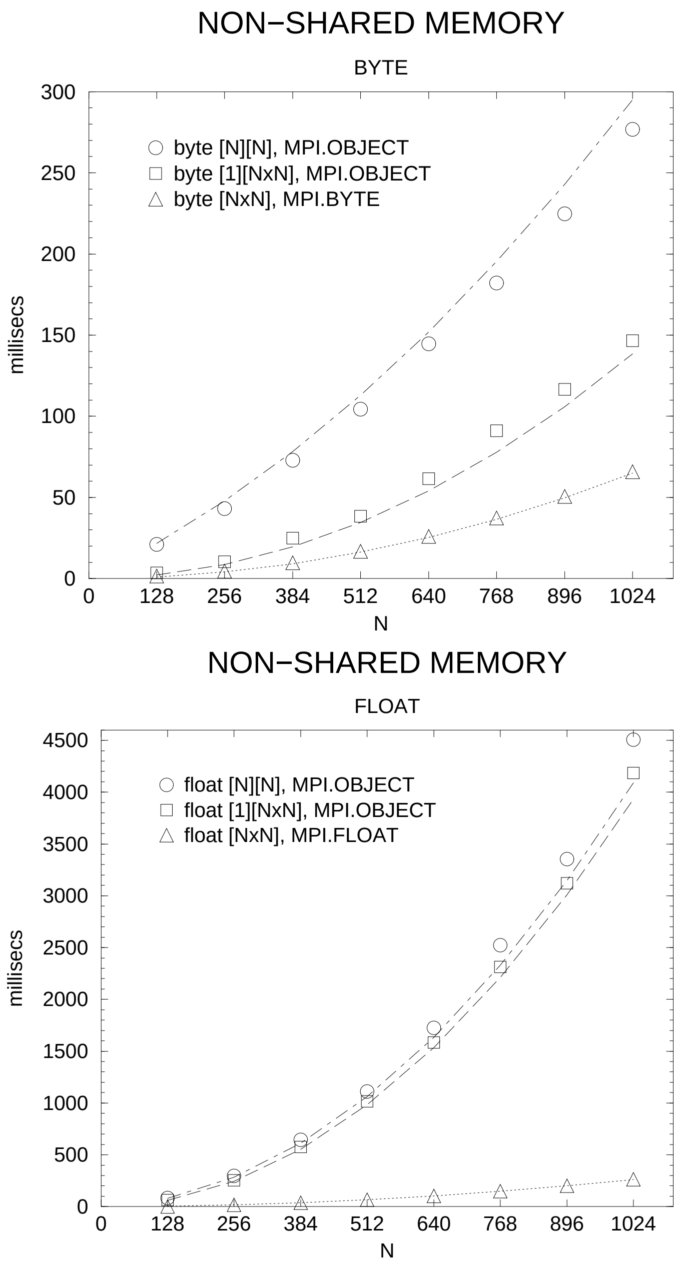


Figure 5.2: Communication times from ping-pong benchmark in non-shared-memory case. The lines represent the model defined by Equations 5.1 to 5.3 in the text, with parameters from Table 5.1.

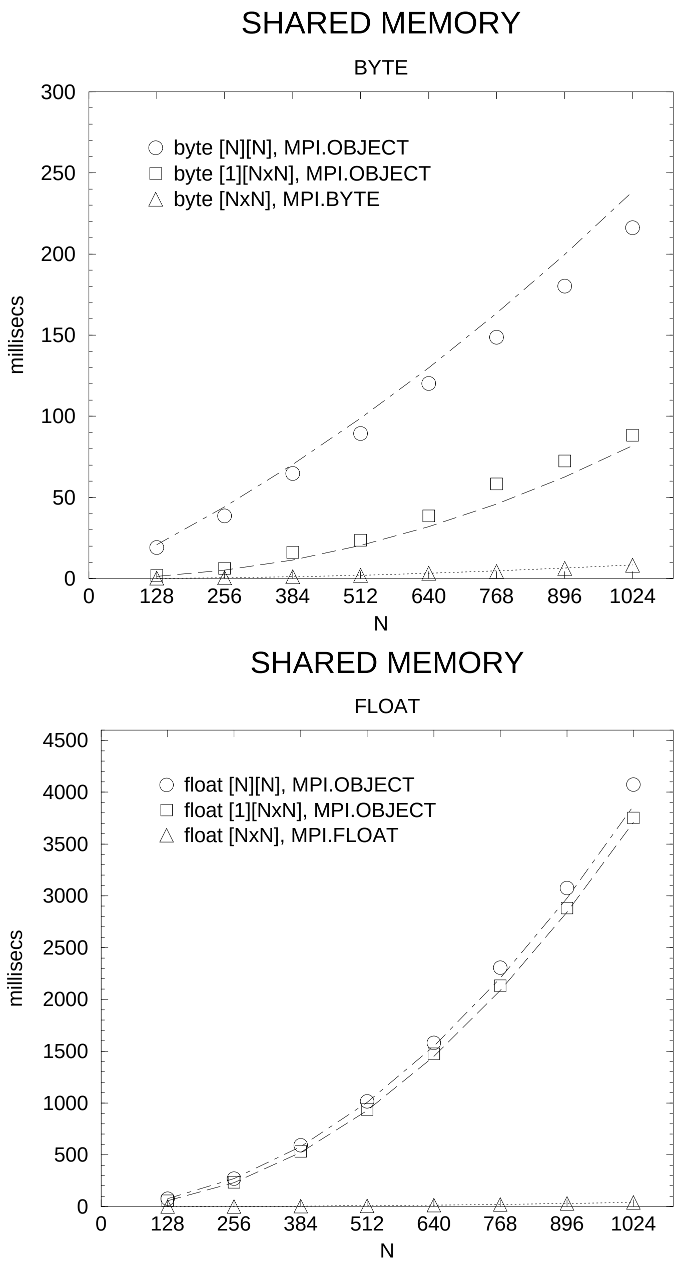


Figure 5.3: Communication times from ping-pong benchmark in shared-memory case. The lines represent the model defined by Equations 5.1 to 5.3 in the text, with parameters from Table 5.1.

inates 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 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 $t^{\text{vec}}_{\text{ser}}$ and $t^{\text{vec}}_{\text{unser}}$ 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

The work of [87] and others mentioned in section 2.3.3, 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

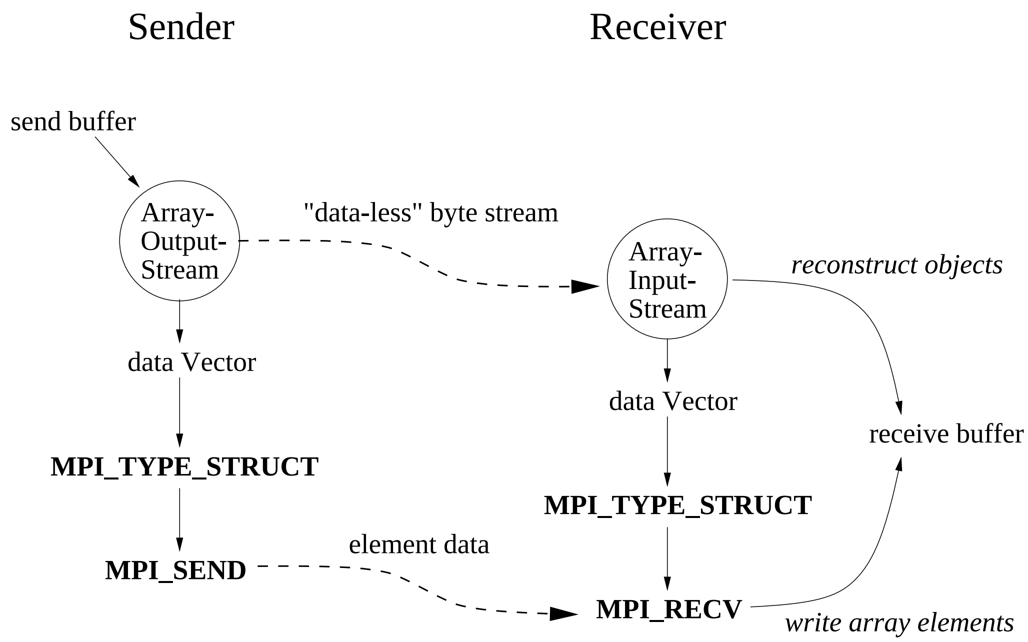


Figure 5.4: Improved protocol for handling arrays of primitive elements.

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 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 5.4.

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 5.5.

Figures 5.6 and 5.7 on Page 80 and 81 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

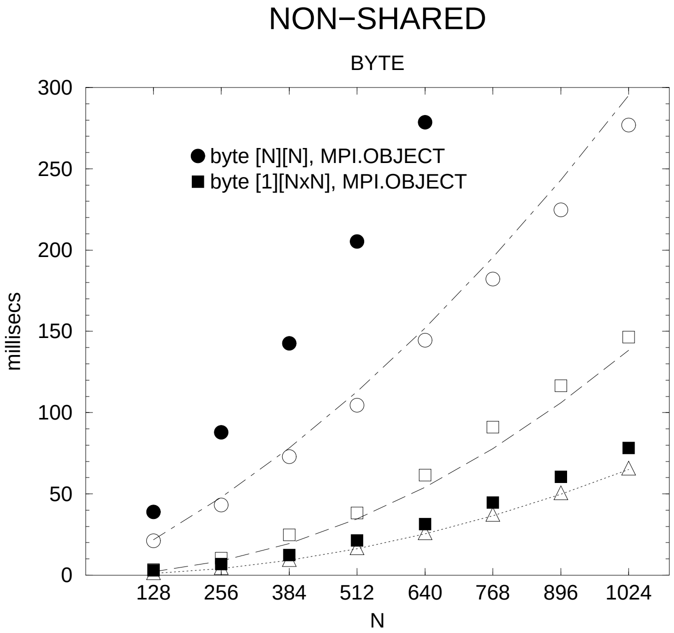
```
class ArrayOutputStream extends ObjectOutputStream {
    Vector dataVector ;

    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 5.5: Pseudocode for `ArrayOutputStream` and `ArrayInputStream`



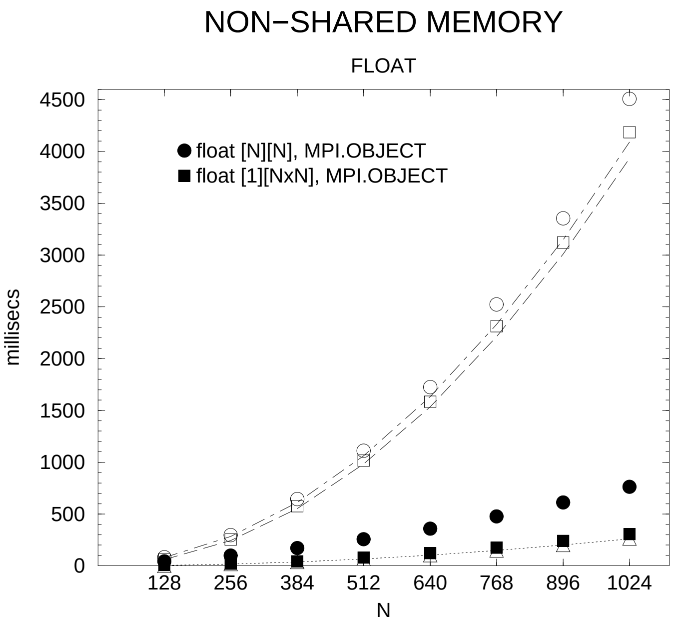


Figure 5.6: Ping-pong timings with primitive array data sent separately (solid points) in distributed memory mode, compared with the unoptimized results from Figure 5.2 (open points). Recall that the goal is to bring times for "object-oriented" sends of arrays down to the "native" send times, most closely approximated by the triangular points.

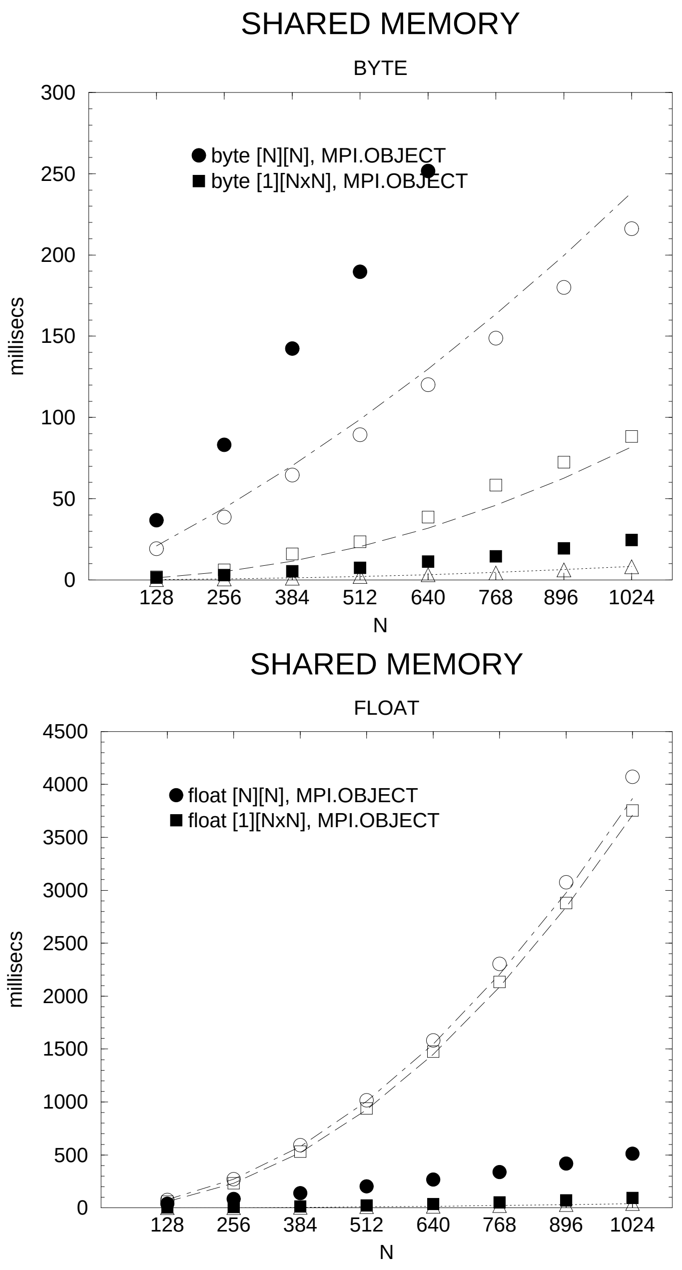


Figure 5.7: Ping-pong timings with primitive array data sent separately (solid points) in shared memory mode, compared with the unoptimized results from Figure 5.3 (open points). Recall that the goal is to bring times for "object-oriented" sends of arrays down to the "native" send times, most closely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

the constant overhead of communicating each subarray (parametrized by $t^{\text{vec}}_{\text{ser}}$ and $t^{\text{vec}}_{\text{unser}}$ 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 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 5.8 and 5.9 on Page 83 and 84 .

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.

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

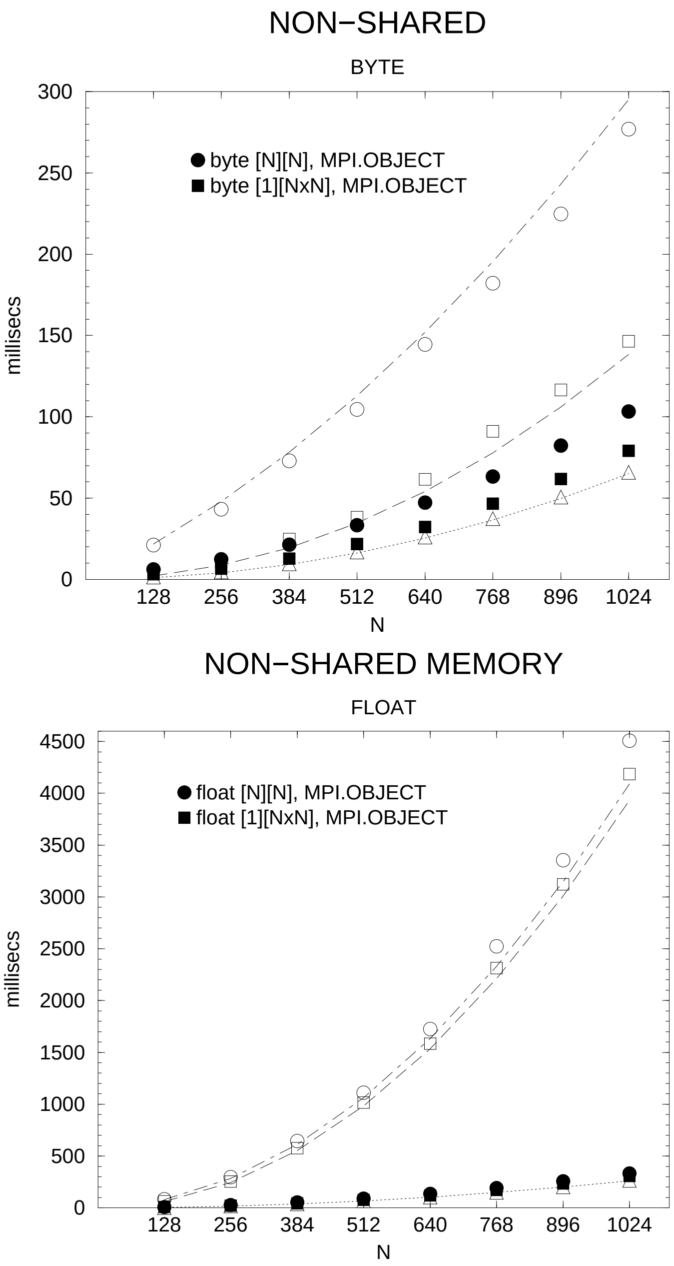


Figure 5.8: Timings allowing *two-dimensional* array proxies in the object stream (solid points) in distributed memory mode, compared with the unoptimized results from Figure 5.2 (open points). Sends of two-dimensional Java arrays (solid circles) are now much closer to the native bandwidth (of which the triangular points are representative).

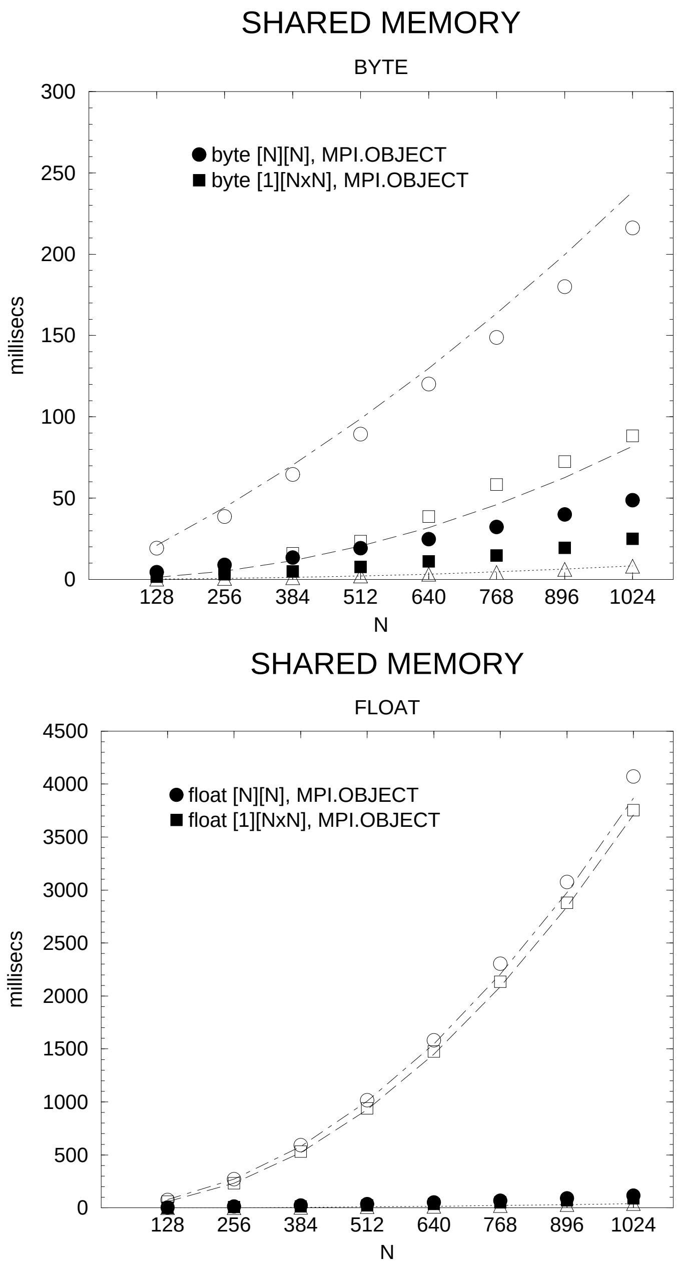


Figure 5.9: Timings allowing *two-dimensional* array proxies in the object stream (solid points) in shared memory mode, compared with the unoptimized results from Figure 5.3 (open points). Sends of two-dimensional Java arrays (solid circles) are now much closer to the native bandwidth (of which the triangular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

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 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 [41]. 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 [80]. 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.

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 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 [25, 100]. 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 the mpiJava software package. In this chapter we discuss two parallel Monte Carlo simulations which can be used for educational purposes and experimental performance evaluation of mpiJava with a real application.

In the first section, we briefly introduce a Java version of parallel random number generator which is used in two Monte Carlo spin model simulations—the Ising model and the Potts model. Then we discuss the mpiJava version of Monte Carlo simulation of the Ising model implemented with two different algorithms, Metropolis and cluster algorithms. This Ising model simulation can be used as a full application Java benchmark, with comparable native MPI-C and MPI-Fortran codes for performance comparison. This is presented in the last section. We also present a 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. It has a 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 [29].

Random number generators use iterative deterministic algorithms for producing a sequence X_i of pseudo-random numbers that approximate a truly random sequence. Probably the most commonly-used algorithms for sequential random number generators are Linear Congruential Generators (LCGs). 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 [2, 28], 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 mpiJava 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 the mpiJava Potts model and Ising model simulations, we use the 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.) [46]. 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 Monte Carlo Simulations of Spin Models

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

6.2.1 The Potts Model

The Potts model [89, 104] 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 $\sigma_i = 0, 1, 2, ..., Q-1$, and 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 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 [35] was invented by Metropolis et al in 1953 for generating an arbitrary probability distribution. This algorithm has been very successful and influential in many the areas of Monte Carlo method.

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.

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 the 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 $r < \exp(-\beta * \Delta E)$. The algorithm is 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 a small amount then the changes will usually be accepted but their small size will make exploration

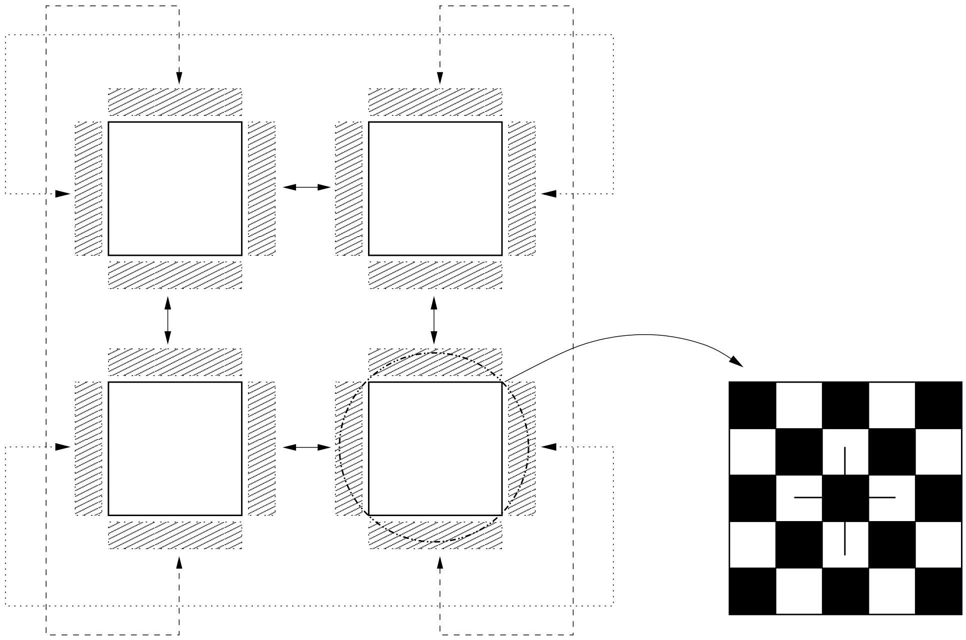


Figure 6.3: Checkerboard Partition and Blocked Communication.

of phase space rather slow.

The Metropolis algorithm for a spin model is well suited to parallelism, since it is regular—so one can use standard data decomposition of the lattice onto the processors to 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 cannot update a site and its neighbors at the same time, since they are dependent on each other, and we would violate *detailed balance*. Detailed balance is a sufficient condition for equilibrium probability distribution and guarantees 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 sites in parallel (at once), since they are all independent, followed by all red sites in parallel. This method is 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 that can be updated at once.

Cluster algorithms were proposed by Swendsen and Wang [93] 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 [103]. 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 from the old one. These parallel Wolff cluster algorithms were developed in several different languages including C code for the CM-5 using CMMD [94] which was

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.

presented in [4].

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 [58] 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

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

Therefore 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 [58].

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 form 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.

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 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)

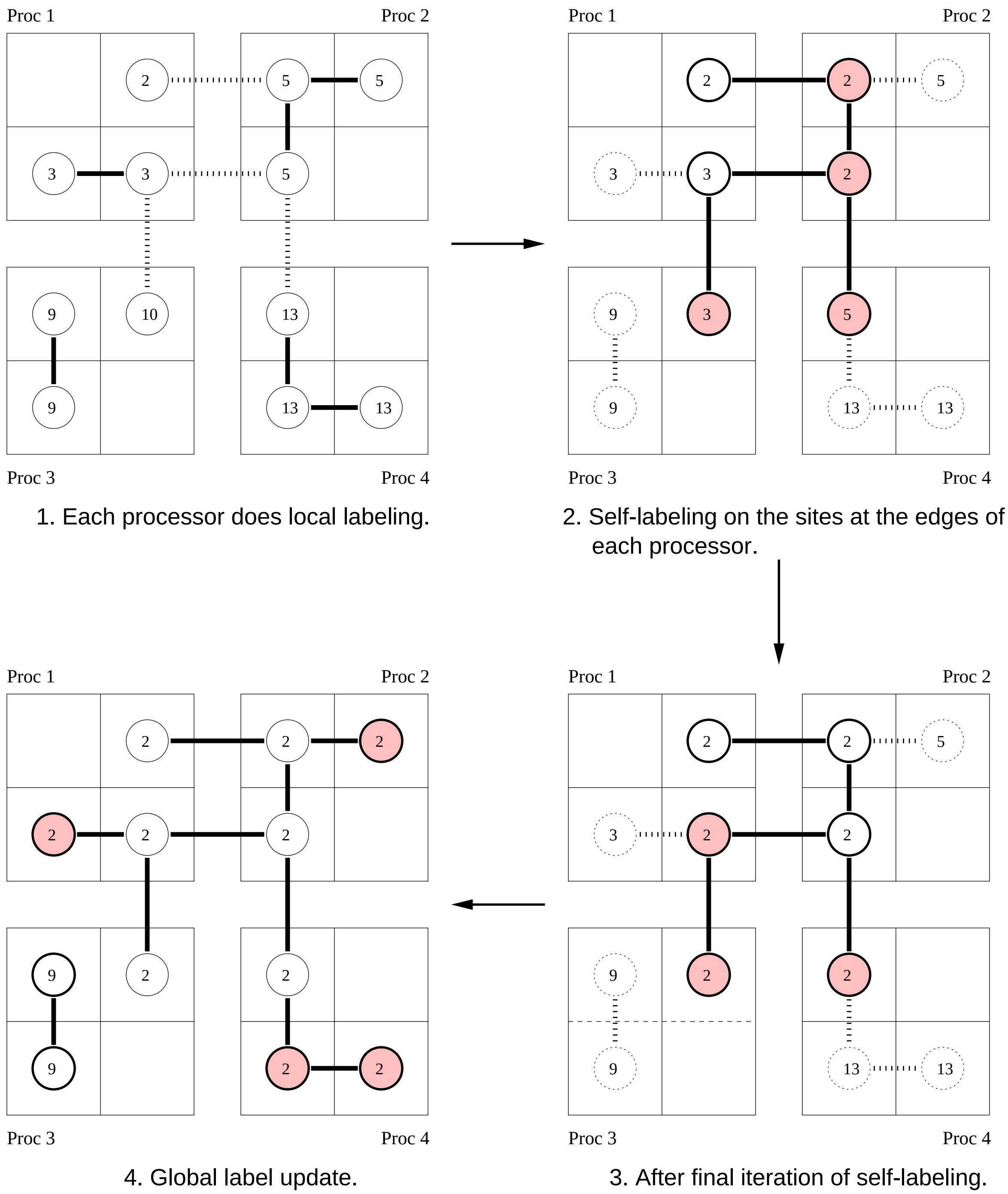


Figure 6.5: MIMD Component Labeling. The bonds are shown as the thick lines.

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

An 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. 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 broadcast to other processors and then, after the processors 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 calculating the quantities locally, sum over sites in each processor, and then combining the partial sums from all processors using `MPI.Allreduce`.

Users can also choose data distribution as row-, column- or block-wise by selecting the nodes in X direction option in GUI. This option is set after the number of running processors is given at the starting time by execution command as follow;

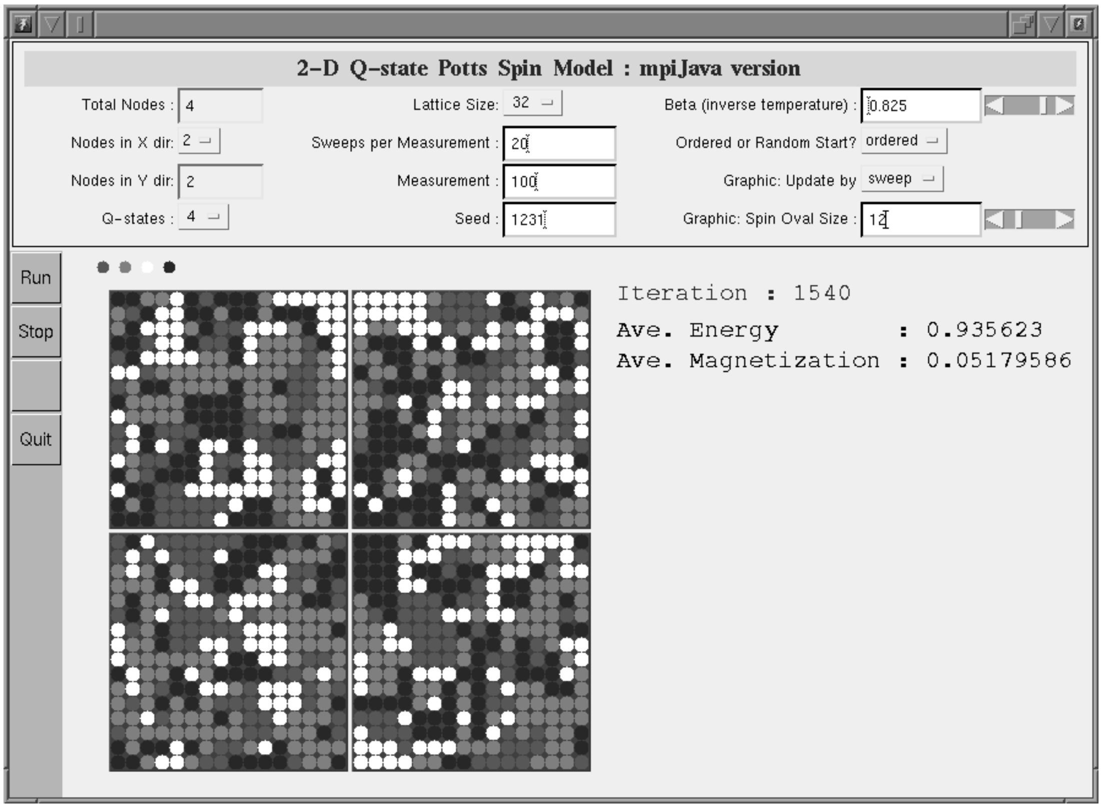


Figure 6.6: The Potts Model Monte Carlo Simulation.

```
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.

6.4 Ising Model Evaluation

In this section we evaluate the performance of some 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 programs 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 [29].

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 [46].

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 the Sun ATM network which is much faster 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 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 codes and the 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 lattice 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. In Metropolis, mpiJava is faster than sequential Java for $L \geq 128$ on $N \geq 2$. On the largest lattice size plotted (1024×1024), mpiJava provides an improvement of up to 83%(Metropolis) and 80% (SW) over sequential Java. 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 or 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 [72]. In addition, mpiJava uses wrapper routines to interface between the Java and the C routines that introduce additional overhead. The sequential and parallel Java to F77 and Java to 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, because the main overhead is in the communications rather than computations. Relatively the mpiJava code perform best when $L = 32$ on 8 nodes. This case the execution times are only 8–14% longer than MPICH F77 or C, but in this case absolute performance is poor because this is the regime where communication overhead dominate. For the largest lattice size $L = 1024$, the results demonstrate that the performance of the mpiJava is within a factor of two to three of MPI F77 or C. Comparing the performance of mpiJava with MPI F77 or C, it is promising, but not acceptable yet. An interesting series of papers from IBM

[76, 77, 105], confirmed that the current generation of Java virtual machines have rather poor performance on Fortran-like, array-intensive computations, but went on to demonstrate how to apply aggressive optimizations in Java compilers to obtain performance competitive with Fortran. In a recent paper [78] they described a case study involving a data mining application that used the Java Array package supported by the Java Grande Numerics Working Group. Using the experimental IBM HPCJ Java compiler they reported obtaining over 90% of the performance of Fortran. Therefore we expect that in the future Java will became quite competive with Fortran.

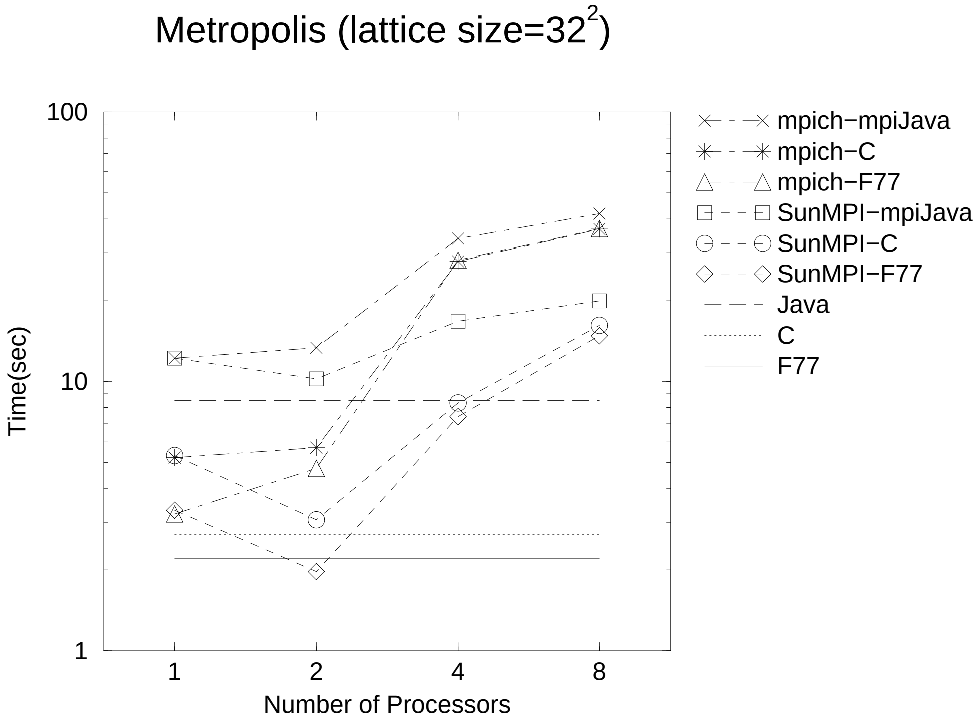


Figure 6.7: Metropolis Performance with Lattice Size 32^2.

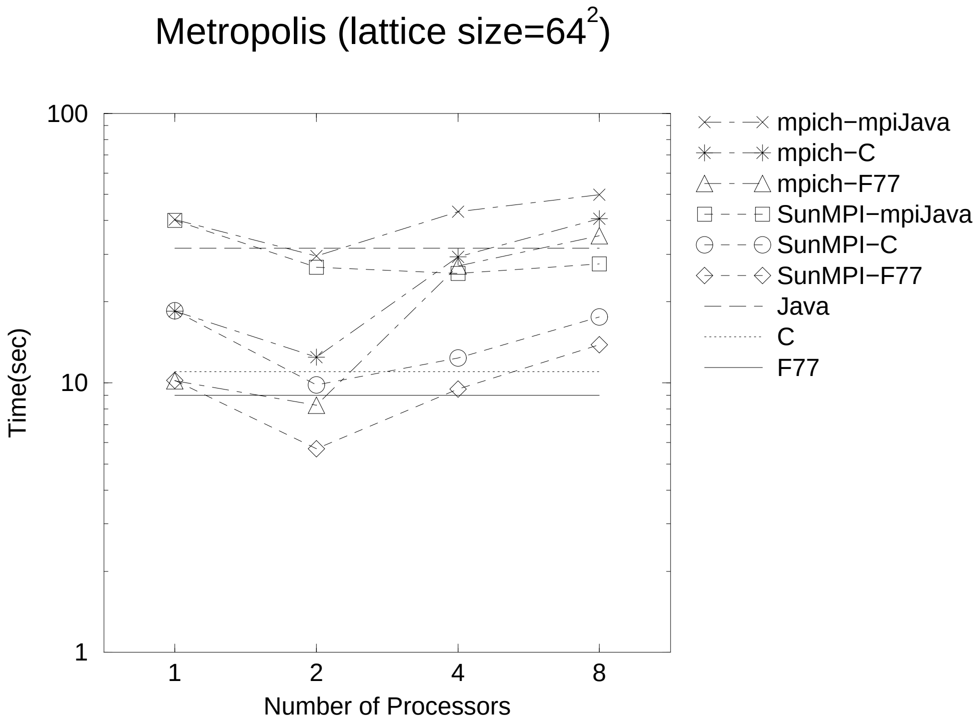


Figure 6.8: Metropolis Performance with Lattice Size 64^2.

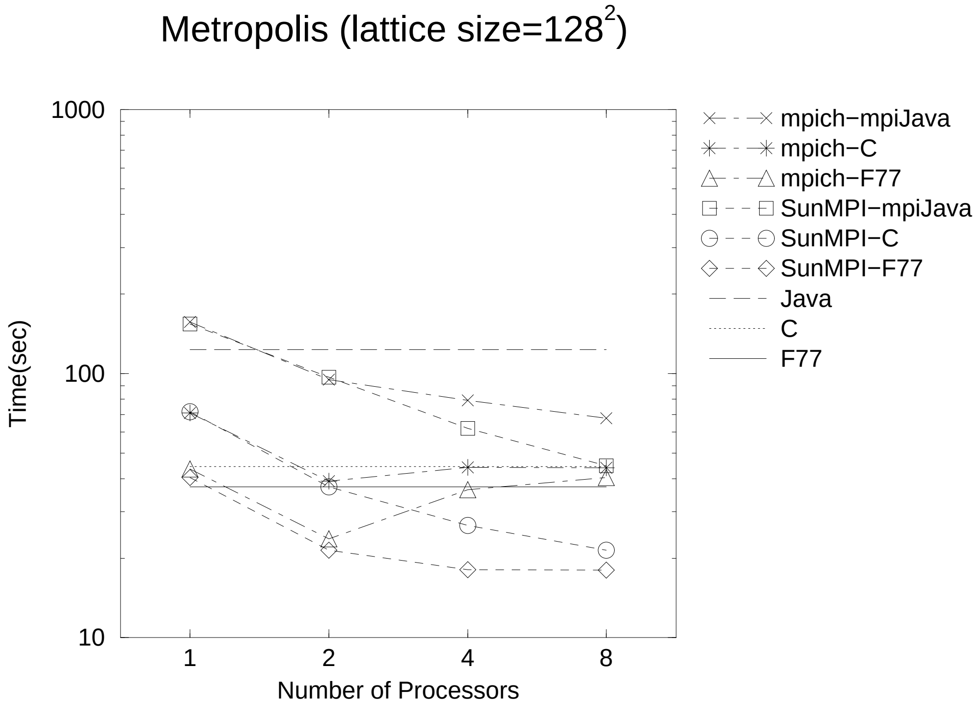


Figure 6.9: Metropolis Performance with Lattice Size 128^2.

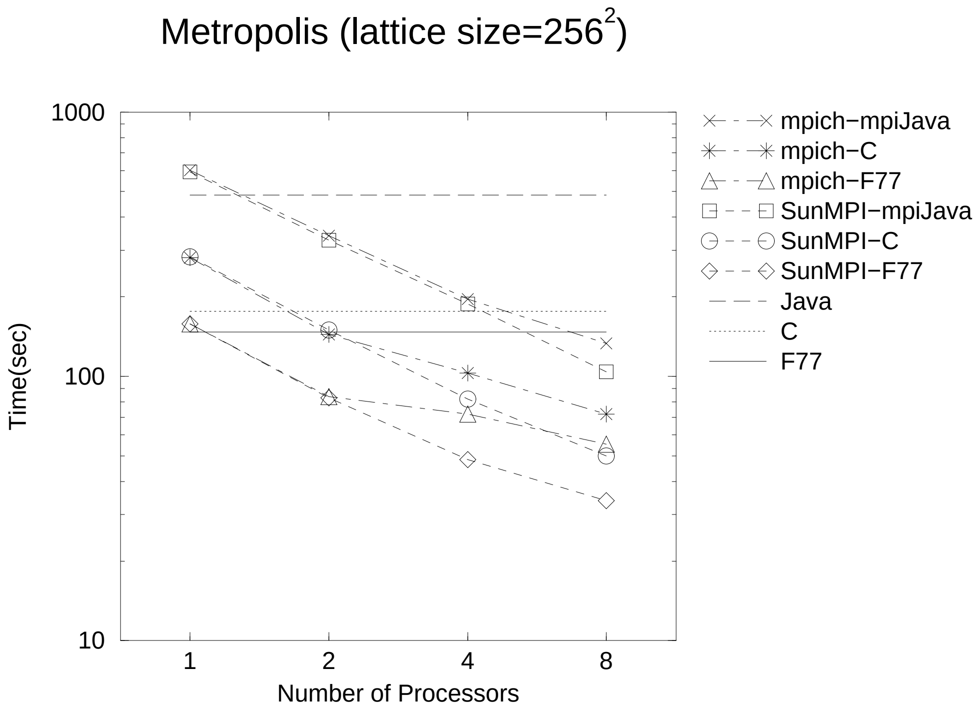


Figure 6.10: Metropolis Performance with Lattice Size 256^2.

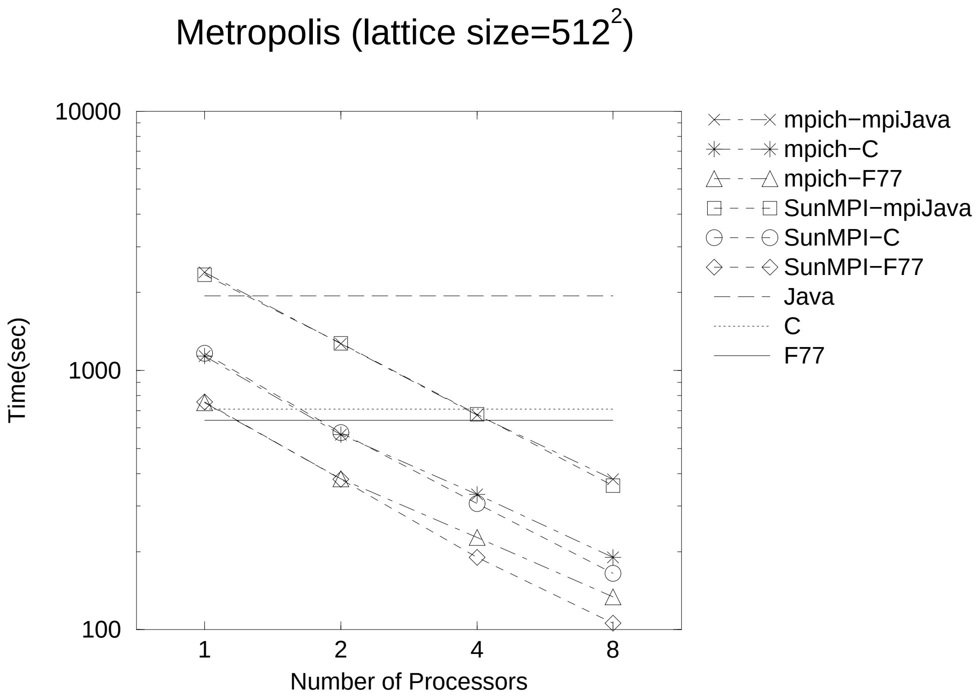


Figure 6.11: Metropolis Performance with Lattice Size 512^2.

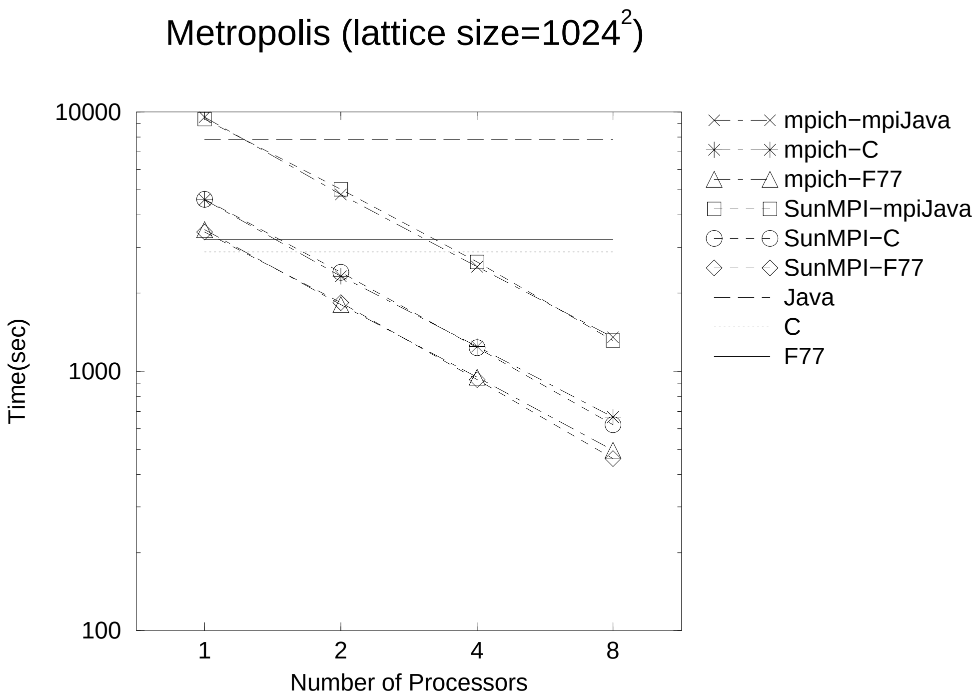


Figure 6.12: Metropolis Performance with Lattice Size 1024^2.

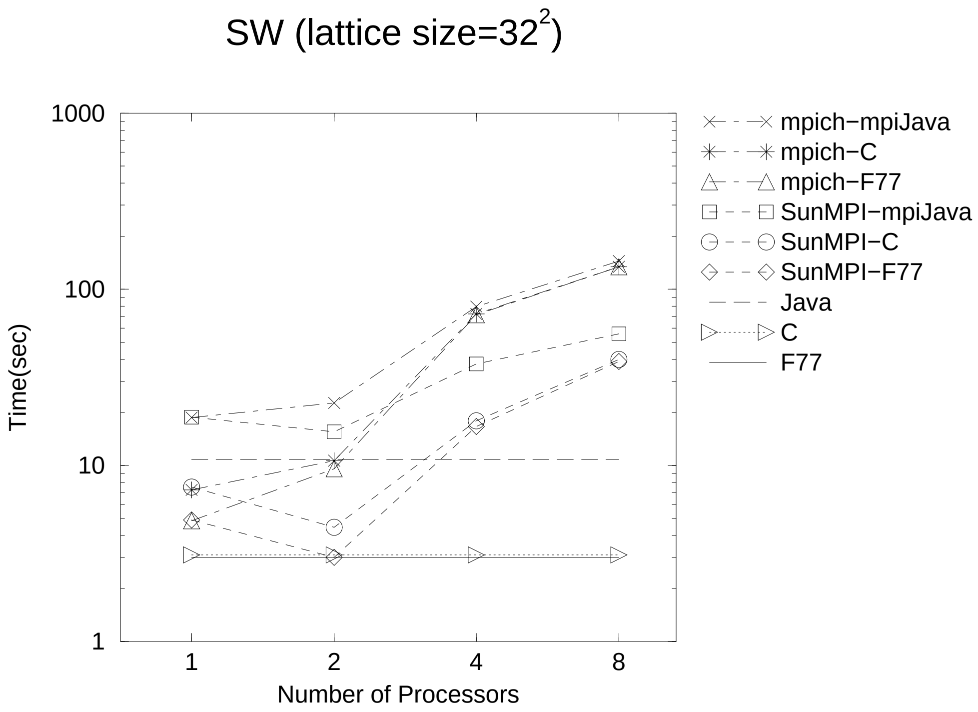


Figure 6.13: Swendsen-Wang Performance with Lattice Size 32^2.

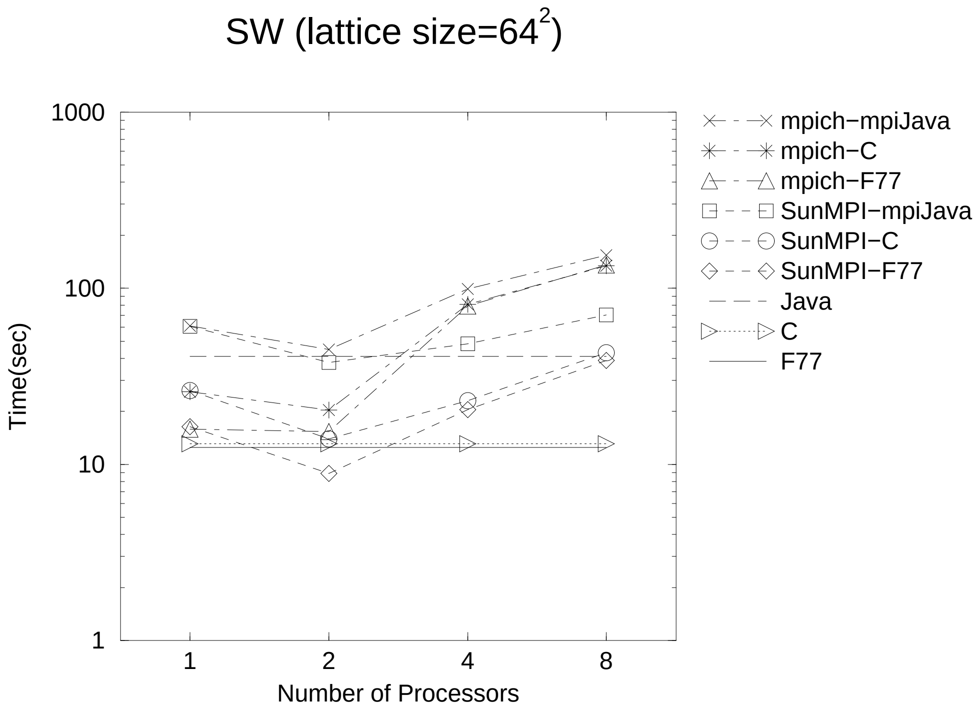


Figure 6.14: Swendsen-Wang Performance with Lattice Size 64^2.

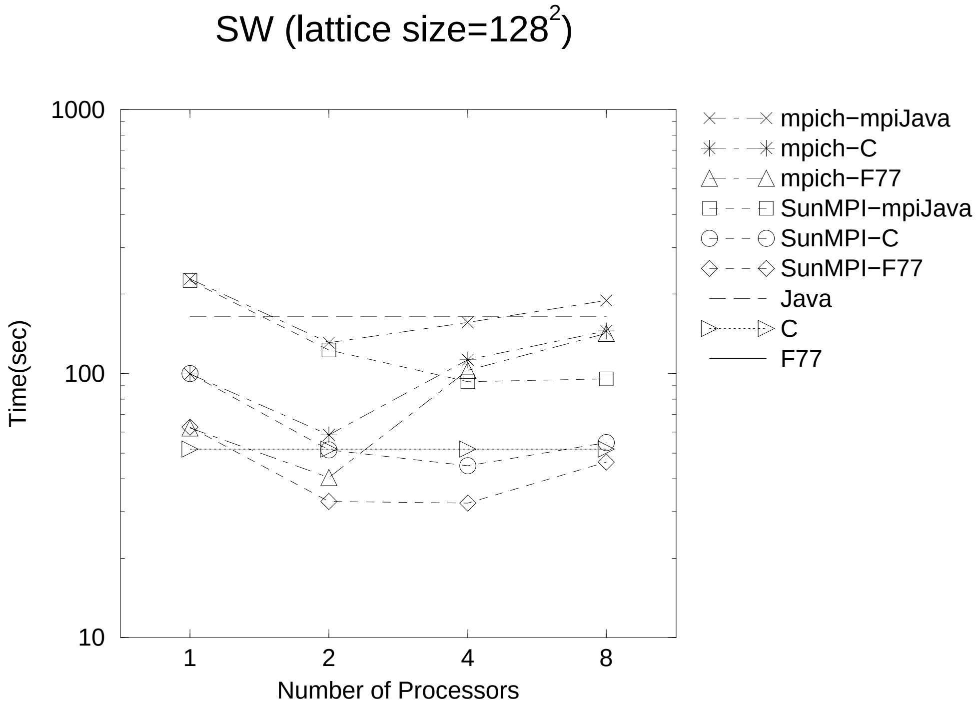


Figure 6.15: Swendsen-Wang Performance with Lattice Size 128^2.

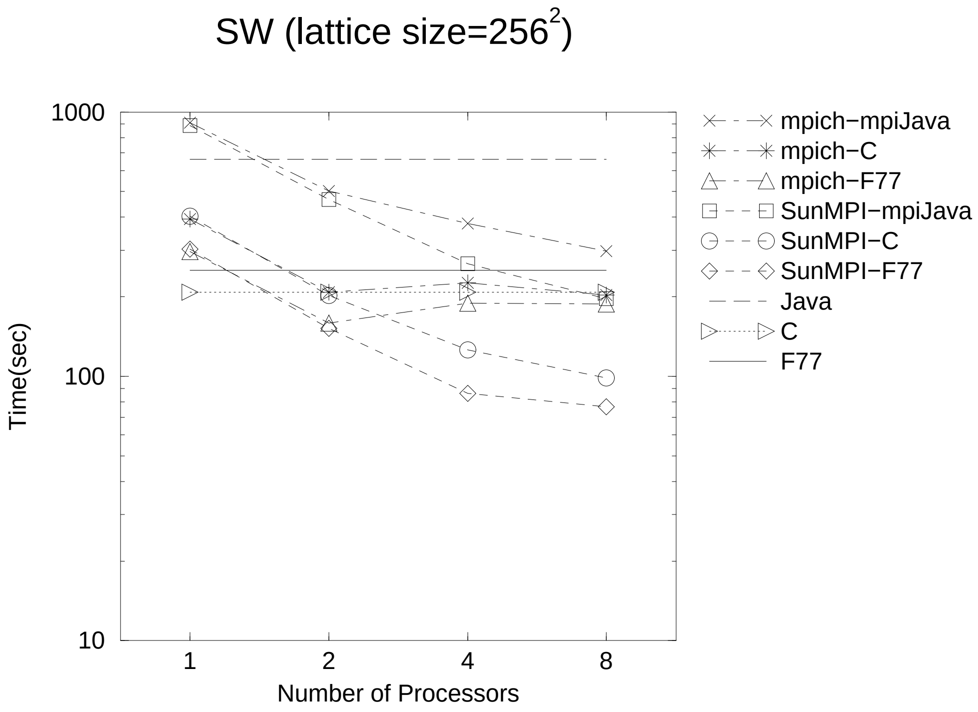


Figure 6.16: Swendsen-Wang Performance with Lattice Size 256^2.

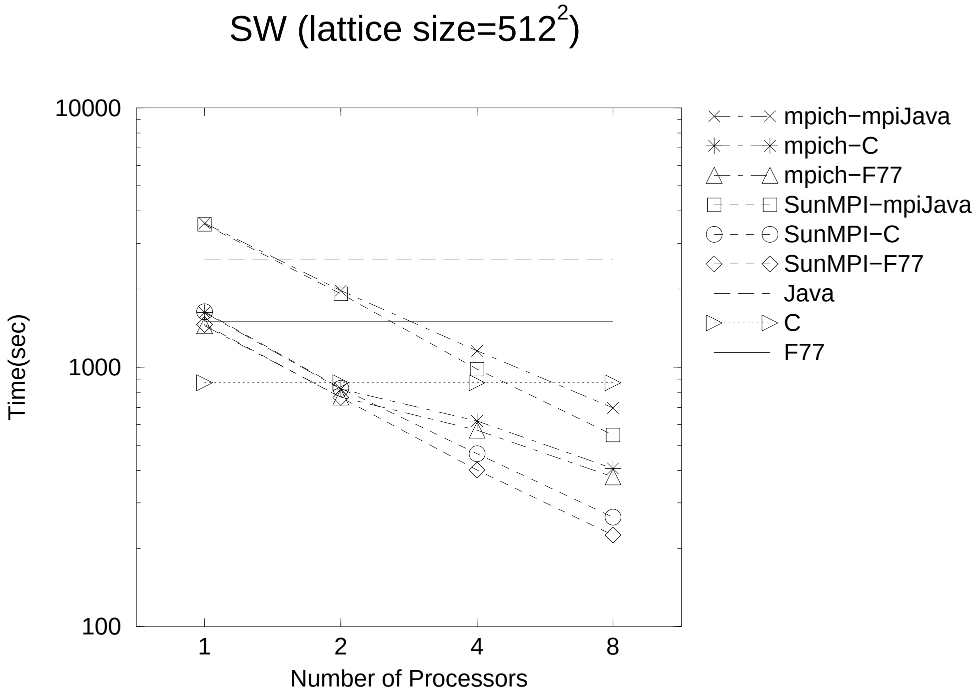


Figure 6.17: Swendsen-Wang Performance with Lattice Size 512^2.

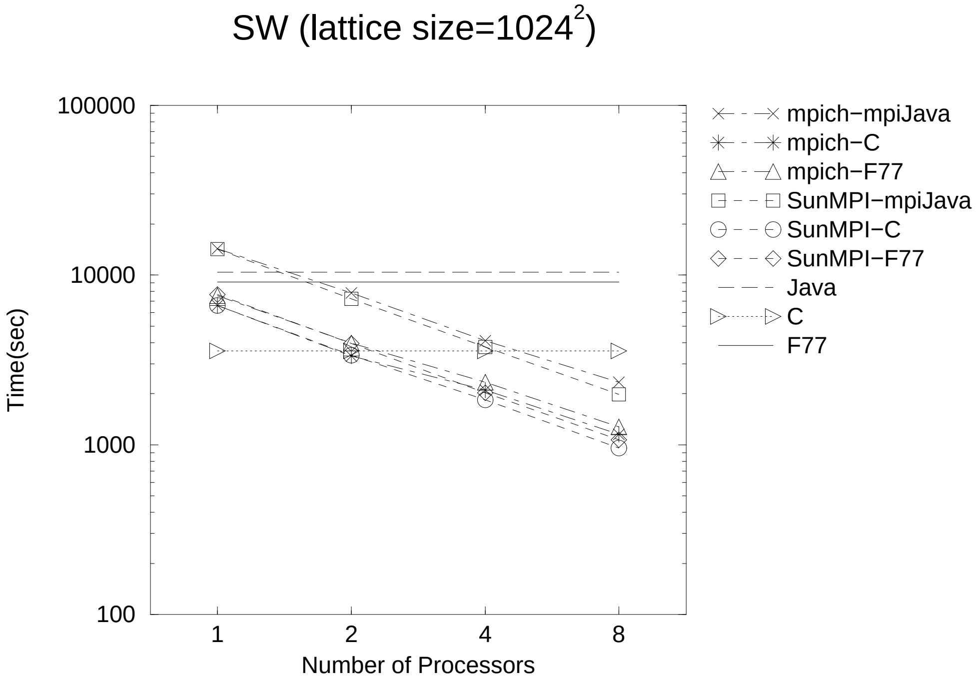


Figure 6.18: Swendsen-Wang Performance with Lattice Size 1024^2.

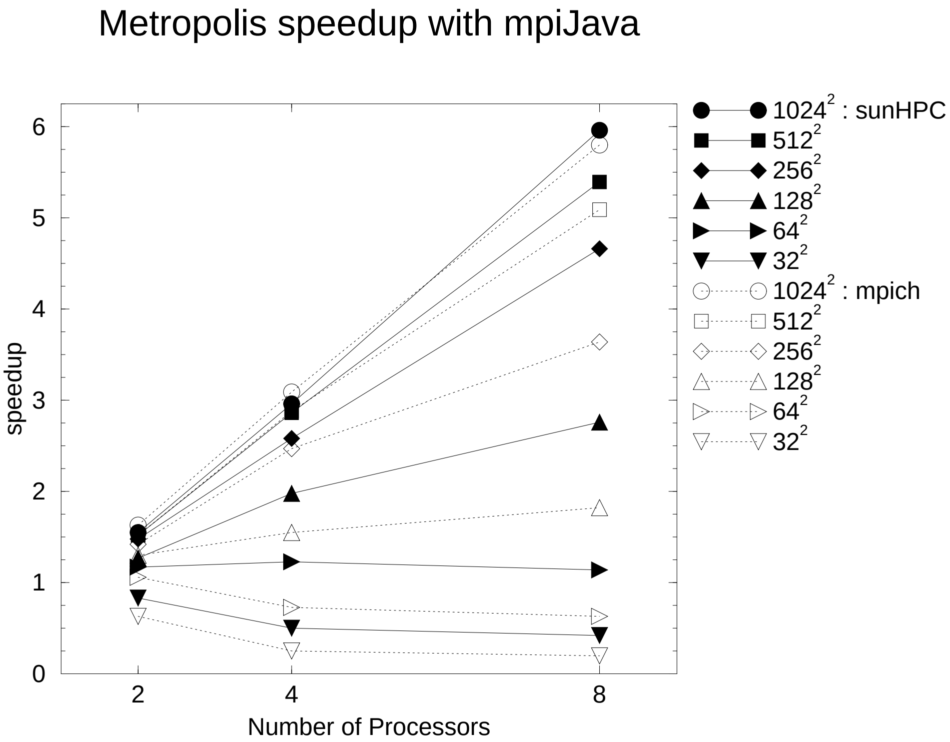


Figure 6.19: Speedup of Metropolis by using mpiJava as compared with serial Java.

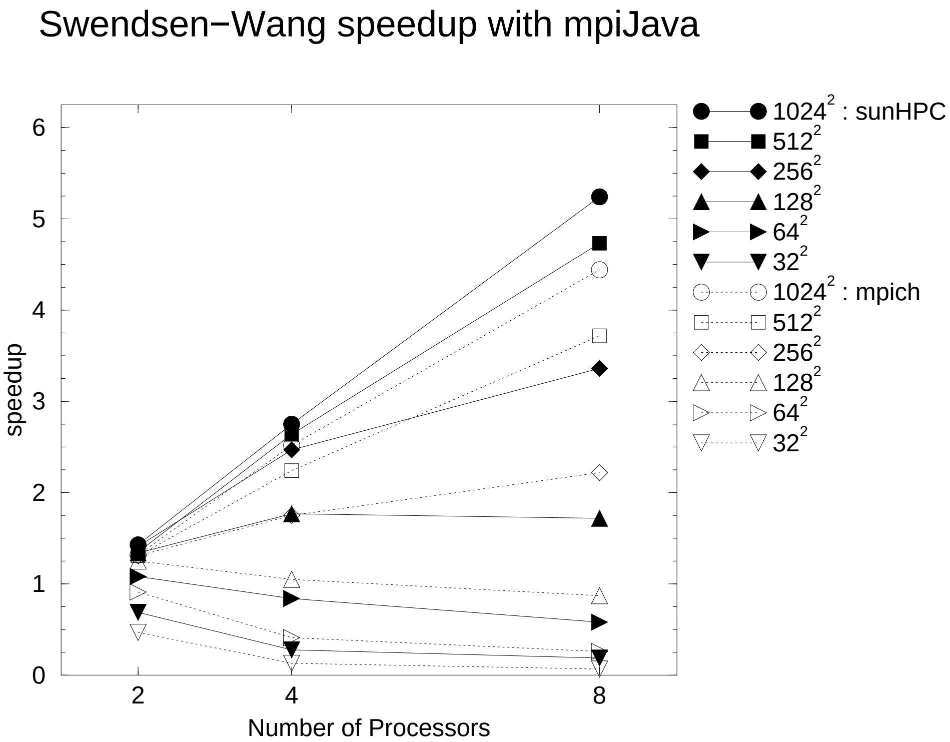


Figure 6.20: Speedup of Swendsen-Wang by using mpiJava as compared with serial Java.

	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 : Comparing Java with F77 and C.

	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: Swendsen-Wang : Comparing Java with F77 and C.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

We have explored the practicality of doing parallel computing in Java, and of providing Java interfaces to High Performance Computing software. 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.

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 3 and 4 involved creating a Java interface to an existing HPC package. 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. For example, IBM is developing an optimizing Java compiler that produces binary code directly.

Our final interface to MPI is quite elegant, and provides much of the functionality of the standard C and Fortran bindings. But creating this interface was a more difficult process than one might hope, both in terms of getting a good specification, and in terms of making the implementation work. We noted that the lack of features like C++ templates (or any form of parametric polymorphism) and user-defined operator overloading (available in many modern languages) made it difficult to produce a completely satisfying interface to a data parallel library. The Java language as currently defined imposes various limits to the creativity of the programmer.

In many respects Java is undoubtedly a better language than Fortran. It is object-oriented 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.

In this dissertation, we have discussed motivations for introducing HPJava, an HPspmd programming model. HPJava language extensions provide much of the expressive power of HPF, but in a strictly SPMD environment with a good communication library. It allows programs to combine data parallel code and SPMD library calls directly. 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. The object-oriented features of Java are also exploited to give an elegant parameterization of the distributed arrays of the extended language.

We have discussed the design and development of mpiJava—a pure Java interface to MPI. mpiJava provides a fully functional and efficient Java interface to MPI. When used for distributed computing the current implementation of mpiJava does not impose a huge overhead on-top of the native MPI interface. 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 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 . 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 2nd version of the mpiJava package has been released and more than 400 people have downloaded it 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://aspen.csit.fsu.edu/pss/HPJava/mpiJava.html`

- We have developed 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 described 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 sequential 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

The work presented in this dissertation can be extended in various directions.

7.3.1 HPJava

In the future version of HPJava, we will complete the HPJava translator. Further research works will include optimization and safety-checking techniques in the compiler for HPJava. At the same time, we plan to integrate further SPMD libraries into the framework. These include more extensive irregular communication (CHAOS-like), one-sided access to array sections (GA-like) and higher-level applications-oriented libraries (DAGH, ScaLAPACK). Implementing a full test suite and evaluating performance of HPJava on large scale applications is also interested.

7.3.2 mpiJava

We plan to continue to improve mpiJava and may add in some of the functionality that has been proposed in MPI-2 [42]. We may also port mpiJava to new MPI environments, including LAM [17] and Globus [44]. We are also planning a *pure* Java MPI environment which does not rely on native MPI services. One of the advantage of pure Java MPI over our mpiJava is its high portability. There are many issues in pure Java implementation. These include starting remote tasks, resource discovery, communication base, and handling partial failures.

7.3.3 The Task and Data-parallel Cluster Algorithms

The major computational task of cluster algorithms is the identification and labeling of the clusters of connected sites, given the configuration of bonds. Due to highly irregular and non-local nature of the cluster, it is very difficult to efficiently implement cluster identification algorithms on SIMD computers [5]. This problem can be solved efficiently by utilizing a mixture of both task and data parallelism in application such as implementing Swendsen-Wang code working in HPF using a call to an extrinsic function to do the component labeling in MPI. However we found that the overhead to switch between HPF and MPI is large, and creating distributed arrays and accessing their local and remote elements is clumsy and errot-prone in HPF. It is expected that component labeling can be implemeted in HPJava efficiently since the HPJava framework has better prospects for dealing effectively with task parallel programming. We plan to develop SPMD style cluster algorithms in HPJava with direct calls to MPI.

In addition, there are many aspects of Java that should be improved. These include Java's object serialization, Remote Method Invocation, and enhancements in the Java Native Interface.

Bibliography

[1] A. Agrawal, A. Sussman, and J. Saltz. An integrated runtime and compile-time approach for parallelizing structured and block structured applications. *IEEE Transactions on Parallel and Distributed Systems*, 6, 1995.

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

[3] 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.

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

[5] Clive F. Baillie and Paul D. Coddington. Cluster Identification Algorithms for Spin Models – Sequential and Parallel. *Concurrency: Practice and Experience*, 3(129), 1991.

[6] Mark Baker. PingPong Benchmarks. http://www.dcs.port.ac.uk/~mab/TOPIC/.

[7] Mark Baker and Bryan Carpenter. Initial thoughts on structure of an MPJ

reference implementation. In *JGMPWG meeting*. NPAC, Syracuse University, October 1999.

[8] 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.).

[9] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object-oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[10] D. E. Bernholdt, E. Aprà, H. A. Früchtl, M. F. Guest, R. J. Harrison, R. A. Kendall, R. A. Kutteh, X. Long, J. B. Nicholas, J. A. Nichols, H. L. Taylor, A. T. Wong, G. I. Fann, R. J. Littlefield, and J. Nieplocha. Parallel computational chemistry made easier: The development of NWChem. *Int. J. Quantum Chemistry: Quantum Chem. Symposium*, 29:475–483, 1995.

[11] 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). http://www.jhpc.org/projects.html, http://www.csam.iit.edu/~pmd/pubs.html.

[12] 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. http://www.extreme.indiana.edu/~ajcbik/JAVAR/.

[13] Blackdown JDK for Linux home page. http://www.blackdown.org/java-linux.html.

[14] 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. http://www.netlib.org/scalapack/scalapack_home.html.

[15] 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.

[16] 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. http://www.extreme.indiana.edu/hpjava/nexusrmi/.

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

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

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

[20] Bryan Carpenter, Yuh-Jye Chang, Geoffrey Fox, Donald Leskiw, and Xiaoming Li. Experiments with HPJava. *Concurrency: Practice and Experience*, 9(6):633, 1997.

[21] Bryan Carpenter, Geoffrey Fox, Donald Leskiw, Xinying Li, Yuhong Wen, and Guansong Zhang. Language bindings for a data-parallel runtime. In

Third International Workshop on High-Level Parallel Programming Models and Supportive Environments. IEEE Computer Society Press, 1998. http://www.npac.syr.edu/projects/pcrc.

[22] 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. http://www.javagrande.org/.

[23] Bryan Carpenter, Guansong Zhang, Geoffrey Fox, Xiaoming Li, Xinying Li, and Yuhong Wen. Towards a Java environment for SPMD programming. In David Pritchard and Jeff Reeve, editors, *4th International Europar Conference*, volume 1470 of *Lecture Notes in Computer Science*. Springer, 1998. http://www.npac.syr.edu/projects/pcrc/HPJava.

[24] 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.

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

[26] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):1–50, 1992.

[27] 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. http://www.cs.ucsb.edu/research/javelin/.

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

[29] 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.

[30] 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.

[31] 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. http://www.baskent.edu.tr/~kdincer/.

[32] 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.

[33] 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.

[34] G. Fox et al. Fortran D language specification. Technical Report CRPC-TR90079, Center for Research on Parallel Computation, Rice University, 1990.

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

[36] Adam J. Ferrari. JPVM: Network Parallel Computing in Java. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, Palo Alto, February 1998. http://www.cs.virginia.edu/~ajf2j/jpvm.html.

[37] Stephen J. Fink and Scott B. Baden. Run-time data distribution for block-structured applications on distributed memory computers. In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

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

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

[40] High Performance Fortran Forum. High Performance Fortran language specification, version 2.0, January 1997. http://www.crpc.rice.edu/HPFF/hpf2.

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

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

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

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

[45] 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. http://www.globus.org/nexus/nexusjava.html.

[46] 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.

[47] 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.
http://www.npac.syr.edu/users/gcf/petastuff/petaweb/.

[48] 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.

[49] Michael Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, 2(3):171–193, 1990.

[50] 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.

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

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

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

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

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

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

[57] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, special issue, 2, 1993.

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

[59] IBM JDK for Linux home page. http://www.alphaworks.ibm.com/tech/linuxjvm,.

[60] 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.

[61] Java Grande Forum home page. http://www.javagrande.org.

[62] JavaPVM home page. http://www.isye.gatech.edu/chmsr/JavaPVM/.

[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. http://ccc.cs.byu.edu/DOGMA/.

[64] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceeding of the 1994 Winter Usenix Conference Systems*, pages 115–131, January 1994.

[65] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, 1991.

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

[67] Vijaykumar Krishnaswamy, Dan Walther, Sumeer Bhola, Ethendranath Bommaiah, George Riley, and Brad Topol. Efficient implementation of Java Remote Method Invocatio(RMI). In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and System(COOTS'98)*, 1998.

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

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

[70] 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. http://www.cs.vu.nl/manta/.

[71] Linux Mandrake home page. http://www.linux-mandrake.com.

[72] 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. http://www.cs.ucsb.edu/conferences/java99.

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

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

[75] 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.

[76] J. E. Moreira. Closing the performance gap between Java and Fortran in technical computing. In *First UK Workshop on Java for High Performance Network Computing*, September 1998. http://www.cs.cf.ac.uk/hpjworkshop/.

[77] J. E. Moreira, S. P. Midkiff, and M. Gupta. From flop to MegaFlops: Java for technical computing. In *Languages and Compilers for Parallel Computing*, volume 1656 of *Lecture Notes in Computer Science*. Springer, 1998.

[78] J. E. Moreira, S. P. Midkiff, M. Gupta, and R. Lawrence. High Performance Computing with the Array Package for Java: A Case Study using Data Mining. In *Supercomputing 99*, November 1999.

[79] MPIJ home page. http://ccc.cs.byu.edu/OnlineDocs/docs/mpij/MPIJ.html.

[80] 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.

[81] 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. http://www.emsl.pnl.gov:2080/docs/global/.

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

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

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

[85] 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 Mathematics and its Applications. Springer-Verlag.

[86] Michael Philippsen and Bernhard Haumacher. Bandwidth, Latency, and other Problems of RMI and Serialization. In *Second Java Grande Forum Meeting*, Palo Alto, May 1998. http://www.javagrande.org/meetings.htm.

[87] 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.

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

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

[90] 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.

[91] Redhat Linux home page. http://www.redhat.com/.

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

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

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

[95] 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. http://www.jhpc.org/projects.html.

[96] 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. http://www.cs.vu.nl/manta/.

[97] 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. http://www.pds.twi.tudelf.nl/~reeuwijk/projects/javaws_poster.html.

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

[99] Matt Welsh. NinjaRMI home page. http://www.cs.berkeley.edu/~mdw/proj/ninja/ninjarmi.html.

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

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

[102] Wmpi home page. http://dsg.dei.uc.pt/w32mpi/.

[103] Ulli Wolff. Collective Monte Carlo Updating for Spin Systems. *Physical Review Letters*, 62(4):361–364, January 1989.

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

[105] Oeng Wu, Sam Midkiff, Jose Moreira, and Manish Gupta. Efficient support for complex numbers in Java. In *ACM 1999 Java Grande Conference*. ACM Press, June 1999.

[106] 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.

[107] 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. http://www.cs.berkeley.edu/Research/Projects/titanium/.

[108] Weimin Yu and Alan Cox. Java/DSM: A Platform for Heterogeneous Computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, November 1997.

[109] Guansong Zhang, Bryan Carpenter, Geoffrey Fox, Xiaoming Li, Xinying Li, and Yuhong Wen. PCRC-based HPF compilation. In Zhiyuan Li et al, editor, *10th International Workshop on Languages and Compilers for Parallel Computing*, volume 1366 of *Lecture Notes in Computer Science*. Springer, 1997. http://www.npac.syr.edu/projects/pcrc.

Vita

NAME OF AUTHOR: SUNG-HOON KO

This dissertation was typeset with $\LaTeX\,2_\varepsilon$[1] by the author.

[1] $\LaTeX\,2_\varepsilon$ 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.

Publications

- Bryan Carpenter, Geoffrey Fox, Sung-Hoon Ko and Sang Lim. Object Serialization for Marshalling Data in a Java Interface to MPI. *ACM 1999 Java Grande Conference*, June 1999.

- Bryan Carpenter, Geoffrey Fox, Sung-Hoon Ko and Sang Lim. Automatic Object Serialization in the mpiJava Interface to MPI, *Third MPI Developer's and User's Conference, MPIDC '99*, March 1999.

- Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung-Hoon Ko, and Sang Lim. mpiJava: An Object-Oriented Java interface to MPI. *International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP 1999*, San Juan, Puerto Rico, April 1999.

- Mark Baker, Bryan Carpenter, Sung-Hoon Ko, and Xinying Li. mpiJava: A Java interface to MPI. *First UK Workshop on Java for High Performance Network Computing, Europar '98*, 1998.

- Jang Sun Lee, Sung-Hoon Ko, Sanjay Ranka, and Byung Eui Min. High-Performance External Computations Using User-Controllable I/O. *2th International Parallel Processing Symposium & 9th Symposium on Parallel and Distributed Processin*, 1998

- Paul D. Coddington and Sung-Hoon Ko, Techniques for Empirical Testing of

Parallel Random Number Generators, *Proc. of the 12th ACM Int. Conf. on Supercomputing*, Melbourne, July 1998

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