THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

PLATFORMS FOR HPJAVA: RUNTIME SUPPORT FOR SCALABLE PROGRAMMING IN JAVA

By

SANG BOEM LIM

A Dissertation submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Summer Semester, 2003

The members of the Committee approve the dissertation of Sang Boem Lim defended on June 11, 2003.

Gordon Erlebacher
Professor Directing Dissertation

Geoffrey C. Fox
Professor Co-Directing Dissertation

Larry Dennis
Outside Committee Member

Daniel G. Schwartz
Committee Member

Robert A. van Engelen
Committee Member

The Office of Graduate Studies has verified and approved the above named committee members.

ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation Division of Advanced Computational Infrastructure and Research, contract number 9872125 (any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation).

This work was additionally supported by the FSU School for Computational Science and Information Technology (CSIT), and utilized the CSIT IBM SP3 system.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The dissertation research is concerned with enabling parallel, high-performance computation—in particular development of scientific software in the network-aware programming language, Java. Traditionally, this kind of computing was done in Fortran. Arguably, Fortran is becoming a marginalized language, with limited economic incentive for vendors to produce modern development environments, optimizing compilers for new hardware, or other kinds of associated software expected of by today's programmers. Hence, Java looks like a very promising alternative for the future.

The dissertation will discuss in detail a particular environment called *HPJava*. HPJava is the environment for parallel programming—especially data-parallel scientific programming– in Java. Our HPJava is based around a small set of language extensions designed to support parallel computation with distributed arrays, plus a set of communication libraries. In particular the dissertation work will concentrate on issues related to the development of efficient run time support software for parallel languages extending an underlying object-oriented language.

Two characteristic run-time communication libraries of HPJava are developed as an application level library and device level library. A high-level communication API, *Adlib*, is developed as an application level communication library suitable for our HPJava. This communication library supports *collective operations* on distributed arrays. We include Java `Object` as one of the Adlib communication data types. So we fully support communication of intrinsic Java types, including primitive types, and Java object types. The Adlib library is developed on top of low-level communication library called *mpjdev*, designed to interface efficiently to the Java execution environment (virtual machine).

The mpjdev API is a device level underlying communication library for HPJava. This library is developed to perform actual communication between processes. The mpjdev API is developed with HPJava in mind, but it is a standalone library and could be used by other

systems. This can be implementing portably on network platforms and efficiently on parallel hardware.

The dissertation describes the novel issues in the interface and implementation of these libraries on different platforms, and gives comprehensive benchmark results on a parallel platform. All software developed in this project is available for free download from `www.hpjava.org`.

CHAPTER 1

INTRODUCTION

The Java programming language is becoming the language of choice for implementing Internet-based applications. Undoubtedly Java provides many benefits—including access to secure, platform-independent applications from anywhere on the Internet. Java today goes well beyond its original role of enhancing the functionality of HTML documents. Few Java developers today are concerned with applets. Instead it is used to develop large-scale enterprise applications, to enhance the functionality of World Wide Web servers, to provide applications for consumer device such as cell phones, pagers and personal digital assistants.

Amongst *computational* scientists Java may well become a very attractive language to create new programming environments that combine powerful object-oriented technology with potentially high performance computing. The popularity of Java has led to it being seriously considered as a good language to develop scientific and engineering applications, and in particular for parallel computing [2, 3, 4]. Sun's claims on behalf of Java, that is simple, efficient and platform-natural—a natural language for network programming—make it attractive to scientific programmers who wish to harness the collective computational power of parallel platforms as well as networks of workstations or PCs, with interconnections ranging from LANs to the Internet. This role for Java is being encouraged by bodies like Java Grande [33].

Over the last few years supporters of the Java Grande Forum have been working actively to address some of the issues involved in using Java for technical computation. The goal of the forum is to develop consensus and recommendations on possible enhancements to the Java language and associated Java standards, for large-scale ("Grande") applications. Through a series of ACM-supported workshops and conferences the forum has helped stimulate research on Java compilers and programming environments.

```
float [[*, *]] c = new float [[M, N]];
float [[*, *]] a = new float [[M, L]];
float [[*, *]] b = new float [[L, N]];

... initialize 'a', 'b'

for(int i = 0; i < M; i++)
  for(int j = 0; j < N; j++) {

    c [i, j] = 0;
    for(int k = 0; k < L; k++)
      c [i, j] += a [i, k] + b [k, j];
  }
```

Figure 1.1. Sequential Matrix multiplication in HPJava.

Our HPJava is an environment for parallel programming, especially suitable for data parallel scientific programming. HPJava is an implementation of a programming model we call the *HPspmd nodel*. It is a strict extension of its base language, Java, adding some predefined classes and some extra syntax for dealing with distributed arrays. We overview of HPJava in following section.

1.1 Features of HPJava

In this section, we present a high level overview of our HPJava. Some predefined classes and some extra syntax for dealing with distributed arrays are added into the basic language, Java. We will briefly review the features of the HPJava language by showing simple HPJava examples. In this section, we will only give an overview of features. Detailed description of those features will be presented in the Chapter 3.

Figure 1.1 is a basic HPJava program for sequential matrix multiplication. This program is simple and similar to the ordinary Java program. It uses simple sequential *multiarrays*—a feature HPJava adds to standard Java. A multiarray uses double brackets to distinguish the type signature from a standard Java array. The multiarray and ordinary Java array have

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

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

  // ... initialize 'a', 'b'

  overall(i = x for : )
    overall(j = y for : ){

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

      c[i, j] = sum;

    }
}
```

Figure 1.2. A parallel Matrix multiplication in HPJava.

many similarities. Both arrays have some index space and stores a collection of elements of fixed type. Syntax of accessing a multiarray is very similar with accessing ordinary Java array which uses single brackets, but an HPJava sequential multiarray uses double bracket and asterisks for its type signature. The most significant difference between ordinary Java array and the multiarray of HPJava is that the distributed array is *true multi-dimensional array* like the arrays of Fortran, while ordinary Java only provides arrays of arrays. These features of Fortran arrays have adapted and evolved to support scientific and parallel algorithms.

HPJava also adds a *distributed array* feature to the base language Java. The distributed array is a very important feature of HPJava. Like the name says, the elements of the distributed array are distributed among processes: each process only has a portion of the data. As we can see from the parallel version (Figure 1.2) of matrix multiplication, HPJava

puts a hyphen in the type signature to indicate a particular dimension is distributed over processes. Input arrays `a` and `b` and result array `c` are *distributed arrays*. The array `c` is distributed in both its dimensions while array `a` and `b` each have one distributed dimension and one sequential dimension.

Figure 1.2 is a simple program but it includes much of the HPJava special syntax. In this program, we can see some unusual key words and operations which we do not see in an ordinary Java program like control constructs *on* and *overall*. There are also some added standard classes, like `Procs2`, and `BlockRange`.

The class `Procs2` represents 2-dimensional grids of processes selected from the set of available processes. The `on` construct makes `p` the *active process group* within its body, and only the processes that belong to `p` execute the code inside of the `on` construct. Distribution format of each dimension of a distributed array is represented by a `Range` class. The `BlockRange` class used in this example describes block-distributed indexes. We also have a control construct which represents distributed parallel loops, similar the *forall* construct in High Performance Fortran (HPF), called `overall`. The `overall` construct introduced a *distributed index* for subscripting a distributed array.

The program in Figure 1.2 depends on a special *alignment relation* between its distributed arrays.

We can create a general purpose matrix multiplication routine that works for arrays with any distributed format (Figure 1.3) from Figure 1.2 using collective communication. This program take arrays which may be distributed in both their dimensions, and copies into the temporary array with a special distribution format for better performance. A collective communication schedule `remap()` is used to copy the elements of one distributed array to another.

From the viewpoint of this dissertation, the most important part of this code is communication method. We can divide the communication library of HPJava into two parts: the high-level *Adlib* library, and the low-level *mpjdev* library. The Adlib library is responsible for the collective communication schedules and the mpjdev library is responsible for the actual communication. One of the most characteristic and important communication library methods, `remap()`, takes two arrays as arguments and copies the elements of the source array to the destination array, regardless of the distribution format of the two arrays.

```
public void matmul(float [[-,-]] c, float [[-,-]] a, float [[-,-]] b) {

  Group2 p = c.grp();

  Range x = c.rng(0);
  Range y = c.rng(1);

  int N = a.rng(1).size();

  float [[-,*]] ta = new float [[x, N]] on p;
  float [[*,-]] tb = new float [[N, y]] on p;

  Adlib.remap(ta, a);
  Adlib.remap(tb, b);

  on(p)
    overall(i = x for : )
      overall(j = y for : ) {

        float sum = 0;
        for(int k = 0; k < N ; k++)
          sum += ta [i, k] * tb [k, j];

        c[i, j] = sum;
      }
}
```

Figure 1.3. A general Matrix multiplication in HPJava.

In addition to the features described above, many more features and communication functions available to the HPJava are described in the Chapter 3 and Chapter 4.

The HPJava software including Adlib and mpjdev is available for free download from `www.hpjava.org`.

1.2 Organization of the Dissertation

This dissertation discusses in detail a particular environment under development at Florida State University and Indiana University called HPJava. The idea behind HPJava is

not only to use Java for scientific computing. It is also to demonstrate a specific programming model for parallel computing that we call the *HPspmd* model. The HPspmd model itself is independent of the choice of Java as base language, but the two go together well. One goal of this dissertation is as a foundation for research and development on the HPJava environment. In particular the dissertation concentrates on issues related to the development of efficient *run time support software* for a class of parallel languages extending an underlying object-oriented language. So the dissertation emphasizes aspects most relevant to this topic—aspects like high performance interprocessor communication.

A review of interesting related research and systems is given first. We will be especially interested in technologies developed in recent years by participants of bodies like the Java Grande Forum and the Global Grid Forum. We start with reviews of general parallel and distributed computing environments for Java including JavaParty, Javelin, and Jini. We discuss communication approaches relevant to parallel computing including Java RMI and Message passing libraries for Java. As an example of distributed computing, we review Peer-to-Peer (P2P) computing and Java enterprise computing. We review SETI@home as a characteristic example of P2P. As examples of P2P in Java, we briefly review Parabon, and discuss JXTA in more detail. We also briefly review contemporary, Java-specific enterprise distributed computing technology, like the Java Message Service and Enterprise JavaBeans. Java as language support for parallel and scientific computing is compared with C++ and C#.

The dissertation continues with the HPJava design and its features. Motivation of HPJava and its syntax features are described in this chapter. An existing runtime library, mpiJava, is also discussed. The mpiJava library, an object-oriented Java interface to MPI, has been developed as part of HPJava. Its usage in HPJava will be described.

Discussion of detailed detailed implementation issues for our high-level communication library follows. New syntax features of HPJava omitted from previous chapters are also illustrated in this chapter. Implementation of a characteristic collective communication function is described in depth. Complete API of Adlib library methods are presented in an appendix. In this appendix we describes complete API, effect, pre-requirements, and restrictions of each method in the Adlib library.

Detailed description of a low-level communication library, mpjdev, is follows. We start with requirements of mpjdev. Its Message buffer API and communication API are described. We describe how a message will be formatted, sent, and received among the processes. Four different implementations of mpjdev are also described.

The following chapter presents applications and analyzes performance of HPJava programs. A full application of HPJava is discussed first to illustrate how HPJava can be used as a tool for complex problems. Some partial differential equations are developed and analyzed for the performance using HPJava. We describe computational fluid dynamics simulation into a grapical user interface using HPJava. We also have performance test and evaluation of a platform-specific, LAPI-based implementation of the libraries.

The final chapter, the conclusion and the directions for future work are discussed.

CHAPTER 2

ENVIRONMENTS FOR PARALLEL AND DISTRIBUTED COMPUTING

In this chapter we give a general overview of some interesting research and systems related to ours. We will be especially interested in technologies developed in recent years by participants of bodies like the Java Grande Forum and the Global Grid Forum. We start with discussion of parallel and distributed computing environments for Java. We briefly review contemporary Java-specific distributed computing technology and how they related to our high performance infrastructures. Finally Java as language to support parallel and scientific computing is compared with C++ and C#.

2.1 Parallel and Distributed Computing Environments for Java

As well as reviewing select computing environments for Java, this section includes discussion of communication frameworks including Java RMI and Message Passing libraries for Java. PVM and MPJ are discussed as examples of message passing libraries. MPJ is an effort by members of the Java Grande Forum [33] to define a consensus MPI-like interface for Java.

2.1.1 Parallel Virtual Machine (PVM) in Java

Communication in parallel programming significantly affects the efficiency of distributed systems. We need to find a way to move data from one local memory to another because a distributed memory system does not share one common global memory. Message passing is very suitable to this task via send/receive API calls which must be written into the application program or used by some higher-level software. To achieve increased bandwidth,

reduced latency, and better reliability within workstation clusters using Java, several projects are under way.

PVM [30], developed at Oak Ridge National Laboratory, provides a mechanism that enables a collection of heterogeneous networked computing systems to be used cooperatively for concurrent or parallel computation. This system can solve large computational problems effectively using the aggregate power and memory of many computers. The PVM system consists of two parts: a daemon, pvmd, and a library of PVM. Any user with a valid login can install a daemon on a computer, adding it to a virtual machine. The PVM library contains a repertoire of primitives that are needed for cooperation between tasks of an application: user-callable routines that include message passing, spawning processes, coordinating tasks, and modifying the virtual machine. C, C++, and Fortran languages are currently supported by the PVM system.

JavaPVM [34] (or jPVM), which is an interface written using the Java native methods capability, and *JPVM* [26], which is a pure Java implementation of PVM are two groups working on making PVM support for programs written in Java. Since JavaPVM is using native methods, cross-platform portability is limited. JPVM is better matched to Java programming styles, much simpler to maintain across heterogeneous machines compared to JavaPVM. The performance of JavaPVM is better compared to JPVM. Communication benchmarks of native PVM, JavaPVM, and JPVM are reported in [57].

2.1.2 MPI and MPJ

The Message-Passing Interface (MPI) [39] is the first message-passing standard for programming parallel processors. It provides a rich set of communication libraries, application topologies and user defined data types. Language-independent specification and language-specific (C and Fortran) bindings are provided in the MPI standard documents. The MPI-2 release of the standard added a C++ binding [29]. A binding of MPI for Java has not been offered and is not planned by the MPI Forum. With the evident success of Java as a programming language, and its inevitable use in connection with parallel as well as distributed computing, the absence of a well-designed language-specific binding of message-passing with Java will lead to divergent, non-portable practices. A likely prerequisite for parallel programming in a distributed environment is a good message passing API.

In 1998, the Message-Passing Working Group of Java Grande Forum was formed in response to the independent development by several groups of Java APIs for MPI-like systems. Discussion of a common API for MPI-like Java libraries was an immediate goal. An initial draft for a common API specification was distributed at Supercomputing '98 [14]. Minutes of meetings that held in San Francisco, Syracuse and Supercomputing '99 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*. Its reference implementation can be implemented two ways. A *pure Java* reference implementation of that API has been proposed, and Java wrapper to a native MPI reference implementation, which will be discussed in section 3.4.1. We will discuss a proposed pure Java reference implementation in section 5.5.4.

2.1.3 Java RMI

Java Remote Method Invocation (RMI), which is a simple and powerful network object transport mechanism, provides a way for a Java program on one machine to communicate with objects residing in different address spaces. Some Java parallel computing environments use RMI for communication, such as JavaParty, discussed in next section. It is also the foundation of Jini technology—discussed on section 2.1.5. RMI is an implementation of the distributed object programming model, comparable with CORBA, but simpler, and specialized to the Java language. An overview of the RMI architecture is shown in Figure 2.1. Important parts of the RMI architecture are the stub class, the object serialization, and the server-side Run-time System.

The stub class implements the remote interface and is responsible for marshaling and unmarshaling the data and managing the network connection to a server. An instance of the stub class is needed on each client. Local method invocations on the stub class will be made whenever a client invokes a method on a remote object.

Java has a general mechanism for converting objects into streams of bytes that can later be read back into an arbitrary JVM. This mechanism, called object serialization, is an essential functionality needed by Java's RMI implementation. It provides a standardized way to encode all the information into a byte stream suitable for streaming to some type of

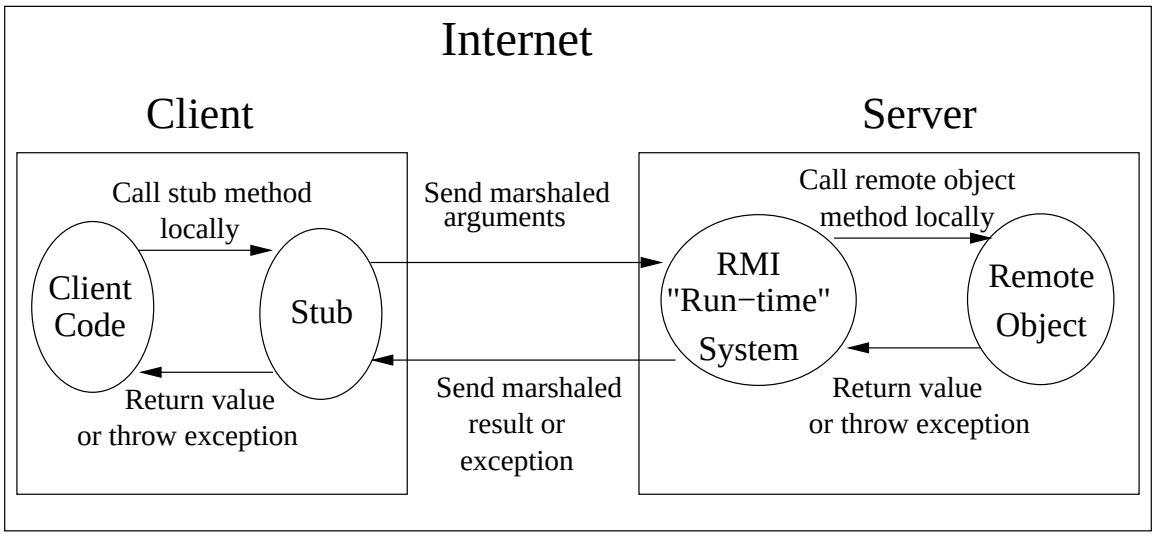

Figure 2.1. RMI Architecture.

network or to a file-system. In order to provide the functionality, an object must implement the `Serializable` interface.

The server-side run-time system is responsible for listening for invocation requests on a suitable IP port, and dispatching them to the proper, remote object on the server.

Since RMI is designed for Web based client-server applications over slow network, it is not clear it is suitable for high performance distributed computing environments with low latency and high bandwidth. A better serialization would be needed, since Java's current object serialization often takes at least 25% and up to 50% of the time [50] needed for a remote invocation.

2.1.4 JavaParty

The *JavaParty* [51] platform, which is a system from the University of Karlsruhe, Germany, has been developed with two main purposes. The two purposes are to serve as a programming environment for cluster applications, and to provide a basis for computer science research in optimization techniques to improve locality and reduce communication time. Remote objects, in JavaParty, are added to Java purely by declaration, avoiding disadvantages of explicit socket communication and the programming overhead of RMI.

JavaParty provides a mechanism that can discover remote objects and remote threads on different nodes without any programmer's extra work.

JavaParty is implemented by a pre-processing phase added to EspressoGrinder [46] and Pizza [47] Java compilers. After JavaParty code is transformed into regular Java code plus RMI hooks, Sun's RMI generator (stub and skeleton generator) take the resulting RMI portions.

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

2.1.5 Jini

As we mentioned in previous chapter, *Jini* technology may eventually become a core part of the underlying communication environment for our HPJava project (see section 5.5.4). In this section we will give a more general overview of Jini technology.

Jini technology [7, 24], built by Sun Microsystems, is a programming model that makes it easier for programmers to get their devices talking to each other. It is a lightweight layer of Java code that rests on top of the Java 2 platform. Jini provides a mechanism to enable smooth adding, removal, and finding of devices and services on the network without device vendors having to agree on network level protocols to allow their devices to interact. A service can be anything that sits on the network and is ready to perform a useful function. Hardware devices, software, and communications channels can all offer up their services to a dynamic network in a standard fashion without a central controlling authority.

Jini's architecture takes advantage of one of the fundamental features of object-oriented programming—the separation of interface and implementation. Jini depends on Java's RMI to provide the underlying distributed communications framework. The runtime infrastructure uses one network-level protocol, called *discovery*, and two object-level protocols, called *join* and *lookup*. Discovery enables clients and services to locate lookup services. Join enables a service to register itself in a lookup service. Lookup enables a client to query a lookup service for services that can help the client accomplish its goals. In addition the Jini

programming model includes three important API collections: *Distributed events*, *distributed leasing*, and *distributed transactions*. The *distributed event* programming model offers a set of interfaces to allow services to notify each other of changes in their state. The distributed leasing model provides a set of interfaces that encourage distributed objects to obtain *leases* for use of particular distributed resources. The service is responsible for renewing that lease before expires. If the lease is not renewed, access to the distributed resource expires, and resources are freed. The purpose of this feature is that it cleans up resources no longer in use by potentially failed distributed objects. For example, the lookup service automatically unregisters a failed service object. Finally, a *distributed transactions* programming model is to provide services for coordinating distributed operations, and guarantee that either all must occur atomically or that none occur at all.

2.2 Peer to Peer Computing

The term peer-to-peer (P2P) became fashionable in the computing field around the middle of the year 2000. A P2P network differs from conventional client/server or multitiered server's networks. A P2P architecture provides direct communication between peers without the reliance on centralized servers or resources. The claim for P2P architecture is that enables true distributed computing, creating networks of computing resources. Computers that have traditionally been used as clients can act as both clients and servers. P2P allows systems to have temporary associations with one other for a while, and then separate. P2P is an umbrella term for a rather diverse set of projects and systems including Napster, Freenet, and SETI@home. Here we discuss a select few that are more relevant to our interests.

2.2.1 Javelin

Javelin predates fashion for the P2P, but resembles other projects described in this section. Javelin [18] from the University of California, Santa Barbara, is an Internet-based global computing infrastructures that supports Java. Main goals of this system are to enable everyone connected to the Internet to easily participate in Javelin, and to provide an efficient infrastructure that supports as many different programming models as possible

without compromising portability and flexibility. The Javelin design is exploit widely used components—Web browsers and the Java language—to achieve those goals.

Three kinds of participating entities exist inside the Javelin system. The three entities are *clients*, which are processes seeking computing resources, *hosts*, which are processes offering computing resources, and *brokers*, which are processes that coordinate the supply and demand for computing resources. The broker gets tasks from clients and assigns to hosts that registered with the broker. Hosts send result back to the clients after finish running tasks. Users can make their computers available to host part of a distributed computation by pointing their browser to a known URL of a broker.

Javelin is good for loosely-coupled parallel applications ("task parallelism") but in itself it does not address the more tightly-coupled SPMD programming considered by HPJava.

2.2.2 SETI@home

SETI@home [52], a project that searches for extraterrestrial intelligence using networked personal computers, was launched early 1998. The Internet clients analyze data that is collected by the Arecibo radio telescope looking for possible indication of extraterrestrial intelligence. The collected data is divided into *work units* of 0.25 Mbyte. A work unit is big enough to keep a computer busy for a while and small enough to transmit in a few minutes even for a 28.8Kbps modem.

The essential part of this project is the client program, which is a screen saver for Windows or Macintosh users. Hence the client program will run only when computer is not being used. A Sign-up client gets a work unit from SETI@home data distribution server. After finishing processing its data, the client sends results back to the server and gets a new work unit. The SETI@home data server connects only when transferring data. To protect the job from computer failure, the client program writes a "check point" file to disk. Hence the program can pick up where it left off. The SETI@home data distribution server keeps track of the work units with a large database. When the work units are returned, they are put back into the database and the server looks for a new work unit and sends it out. Each work unit is sent out multiple times in order to make sure that the data is processed correctly.

SETI@home is faster then ASCI White, which is currently the fastest supercomputer, having peak performance of 12.3×1024 floating-point operations per seconds (TFLOPS).

3.1 trillion floating-point operations are required by a work unit to compute an FFT. SETI@home clients (between them) process about 700,000 work units in a typical day. This works out to over 20 TFLOPS. SETI@home also costs less then 1% compared to ASCI White.

In the first week after the launch, over 200,000 people downloaded and ran the client. This number has grown to 2,400,000 as of October 2000. People in 226 countries around the world run SETI@home. You can find more information about this project and client sign up are available from `http://setiathome.ssl.berkeley.edu`.

The SETI@home approach is not suitable for all problems. It must be possible to factor the problem into a large number of pieces that can be handled in parallel, with few or no interdependencies between the pieces.

2.2.3 Parabon

Parabon Computation, Inc. [48] announced a commercial distributed application called *Frontier* at the Supercomputing 2000 conference. The Frontier is available as a service over the Web or as service software. It claims to deliver scalable computing capacity via the Internet to a variety of industries.

The *Client Application*, the *Pioneer Compute Engine*, and the *Frontier Server* are three components of the Frontier platform. The client application runs on a single computer by an individual or organization wishing to utilize the Frontier platform by communicating with the Frontier server. The Pioneer Compute Engine is a desktop application that utilizes the spare computational power of an Internet-connected machine to process small units of computational work called tasks during idle time. The Frontier Server is the central hub of the Frontier platform, which communicates with both the client application and multiple Pioneer compute engines. It coordinates the scheduling and distribution of tasks.

Frontier gets a "job"—defined by a set of elements and a set of tasks to perform computational work—from user. Then like work units in SETI@home, a job is divided into an arbitrary set of individual tasks that is executed independently on a single node. Each task is defined by a set of elements it contains a list of parameters, and an entry point in the form of a JavaTM class name.

The inherent security features of JVM technology was the main reason to choose Java over other programming languages as programming language of Frontier. The JVM provides a "sandbox" inside which an engine can securely process tasks on a provider's computer. Valid Java bytecode has to be sent to engines and used to run tasks within the JVM.

In this project sustaining a large network of "volunteer" machines is a problem. Not many consumers are willing to donate computing cycles for purely commercial projects. Parabon system is good for task parallel applications but arguably is not peer to peer computing in sense tasks cannot communicate. Parabon system is less appropriate for the more tightly-coupled SPMD programming we are interested in.

2.2.4 JXTA

Project JXTA [36], an industry-wide research project led by Sun Microsystems, was launched on April 25th, 2001. The goal of Project JXTA is to develop protocols for cross-platform communication that provide direct access from one node to another without any centralized server control, and to create a simple, open, lightweight layer of standards that ensures interoperability and scalability when existing software stacks are integrated. Even though current JXTA is developed on top of Java technology, JXTA supposed to be independent from programming platforms, systems platforms, and networking platforms. The claim is that it can be embraced by all developers, independent of their preferred programming languages, existing development environments, or targeted deployment platforms.

Currently existing software technologies such as Java, Jini, and Extensible Markup Language (XML) are used by JXTA technology. The goal is a P2P system that is familiar to developers and easy to use. The benefit of using Java technology is the ability to compute on different machines without worrying about operating system limitations. Jini network technology enables spontaneous networking of a wide variety of hardware, and services. XML technology allows data to move across a network in a widely used format.

JXTA is developed with the Java 2 Platform, Micro Edition (J2ME) environment. The core classes of JXTA are packed into a jar file of about 250Kbytes. Hence JXTA can easily be stored in many different wireless mobile devices such as Personal Data Assistants, cell phones, and laptops, which can thus be nodes on a P2P network. This makes it possible

to access information directly from a PDA to a laptop without going through a centralized server.

JXTA has a limited number of concepts at it core. Here we overview some important concepts. A *peer* is any network device that implements one or more of the JXTA protocols. A collection of peers which have common interests can organize into a *peer group*. We can identify each peer group by a unique *peer group id*. Two peers can send and receive message using *pipes*—currently unidirectional virtual communication channels. All network resources, such as peers, peer groups, pipes and services are represented by an advertisement, which is JXTA's language neutral metadata structure for describing such resources.

Along with Jini technology, JXTA technology may become a very useful tool for our pure Java version of HPJava runtime environment.

2.3 Java Enterprise Computing

Our project is not particularly targeting enterprise computing. However we will review some related Java technologies in this area, namely the Java Message Service, which makes possible to exchange messages between Java programs, and Enterprise Java Beans, which implements server-side applications.

2.3.1 Java Message Service

Java Message Service (JMS) [35], designed by Sun Microsystems and several partner companies provides standard APIs that allows applications to create, send, receive, and read messages. It has been a part of the Java 2 Enterprise Edition since release 1.2. Java programs can exchange messages with other Java programs by sharing a messaging system in JMS. A messaging system, sometimes called Message-Oriented Middleware (MOM), accepts messages from "producer" clients and delivers them to "consumer" clients. Messaging systems are peer-to-peer distributed systems in the sense that every registered client can communicate with every other registered client. The term *messaging* is used to describe asynchronous communication between enterprise applications.

JMS supports point-to-point (PTP) and publish/subscribe message passing strategies. The PTP message model is based on *message queues*. A `QueueSender` class (producer)

sends a message to a specified queue. A `QueueReceiver` class (consumer) receives messages from the queue. The publish/subscribe messaging model is organized around *topics*. A `TopicPublishers` class (producer) sends messages to a topic. A `TopicSubscribers` class (consumer) retrieves messages from a topic. JMS also supports both synchronous and asynchronous message passing.

A JMS application is composed of the following parts: a *JMS provider*, *JMS clients*, *messages*, *administered objects*, and *native clients*. A JMS provider is a messaging product that implements the JMS interfaces and provides administrative and control features. JMS clients are programs or components written in the Java programming language that produce and consume messages. Messages are the objects that communicate information between JMS clients. An administrator creates administered objects, which are JMS objects, for use of client. There are two kinds of administered objects: *destinations* and *connection factories*. A client uses a destination object to specify the target and the source of messages. Connection configurations that have been defined by the administrator are contained in a connection factory. Native clients are programs that use a messaging product's native client API instead of the JMS API. Figure 2.2 illustrates the way these parts interact. After we bind destinations and connection factories into a Java Naming Directory Interface (JNDI) namespace, a JMS client looks up the administered objects in the namespace. Then the JMS provider provides the same objects for a logical connection to the JMS client.

Unlike the Remote Procedure Call (RPC) synchronous model, where we block and wait until each system finishes processing a task, a fundamental concept of MOM is that communication between applications is asynchronous. Code that is written to communicate assumes there is a one-way message that requires no immediate response from another application. There is also a high degree of anonymity between producer and consumer. Information about the owner of produced message is not important for the message consumer. This anonymity helps provide dynamic, reliable, and flexible systems.

2.3.2 Enterprise JavaBeans (EJB)

Enterprise JavaBeans (EJB) [25] implements server-side, arbitrarily scalable, transactional, multi-user, secure enterprise-level applications. It is Java's component model for enterprise applications. EJB combines distributed object technologies such as CORBA and

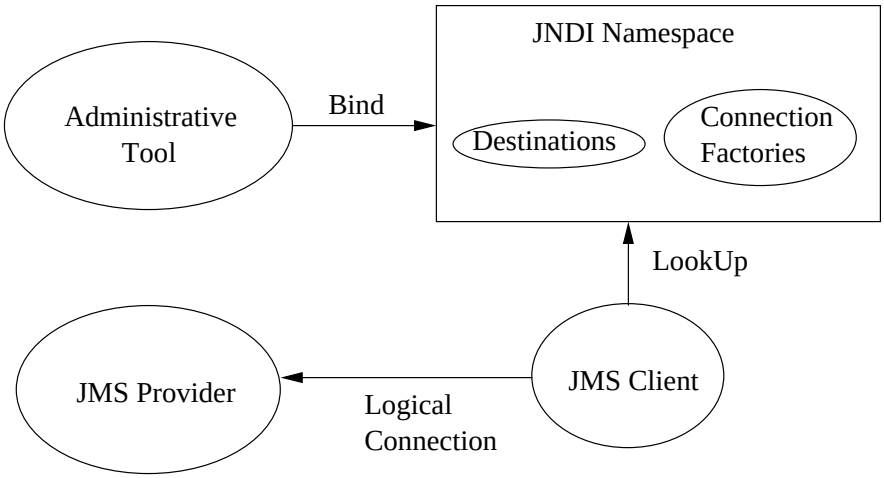

Figure 2.2. The JMS API Architecture

Java RMI with server-side components to simplify the task of application development. EJBs can be built on top of existing transaction processing systems including traditional transaction processing monitors, Web servers, database servers, and application servers. Sun claims that EJB is not just platform independent—it is also implementation independent. An EJB component can run in any application server that implements the EJB specification [22].

User applications and beans are isolated from the details of the component services. The ability to reuse the same enterprise bean in different specific applications is one advantage of this separation. The bean implementation or the client application need not have the parameters that control a bean's transactional nature, persistence, resource pooling, or security management. These parameters can be specified in separate deployment descriptors. So, when a bean is deployed in a distributed application, the properties of the deployment environment can be accounted for and reflected in the setting of the bean's options.

EJB is a distributed component model. A distributed component model defines how components are written. Hence different people can build systems from components with little or no customization. EJB defines a standard way of writing distributed components. The EJB client only gets a reference to an `EJBObject` instance and never really gets a reference to the actual EJB Bean instance itself. The `EJBObject` class is the client's view of the enterprise Bean and implements the remote interface.

EJB is a rather specialized architecture—aimed at transaction processing and database access. It is not really an environment for general distributed computing in our sense.

2.4 Comparison of Languages

In this section we compare Java as a language to support parallel and scientific computing with other object-oriented programming languages like C++ and *C#*, which is a new .NET platform language from Microsoft. In this comparison we discuss issues like performance and added functionality, such as multidimensional arrays, operator overloading, and lightweight objects. We also compare the Microsoft intermediate language with the Java Virtual Machine (JVM).

2.4.1 Java

Java started out as a Sun Microsystems funded internal corporate research project that tried to develop intelligent consumer electronic devices in 1990. A C and C++ based language called Oak—the former name of Java—was developed as a result of this project. After the World Wide Web exploded in popularity in 1993, the potential of using Java to create Web pages emerged. Although Java applets have lost some popularity, the Java language has continued to gain new applications.

The Java Grande Forum [33] has been set up to co-ordinate the community's efforts to standardize many aspects of Java and so ensure that its future development makes it more appropriate for scientific programmers. A "Grande" application can be described as one with a large scale nature, potentially requiring any combination of computers, network, I/O, and memory-intensive applications. Examples include, financial modeling, aircraft simulation, climate and weather, satellite image processing and earthquake predication. Two major working group of the Java Grande Forum are the Numerics Working Group and the Applications and Concurrency Working Group.

The Applications and Concurrency Working Group has been looking directly at uses of Java in parallel and distributed computing. The Numerics Working Group [45] has focused 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.

Associated developments have helped establish the case that Java can meet the vital performance constraints for numerically intensive computing. A series of papers from IBM [40, 41, 56], for example, demonstrated how to apply aggressive optimizations in Java compilers to obtain performance competitive with Fortran. In a recent paper [42] 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.

2.4.2 C++

C++, an extension of C, was developed by Bjarne Stroustrup in the early 1980s at Bell Laboratories. The most important feature of C++, and difference from C, is that C++ provides capabilities for object-oriented programming. C++ was recently standardized by the ANSI and ISO committees.

Java is a descendant of C++. Although a lot of syntax has been borrowed from C++, Java made no attempt to be compatible with C++. Java is designed to run over the Internet, and that required it take advantage of new language technology—for example, advances in automatic garbage collection, and just-in-time compilation.

There are some similarities of Java and C++, like C-style statements, expressions, and declarations—and classes, access privileges, virtual functions and overloading. Java, however, omits various features of C and C++ that are considered "difficult"—notably, pointers. Poor compiler analysis has often been blamed on these features. Java has fewer rules to remember. This is sometimes because it doesn't support the operations (operator overloading, for example), and sometimes because it does the work for you (automatic garbage collection, for example).

In recent years numerous variations on the theme of C++ for parallel computing have appeared. See, for example [8, 17, 23, 28, 31, 55].

2.4.3 C#

C# [20], which is an important part of the new Microsoft .NET platform, is a modern, object-oriented and type-safe programming language based on C and C++. Header files, Interface Definition Language, and complicated interfaces are not needed in C#. C# is supposed to offer an alternative to C++ programmers who find that language too complicated, and to Java programmers who miss certain features of C and C++, like pointers, operator overloading, and templates.

C# has all the data types provided by the Java, with additional unsigned counterparts and a new 12-byte-decimal floating-point number. Java uses primitive types that are distinguished from object-based types. Java primitive types must be put into an instance of a wrapper class to participate in the object-based world. C# provides what it calls a "unified type system". This means all types—including value types—derive from the type `object`. Primitive types are stack–allocated as in Java, but are also considered to derived from the ultimate base class, `object`. This means that the primitive types can have member functions called on them. The example:

```
using System;
class Test
{
  static void Main()
  {
    Console.WriteLine(3.ToString());
  }
}
```

calls the object-defined `ToString` method on an integer literal.

Whenever a primitive type is used in a situation where a parameter of type `object` is required, the compiler will automatically *box* the primitive type into a heap-allocated wrapper. An "object box" is allocated to hold the value, and the value is copied into the box. Unboxing is just the opposite. When an object box is cast back to its original value type, the value is copied out of the box and into the appropriate storage location. For example:

```
class Test
{
  static void Main()
  {
    int i = 123;
    object o = i;     // boxing
    Console.WriteLine(‘‘Value is: {0}’’, o);
    int j = (int) o;  // unboxing
  }
}
```

Declaring the object variable is done for illustration only; in real code, `i` would be passed directly, and the boxing would happen at the call site.

Unlike Java we can pass method arguments *by reference* in C# using `ref` and `out` modifiers. A reference parameter is declared with a `ref` modifier. The same storage location is used between a reference parameter and the variable given as the argument. An example usage of these parameters may look like:

```
void Swap(ref int x, ref int y) {
  int temp = x;
  x = y;
  y = temp;
}
```

A `out` modifier is used to declare a output parameter. Like reference parameters, output parameters do not create a new storage location. An output parameter need not be assigned *before* it is passed as a argument of a method.

C# also provides C-like pointers through unsafe code. In order to use unsafe code we should specify the `unsafe` modifier on the code block. Unsafe code can be used in place of Java Native Methods in many cases.

As mentioned in section 2.4.1, the Java Grande Numerics Working Group identified various critical areas to improve the Java language. C# addresses some of issues by supporting *structs*, *operator overloading*, and *multidimensional arrays*.

Structs can be used instead of classes when the user wants to create an object that behaves like one of the built-in types; one that is cheap and fast to allocate and doesn't have the overhead of references. Structs act similarly to classes, but with a few added restrictions.

They are value types rather than reference types, and inheritance is not supported for structs. Struct values are stored either "on the stack" or "in-line".

C# provides operator overloading that allows user-defined operators to be implemented on classes or structs so that they can be used with operator syntax. Unlike C++, it is not possible to overload member access, member invocation (function calling), or the +, &&, ||, ?:, or `new` operators. The `new` operation can't be overloaded because the .NET Runtime is responsible for managing memory.

Like our HPJava system, C# supports both "rectangular" and "jagged" multi-dimensional arrays[1]. Rectangular arrays always have a rectangular shape. Given the length of each dimension of the array, its rectangular shape is clear. A jagged array is merely an array of arrays and it doesn't have to be square.

Microsoft's .NET framework is based on its Common Language Runtime (CLR), which is a specification for language-independent intermediate language (IL) code, and a runtime that provides memory management and security. Whereas Java programs can run on any platform supporting JVM, and are compiled to byte code, which is an intermediate language only for Java, C# demands the standardization of Microsoft intermediate language (MSIL) [1]. Unlike Java, which is platform independent, MSIL has a feature called *language independence*: code and objects written in one language can be compiled to MSIL format and interoperate with other languages. However, for classes to be usable from .NET languages in general, the classes must adhere to the *Common Language Specification (CLS)*, which describes what features can be used internally in a class. This significantly restricts C++ programs, for example, if they are to run under .NET.

The MSIL instruction set is very similar in many ways to the JVM instruction set. The MSIL assumes a stack-based abstract machine very similar to the JVM, with a heap, a frame stack, the same concept of stack frame, and bytecode verification.

There are some rather simple differences—for example JVM words are big endian (most significant byte first) whereas MSIL uses a little endian (least significant byte first) binary representation. Also MSIL instructions do not include information that specifies the type of the arguments. Rather, that is inferred by what is been pushed on the stack. We can find

[1]Of course C# doesn't support *distributed* multidimensional arrays. Nor does it support Fortran 90-like regular sections

more fundamental differences between JVM instruction sets and MSIL in the verification of compiled code, and pointers.

JVM verification ensures that the binary representation of a class or interface is structurally correct. The following lists of checks are performed during JVM verification.

- Every instruction has a valid operation code.
- Every branch instruction branches to the start of some other instruction, rather than into the middle of an instruction.
- Every method is provided with a structurally correct signature.
- Every instruction obeys the type discipline of the Java language.

Unlike JVM, several important uses of MSIL instructions are not verifiable, such as the pointer arithmetic versions of `add` that are required for the faithful and efficient compilation of `C` programs. For nonverifiable code, memory safety is the responsibility of the application programmer.

The only kind of pointer manipulated by the JVM instruction set is the *object reference*. MSIL manipulates three different kinds of pointers: object reference like JVM, *managed pointers* for reference argument variables, and *unmanaged pointers* for unsafe code pointers. Managed pointers must be reported to the garbage collector. The garbage collector can modify the managed pointer itself, as well as the contents of the location which is pointed to. The unmanaged pointers get great flexibility in their operation since they are not reported to the garbage collector. But memory safety is endangered by performing arithmetic on unmanaged pointers; hence they are not verifiable.

2.4.3.1 MPI with C#

The MPI binding of the C# language and the CLI [54] has been developed by the Open Systems Laboratory at Indiana University. Two levels of language bindings are presented. A set of C# bindings to MPI is a low-level interface similar to the existing C++ bindings given in the MPI-2 specification. The other library called MPI.NET is a high-level interface. This interface integrates modern programming language features of C#.

The C# binding is relatively straightforward. Each object of C# bindings contains the underlying C representation of the MPI object. Similarly, the high-level objects in the MPI.NET are usually containers of underlying MPI objects. Both the C# binding and MPI.NET libraries are built on top of a native implementation of MPI. The P/Invoke feature of CLI is used to bind lowest level of both libraries to the C MPI functions. A P/Invoke is a mechanism to create direct interface to existing libraries written in C and C++ in the .NET programming environments.

Since the MPI.NET binding uses language features of C#, it has unique features. The MPI.NET uses a feature of C# properties. This feature allows to access a field of object more conveniently by providing special syntax like method. This feature simplifies interface of information like rank of the current process or number of nodes. Unlike the C# bindings, automatic pinning and unpinning of user defined data is also provided in the MPI.NET binding. A more minor difference is the C# bindings try to follow naming convention of MPI C++ bindings and the MPI.NET library tries to follow C# naming convention.

According to [54], performance of the current MPI binding of the C# and the CLI is slightly slower than our MPI binding to Java (mpiJava).

CHAPTER 3

REVIEW OF THE HPJAVA ENVIRONMENT

In this chapter we will discuss ongoing work within our research group on the HPJava system. HPJava is an environment for SPMD (Single Program, Multiple Data) parallel programming—especially, for SPMD programming with distributed arrays. The HPJava language design and its features are discussed first. The results reported in this dissertation concentrate in particular on issues related to the development of efficient run time support software for parallel languages extending an underlying object-oriented language. So the review emphasizes aspects most relevant to this topic—aspects like high performance interprocessor communication.

3.1 Introduction

The *SPMD (Single Program Multiple Data)* programming style is the most commonly used style of writting data-parallel programs for *MIMD (Multiple Instruction Multiple Data)* systems. This style provides a *Loosely Synchronous Model*. In SPMD programming style, each processor has its own local copy of control variables and datas. Processors execute the same SPMD program asynchronously across MIMD nodes. Explicit or implicit synchronization takes place only when processors need to exchange data. Conditional branches within the source code may provide the effect of running different programs. There have been many data-parallel languages. These programming languages provided some consensus about outline of a standard language for SPMD programming. The High Performance Fortran (HPF) standard was established by the *High Performance Fortran Forum* in 1993.

HPF is an extension of Fortran 90 to support the data-parallel programming model on distributed memory parallel computers. It extends the set of parallel features to Fortran 90. The HPF language especially targets the SPMD model. Each processor runs same

program and operates on parts of the overall data. The HPF programs are simpler to write than explicit message-passing programs, but performance of HPF rarely achieves the efficiency of the message-passing. Although the HPF language might never be widely adopted, many of the ideas—for example its standardization of a distributed data model for SPMD computing—remain important.

Over many years the SPMD programming style has been very popular among the high-level parallel programming environment and library developers. SPMD framework libraries try to overcome weakness of HPF. While HPF has strength on rather regular data structures and regular data access patterns problems, it is not particularly suitable for irregular data access. Some libraries are dealing directly with irregularly distributed data, DAGH [49], Kelp [27], and other libraries support unstructured access to distributed arrays, CHAOS/PARTI [21] and Global Arrays [44]. While the library-based SPMD approach to data-parallel programming may address weakness of HPF, it loses good features like the uniformity and elegance that promised by HPF. There are no compile-time or compiler-generated run-time safety checks for the distributed arrays because libraries manage the arrays. The our *HPspmd model* is an attempt to address such shortcomings.

The HPspmd model is SPMD programming supported by additional syntax for HPF-like distributed arrays. This model provides a hybrid of the data parallel model of HPF and the low-level SPMD programming style—as often implemented using communication libraries like MPI. In the HPspmd model, a subscripting syntax can be used to directly access *local* elements of distributed arrays. References to these elements can only be made on processors that hold copies of the elements concerned. To ensure this, a well-defined set of rules are automatically checked by the translator. Even though the HPspmd model does provide special syntax for HPF-like distributed arrays, all access to non-local array elements should go through library function calls *in the source program*. These library calls must be placed in the original HPJava program by the programmer. This requirement may be unusual to people expecting high-level parallel languages like HPF, but it should not seem particularly unnatural to programmers used to writing parallel programs using MPI or other SPMD libraries. The exact nature of the communication library used is not part of the HPJava language design. Collective operations on whole distributed arrays , or some kind of `get` and `put` functions for access to remote blocks of a distributed array might be provided

by an appropriate communication library. This dissertation concentrates especially on one particular communication library called Adlib.

In the current system, syntax extensions are handled by a preprocessor that outputs an ordinary SPMD program in the base language. The HPspmd syntax provides a fairly thin veneer on low-level SPMD programming, and the transformations applied by the translator are correspondingly direct—only limited analysis should be needed to obtain good parallel performance. But the language does provide a uniform model of a distributed array, which can be targeted by reusable libraries for parallel communication and arithmetic. The model adopted very closely follows the distributed array model defined in the High Performance Fortran standard.

3.2 HPJava—an HPspmd language

HPJava [15] is a particular implementation of the HPspmd idea. It is a strict extension of its base language, Java, adding some predefined classes and some extra syntax for dealing with distributed arrays. HPJava is thus an environment for parallel programming, especially suitable for data parallel scientific programming. To some extent the choice of base language is accidental, and we could have added equivalent extensions to another language, such as Fortran itself. But Java does seem to be a better language in various respects, and it seems likely that in the future more software will be available for modern object-oriented languages like Java than for Fortran.

An HPJava program can freely invoke any existing Java classes without restrictions because it incorporates all of Java as a subset. A concept of multidimensional distributed arrays—closely modeled on the arrays of HPF[1]—has been added to Java. Regular sections of distributed arrays are fully supported. Distributed arrays can have any rank greater than or equal to zero and the elements of distributed arrays can be of any standard Java type, including primitive types, Java class types and ordinary Java array types.

A standard Java class file is produced after translating and compiling a HPJava program. This Java class file will be executed by a distributed collection of Java Virtual Machines. All externally visible attributes of an HPJava class—e.g. existence of distributed-array-valued

[1]"Sequential" multi-dimensional arrays are available as a subset of the HPJava distributed arrays.

```
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.

fields or method arguments—can be automatically reconstructed from Java signatures stored in the class file. This makes it possible to build libraries operating on distributed arrays, while maintaining the usual portability and compatibility features of Java. The libraries themselves can be implemented in HPJava, or in standard Java, or through Java Native Interface (JNI) wrappers to code implemented in other languages. The HPJava language specification carefully documents the mapping between distributed arrays and the standard-Java components they translate to.

Figure 3.1 is a simple HPJava program. It illustrates creation of distributed arrays, and access to their elements. An HPJava program is started concurrently in some set of processes that are named through *grids* objects. The class `Procs2` is a standard library class, and represents a two dimensional grid of processes. During the creation of p, P by P processes are selected from the *active process group*. The `Procs2` class extends the special base class `Group` which represents a group of processes and has a privileged status in the HPJava language. An object that inherits this class can be used in various special places. For example, it can be used to parameterize an *on construct*. The `on(p)` construct is a new control construct specifying that the enclosed actions are performed only by processes in group p.

The *distributed array* is the most important feature HPJava adds to Java. A distributed array is a collective array shared by a number of processes. Like an ordinary array, a distributed array has some index space and stores a collection of elements of fixed type. The type signature of an r-dimensional distributed array involves double brackets surrounding r comma-separated slots. A hyphen in one of these slots indicates the dimension is distributed. Asterisks are also allowed in these slots, specifying that some dimensions of the array are not to be distributed, i.e. they are "sequential" dimensions (if *all* dimensions have asterisks, the array is actually an ordinary, non-distributed, Fortran-like, multidimensional array—a valuable addition to Java in its own right, as many people have noted [42, 43]).

In HPJava the subscripts in distributed array element references must normally be distributed indexes (the only exceptions to this rule are subscripts in sequential dimensions, and subscripts in arrays with ghost regions, discussed later). The indexes must be in the distributed range associated with the array dimension. This strict requirement ensures that referenced array elements are held by the process that references them.

The variables a, b, and c are all distributed array variables. The creation expressions on the right hand side of the initializers specify that the arrays here all have ranges `x` and `y`—they are all `M` by `N` arrays, block-distributed over `p`. We see that mapping of distributed arrays in HPJava is described in terms of the two special classes `Group` and `Range`.

The *Range* is another special class with privileged status. It represents an integer interval 0,..., N - 1, distributed somehow over a *process dimension* (a dimension or axis of a grid like p). `BlockRange` is a particular subclass of `Range`. The arguments in the constructor of `BlockRange` represent the total size of the range and the target process dimension. Thus, x has `M` elements distributed over first dimension of p and y has `N` elements distributed over second dimension of p.

HPJava defines a class hierarchy of different kinds of range object (Figure 3.2). Each subclass represents a different kind of distribution format for an array dimension. The simplest distribution format is *collapsed* (sequential) format in which the whole of the array dimension is mapped to the local process. Other distribution formats (motivated by High Performance Fortran) include *regular block* decomposition, and *simple cyclic* decomposition. In these cases the index range (thus array dimension) is distributed over one of the dimensions of the process grid defined by the group object. All ranges must be distributed over different

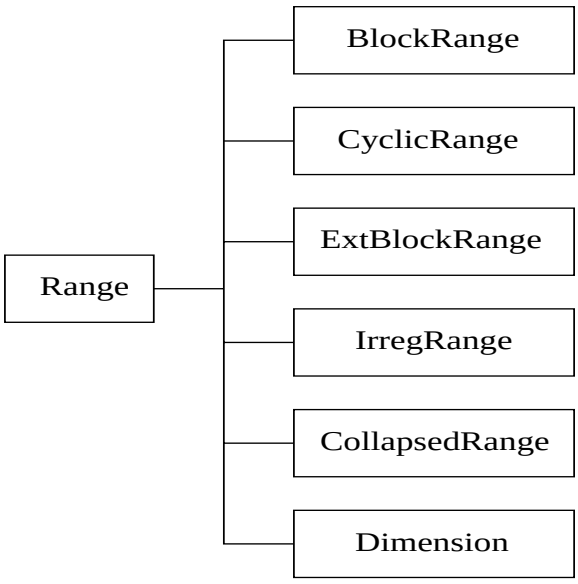

Figure 3.2. The HPJava Range hierarchy

dimensions of this grid, and if a particular dimension of the grid is targeted by none of the ranges, the array is said to be *replicated* in that dimension[2]. Some of the range classes allow *ghost extensions* to support stencil-based computations.

A second new control construct, `overall`, implements a distributed parallel loop. It shares some characteristics of the *forall* construct of HPF. The symbols `i` and `j` scoped by these constructs are called *distributed indexes*. The indexes iterate over all locations (selected here by the degenerate interval ":") of ranges `x` and `y`.

HPJava also supports Fortran-like array sections. An *array section expression* has a similar syntax to a distributed array element reference, but uses double brackets. It yields a reference to a new array containing a subset of the elements of the parent array. Those elements can be accessed either through the parent array or through the array section—HPJava sections behave something like array pointers in Fortran, which can reference an arbitrary regular section of a target array. As in Fortran, subscripts in section expressions can be index triplets. HPJava also has built-in ideas of *subranges* and *restricted groups*. These describe the range and distribution group of sections, and can be also used in array

[2]So there is no direct relation between the array rank and the dimension of the process grid: collapsed ranges means the array rank can be higher; replication allows it to be lower.

constructors on the same footing as the ranges and grids introduced earlier. They allow HPJava arrays to reproduce any mapping allowed by the `ALIGN` directive of HPF.

The examples here have covered the basic syntax of HPJava. The language itself is relatively simple. Complexities associated with varied or irregular patterns of communication are supposed to be dealt with in communication libraries like the ones discussed in the remainder of this dissertation.

The examples given so far look very much like HPF data-parallel examples, written in a different syntax. We will give one last example to emphasize the point that the HPspmd model is not the same as the HPF model. If we execute the following HPJava program

```
Procs2 p = new Procs2(2, 3) ;
on(p) {
  Dimension d = p.dim(0), e = p.dim(1) ;

  System.out.println("My coordinates are (" + d.crd() +
                                          , " + e.crd() + ")") ;
}
```

we could see output like:

```
My coordinates are (0, 2)
My coordinates are (1, 2)
My coordinates are (0, 0)
My coordinates are (1, 0)
My coordinates are (1, 1)
My coordinates are (0, 1)
```

There are 6 messages. Because the 6 processes are running concurrently in 6 JVMs, the order in which the messages appear is unpredictable. An HPJava program is a MIMD program, and any appearance of collective behavior in previous examples was the result of a particular programming style and a good library of collective communication primitives. In general an HPJava program can freely exploit the weakly coupled nature of the process cluster, often allowing more efficient algorithms to be coded.

```
public void matmul(float [[-,-]] c, float [[-,-]] a, float [[-,-]] b) {

  Group2 p = c.grp();

  Range x = c.rng(0);
  Range y = c.rng(1);

  int N = a.rng(1).size();

  float [[-,*]] ta = new float [[x, N]] on p;
  float [[*,-]] tb = new float [[N, y]] on p;

  Adlib.remap(ta, a);
  Adlib.remap(tb, b);

  on(p)
    overall(i = x for : )
      overall(j = y for : ) {

        float sum = 0;
        for(int k = 0; k < N ; k++)
          sum += ta [i, k] * tb [k, j];

        c[i, j] = sum;
      }
}
```

Figure 3.3. A general Matrix multiplication in HPJava.

3.3 High-level Communication Library

In this section we discuss extra syntax and usage of high-level communication library in HPJava programs. Two characteristic collective communication methods `remap()` and `writeHalo()` are described as examples.

We discuss more detail information about the general purpose matrix multiplication routine (Figure 3.3). The method has two temporary arrays `ta`, `tb` with the desired distributed format. This program is also using information which is defined for any distributed array: `grp()` to fetch the distribution group and `rng()` to fetch the index ranges.

This example relies on a high-level Adlib communication schedule that deals explicitly with distributed arrays; the `remap()` method. The `remap()` operation can be applied to various ranks and type of array. Any section of an array with any allowed distribution format can be used. Supported element types include Java primitive and `Object` type. A general API for the `remap` function is

```
void remap (T [[]] dst, T [[]] src) ;
void remap (T [[-]] dst, T [[-]] src) ;
void remap (T [[-,-]] dst, T [[-,-]] src) ;
...
```

where `T` is a Java primitive or `Object` type. The arguments here are zero-dimensional, one-dimensional, two-dimensional, and so on. We will often summarize these in the shorthand interface:

```
void remap (T # dst, T # src) ;
```

where the signature `T #` means any distributed array with elements of type `T` (This *syntax* is not supported by the current HPJava compiler, but it supports method signatures of this generic kind in externally implemented libraries—ie. libraries implemented in standard Java. This more concise signature does not incorporate the constraint that `dst` and `src` have the same rank—that has to be tested at run-time.)

As another example, Figure 3.4 is a HPJava program for the Laplace program that uses *ghost regions*. It illustrates the use the library class `ExtBlockRange` to create arrays with ghost extensions. In this case, the extensions are of width 1 on either side of the locally held "physical" segment. Figure 3.5 illustrates this situation.

From the point of view of this dissertation the most important feature of this example is the appearance of the function `Adlib.writeHalo()`. This is a *collective communication operation*. This particular one is used to fill the *ghost cells* or *overlap regions* surrounding the "physical segment" of a distributed array. A call to a collective operation must be invoked simultaneously by all members of some active process group (which may or may not be the entire set of processes executing the program). The effect of `writeHalo` is to overwrite the ghost region with values from processes holding the corresponding elements in their physical segments. Figure 3.6 illustrates the effect of executing the `writeHalo` function.

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

  float [[-,-]] a = new float [[x, y]] ;

  ... initialize edge values in 'a'

  float [[-,-]] b = new float [[x, y]], r = new float [[x, y]] ;

  do {
    Adlib.writeHalo(a) ;

    overall(i = x for 1 : N - 2)
      overall(j = y for 1 : N - 2) {
        float newA = 0.25 * (a[i - 1, j] + a[i + 1, j] +
                             a[i, j - 1] + a[i, j + 1]);

        r[i,j] = Math.abs(newA - a[i,j]);
        b[i,j] = newA ;
      }

    HPutil.copy(a,b) ; // Jacobi relaxation.
  } while(Adlib.maxval(r) > EPS);
}
```

Figure 3.4. Solution of Laplace equation by Jacobi relaxation.

More general forms of `writeHalo` may specify that only a subset of the available ghost area is to be updated, or may select cyclic wraparound for updating ghost cells at the extreme ends of the array.

If an array has ghost regions the rule that the subscripts must be simple distributed indices is relaxed; *shifted indices*, including a positive or negative integer offset, allow access to elements at locations neighboring the one defined by the overall index.

Besides `remap()` and `writeHalo()`, Adlib includes a family of related regular collective communication operations (shifts, skews, and so on). It also incorporates a set of collective gather and scatter operations for more irregular communications, and a set of reduction

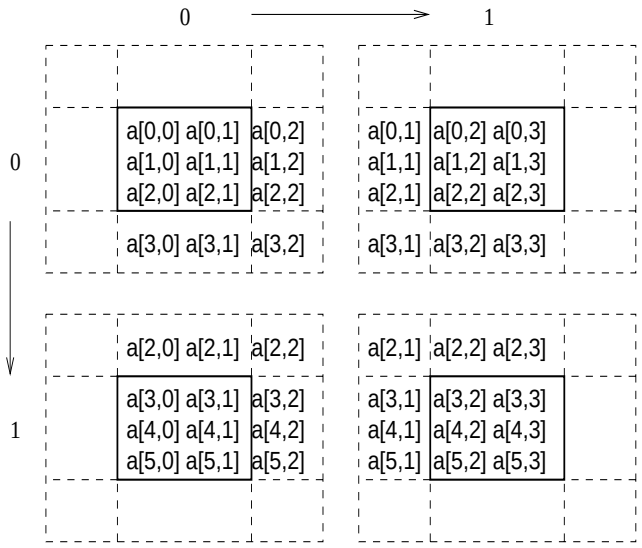

Figure 3.5. Example of a distributed array with ghost regions.

operations based on the corresponding Fortran 90 array intrinsics. Reduction operations take one or more distributed arrays as input. They combine the elements to produce one or more scalar values, or arrays of lower rank.

Currently our collective communication library is built on top of device level communication library called *mpjdev*. The mpjdev API is designed with the goal that it can be implemented portably on network platforms and efficiently on parallel hardware. This library is developed with HPJava in mind, but it is a standalone library and could be used by other systems. The main purpose of this library is to perform actual point-to-point communications between processes.

We will discuss implementation issues of high- and low-level communication libraries in Chapter 4 and Chapter 5.

3.4 Message-passing for HPJava

High level libraries like Adlib are very powerful. But in "real" parallel programming lower level SPMD approaches may also be necessary. HPJava therefore supports low-level message passing as well.

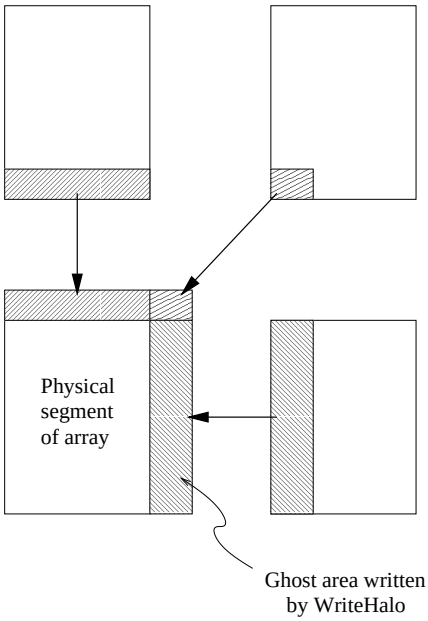

Figure 3.6. Illustration of the effect of executing the writeHalo function.

3.4.1 The mpiJava wrapper

The Message-Passing Interface (MPI) [39] was developed by a group of researchers from industry, government, and academia. It provides a rich set of communication libraries, application topologies and user defined data types. MPI is the first message-passing standard for programming parallel processors.

Our mpiJava software implements a Java binding for MPI proposed late in 1997. The API is modeled as closely as practical on the C++ binding defined in the MPI 2.0 standard, specifically supporting the MPI 1.1 subset of that standard. The mpiJava is developed by our research group and currently version 1.2 is available.

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

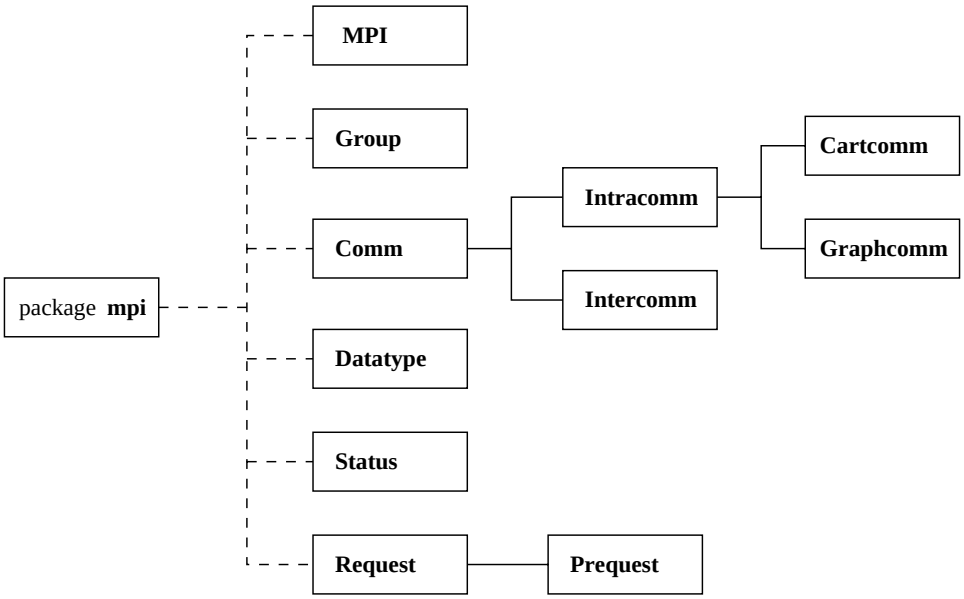

Figure 3.7. Principal classes of mpiJava

this model, lifting the structure of its class hierarchy directly from the C++ binding. The major classes of mpiJava are illustrated in Figure 3.7.

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

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

Another class that is important later discussion 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 data types are predefined in the package. These mainly correspond to the primitive types of Java, shown in Table 3.1 that allows Java objects to be communicated. The `MPI.OBJECT` type is a new predefined basic data type of mpiJava.

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

Table 3.1. Basic data types of mpiJava

MPI data type	Java data type
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`

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

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

In both cases the actual argument corresponding to `buf` must be a Java array. In 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.

A native MPI library, a version of the Java Development Toolkit (JDK), and a C compiler is necessary to install mpiJava. The MPI Java classes and the C stubs that bind the MPI Java classes to the underlying native MPI implementation are two core parts of mpiJava. We create these C stubs using Java Native Interface (JNI), which Java can call and pass parameters to and from a native API. The new version 1.2 of the software supports direct communication of objects via object serialization, which is an important step toward implementing the specification in [14].

The mpiJava software is available from

`http://www.hpjava.org/mpiJava.html`

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

3.4.2 Task-Parallelism in HPJava

Sometimes some parts of a large parallel program cannot be written efficiently in the pure data parallel style, using `overall` constructs to process all elements of distributed arrays homogeneously. Sometimes, for efficiency, a process has to do some procedure that combines just the locally held array elements in a non-trivial way.

The HPJava environment is designed to facilitate direct access to SPMD library interfaces. HPJava provides constructs to facilitate both data-parallel and task-parallel programming. Different processors can either simultaneously work on data in globally subscripted arrays, or independently execute more complex procedures on their own local data. The conversion between these phases is supposed to be relatively seamless.

As an example of the HPJava binding for MPI, we will consider a fragment from a parallel *N-body* classical mechanics problem. As the name suggests, this problem is concerned with the dynamics of a set of N interacting bodies. The total force on each body includes a contribution from all the other bodies in the system. The size of this contribution depends on the position, x, of the body experiencing the force, and the position, y, of the body exerting it. If the individual contribution is given by `force(`x, y`)`, the net force on body i is

$$\sum_j \text{force}(a_i, a_j)$$

where now a_j is the position of the jth body. A simplified pure data parallel version of force computation in a N-body program is illustrated in Figure 3.8. There are three distributed arrays in the program, `a`, `b` and `f`. We repeatedly rotate a copy, `b`, of the position vector, `a`, and contributions to the force are accumulated as we go. The trouble is that this involves N small shifts. Calling out to the communication library so many times (and copying a whole array so many times) is likely to produce an inefficient program.

One way to express the algorithm is in a direct SPMD message-passing style. Example code is given in Figure 3.9. In this HPJava/MPI version of N-body, the HPJava will manage process group arrangements and initialization for distributed arrays. We have used the method `Sendrecv_replace()`, a point-to-point communication routine between processors from the mpiJava binding of MPI, instead of the shift-operation from Figure 3.8. The local variables `a_block`, `b_block` and `f_block` in the program are not distributed arrays. And

```
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 'a' ...

  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 3.8. HPJava data parallel version of the N-body force computation.

they are assigned by an inquiry function call `dat()` that returns a sequential Java array containing the locally held elements of the distributed array. This HPJava/MPI version does P shifts of whole blocks of size B for sending N data instead of N small shifts in pure data parallel version. This reduces communication between nodes. The HPJava/MPI version also requires less copying operations (P times) than the pure data parallel version (N times), where typically $P \ll N$.

This example leaves some issues unresolved—in general what is the mapping from distributed-array elements to local-data-segment elements? It assumes each processor hold identical sized blocks of data (P exactly divides N). For a general distributed array or section, the local segment may be some stride subset of the vector returned by `dat()`. The complete

specification of HPJava addresses these issues. There is also an issue about the mapping between HPJava process groups and MPI groups. We need an MPI like library that is better integrated with HPJava constructs. We envisage an API tentatively called OOMPH (Object-oriented Message Passing for HPJava). The details have not been worked out. OOMPH would built on mpjdev, and fully interoperable with HPJava Adlib.

```
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 'a' ...

  overall(i = x for :) {
    f[i] = 0.0 ;
    b[i] = a[i] ;
  }

  // extract the local vectors of elements:

  float [] f_blk = f.dat(), a_blk = a.dat(), b_blk = b.dat() ;

  int myID = MPI.COMM_WORLD.Rank();

  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_blk[i], b_blk[j]) ;


    // cyclically shift 'b_blk' (by amount B in x dim)...

    int right = (myID + 1) % P, left = (myID + P -1) % P;

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

  }

}
```

Figure 3.9. Version of the N-body force computation using reduction to Java array.

CHAPTER 4

A HIGH-LEVEL COMMUNICATION LIBRARY FOR HPJAVA

In this chapter we discuss our high-level communication library, Adlib, in depth. Its detailed functionalities and implementation issues are described. We start with background of this library and move into two characteristic and important collective operations, `remap` and `writeHalo`. Based on those operations, we describe detailed implementation issues of collective communication. Finally other functionalities of collective communication operation are described.

4.1 Background

A C++ library called Adlib [12] was completed in the Parallel Compiler Runtime Consortium (PCRC) [19] project. It was a high-level runtime library designed to support translation of data-parallel languages [16]. Initial emphasis was on High Performance Fortran (HPF), and two experimental HPF translators used the library to manage their communications [38, 58]. It incorporated a built-in representation of a distributed array, and a library of communication and arithmetic operations acting on these arrays. The array model supported general HPF-like distribution formats, and arbitrary regular sections.

Initially HPJava used a JNI wrapper interface to the C++ kernel of the PCRC library. The library described here borrows many ideas form the PCRC library, but for this project we rewrote high-level library from the scratch for Java. It was extended to support Java object types, to target Java based communication platforms and to use Java exception handling—making it "safe" for Java. The Java version of the Adlib library is developed on top of *mpjdev*. The mpjdev API can be implemented portably on network platforms and efficiently on parallel hardware (see Chapter 5).

The Adlib series of libraries support *collective operations* on distributed arrays. A call to a collective operation must be invoked simultaneously by all members of some *active process group*, which may or may not be the entire set of processes executing the program. Communication patterns supported include HPF/Fortran 90 intrinsics such as `cshift`. More importantly they include the regular-section copy operation, `remap`, which copies elements between shape-conforming array sections regardless of source and destination mapping. Another function, `writeHalo`, updates ghost areas of a distributed array. Various collective gather and scatter operations allow irregular patterns of access. The library also provides essentially all Fortran 90 arithmetic transformational functions on distributed arrays and various additional HPF library functions.

4.2 Implementation of Collectives

In this section we will discuss Java implementation of the Adlib collective operations. For illustration we concentrate on the important `Remap` operation. Although it is a powerful and general operation, it is actually one of the more simple collectives to implement in the HPJava framework.

General algorithms for this primitive have been described by other authors in the past. For example it is essentially equivalent to the operation called *Regular_Section_Copy_Sched* in [6]. In this section we want to illustrate how this kind of operation can be implemented in term of the particular `Range` and `Group` classes of HPJava, complemented by suitable set of messaging primitives.

All collective operations in the library are based on communication *schedule* objects. Each kind of operation has an associated class of schedules. Particular instances of these schedules, involving particular data arrays and other parameters, are created by the class constructors. Executing a schedule initiates the communications required to effect the operation. A single schedule may be executed many times, repeating the same communication pattern. In this way, especially for iterative programs, the cost of computations and negotiations involved in constructing a schedule can often be amortized over many executions. This pattern was pioneered in the CHAOS/PARTI libraries [21]. If a communication pattern is to be executed only once, simple wrapper functions are made available to construct a schedule,

execute it, then destroy it. The overhead of creating the schedule is essentially unavoidable, because even in the single-use case individual data movements generally have to be sorted and aggregated, for efficiency. The data structures for this are just those associated with schedule construction.

Constructor and public method of the `remap` schedule for distributed arrays of float element can be summarized as follows:

```
class RemapFloat extends Remap {
  public RemapFloat (float # dst, float # src) {...}

  public void execute() {...}
  . . .
}
```

The # notation was explained in section 3.3.

The remap schedule combines two functionalities: it reorganizes data in the way indicated by the distribution formats of source and destination array. Also, if the destination array has a *replicated* distribution format, it broadcasts data to all copies of the destination. Here we will concentrate on the former aspect, which is handled by an object of class `RemapSkeleton` contained in every `Remap` object.

During construction of a `RemapSkeleton` schedule, all send messages, receive messages, and internal copy operations implied by execution of the schedule are enumerated and stored in light-weight data structures. These messages have to be sorted before sending, for possible message agglomeration, and to ensure a deadlock-free communication schedule. These algorithms, and maintenance of the associated data structures, are dealt with in a base class of `RemapSkeleton` called `BlockMessSchedule`. The API for the superclass is outlined in Figure 4.1. To set-up such a low-level schedule, one makes a series of calls to `sendReq` and `recvReq` to define the required messages. Messages are characterized by an offset in some local array segment, and a set of strides and extents parameterizing a multi-dimensional patch of the (flat Java) array. Finally the `build()` operation does any necessary processing of the message lists. The schedule is executed in a "forward" or "backward" direction by invoking `gather()` or `scatter()`.

In general Top-level schedules such as `Remap`, which deal explicitly with distributed arrays, are implemented in terms of some lower-level schedules such as `BlockMessSchedule` that

```
public abstract class BlockMessSchedule {

  BlockMessSchedule(int rank, int elementLen,boolean isObject) { ... }

  void sendReq(int offset, int[] strs, int[] exts, int dstId) { ... }

  void recvReq(int offset, int[] strs, int[] exts, int srcId) { ... }

  void build()   { ... }

  void gather()  { ... }

  void scatter() { ... }

  ...
}
```

Figure 4.1. API of the class `BlockMessSchedule`

Table 4.1. Low-level Adlib schedules

	operations on "words"	operations on "blocks"
Point-to-point	`MessSchedule`	`BlockMessSchedule`
Remote access	`DataSchedule`	`BlockDataSchedule`
Tree operations	`TreeSchedule`	`BlockTreeSchedule`
	`RedxSchedule`	`BlockRedxSchedule`
	`Redx2Schedule`	`BlockRedx2Schedule`

simply operate on blocks and words of data. These lower-level schedules do not directly depend on the `Range` and `Group` classes. The lower level schedules are tabulated in Table 4.1. Here "words" means contiguous memory blocks of constant (for a given schedule instance) size. "Blocks" means multidimensional (r-dimensional) local array sections, parameterized by a vector of r extents and a vector of memory strides. The point-to-point schedules are used to implement collective operations that are deterministic in the sense that both sender and receiver have advanced knowledge of all required communications. Hence `Remap` and other regular communications such as `Shift` are implemented on top of `BlockMessSchedule`. The "remote access" schedules are used to implement operations where one side must inform the

other end that a communication is needed. These negotiations occur at schedule-construction time. Irregular communication operations such as collective `Gather` and `Scatter` are implemented on these schedules. The tree schedules are used for various sorts of broadcast, multicast, synchronization, and reduction.

We will describe in more detail the implementation of the higher-level `RemapSkeleton` schedule on top of `BlockMessSchedule`. This provides some insight into the structure HPJava distributed arrays, and the underlying role of the special `Range` and `Group` classes.

To produce an implementation of the `RemapSkeleton` class that works independently of the detailed distribution format of the arrays we rely on virtual functions of the `Range` class to enumerate the blocks of index values held on each processor. These virtual functions, implemented differently for different distribution formats, encode all important information about those formats. To a large extent the communication code itself is distribution format independent.

The range hierarchy of HPJava was illustrated in Figure 3.2, and some of the relevant virtual functions are displayed in the API of Figure 4.2. Most methods optionally take arguments that allow one to specify a contiguous or strided subrange of interest. The `Triplet` and `Block` instances represent simple struct-like objects holding a few `int` fields. Those integer files are describing respectively a "triplet" interval, and the strided interval of "global" and "local" subscripts that the distribution format maps to a particular process. In the examples here `Triplet` is used only to describe a range of *process coordinates* that a range or subrange is distributed over.

Now the `RemapSkeleton` communication schedule is built by two methods called *sendLoop* and *recvLoop* that enumerate messages to be sent and received respectively. Figure 4.3 sketches the implementation of `sendLoop`. This is a recursive function—it implements a multidimensional loop over the `rank` dimensions of the arrays. It is initially called with `r = 0`. An important thing to note is how this function uses the virtual methods on the range objects of the source and destination arrays to enumerate blocks—local and remote—of relevant subranges, and enumerates the messages that must be sent. Figure 4.4 illustrates the significance of some of the variables in the code. When the offset and all extents and strides of a particular message have been accumulated, the `sendReq()` method of the base

```
public abstract class Range {
    public int size() {...}
    public int format() {...}
    ...
    public Block localBlock() {...}
    public Block localBlock(int lo, int hi) {...}
    public Block localBlock(int lo, int hi, int stp) {...}

    public Triplet crds() {...}
    public Block block(int crd) {...}

    public Triplet crds(int lo, int hi) {...}
    public Block block(int crd, int lo, int hi) {...}

    public Triplet crds(int lo, int hi, int stp) {...}
    public Block block(int crd, int lo, int hi, int stp) {...}
    . . .
}
```

Figure 4.2. Partial API of the class `Range`

class is invoked. The variables `src` and `dst` represent the distributed array arguments. The inquiries `rng()` and `grp()` extract the range and group objects of these arrays.

Not all the schedules of Adlib are as "pure" as `Remap`. A few, like `WriteHalo` have built-in dependency on the distribution format of the arrays (the existence of ghost regions in the case of `WriteHalo`). But they all rely heavily on the methods and inquiries of the `Range` and `Group` classes, which abstract the distribution format of arrays. The API of these classes has evolved through C++ and Java versions of Adlib over a long period.

In the HPJava version, the lower-level, underlying schedules like *BlockMessSchedule* (which are not dependent on higher-level ideas like distributed ranges and distributed arrays) are in turn implemented on top of a messaging API, called *mpjdev*, described in the section 5.1. To deal with preparation of the data and to perform the actual communication, it uses methods of the mpjdev like `read()`, `write()`, `strGather()`, `strScatter()`, `isend()`, and `irecv()`.

The `write()` and `strGather()` are used for packing the data and `read()` and `strScatter()` are used for unpacking the data where two of those methods (`read()`

```
private void sendLoop(int offset, Group remGrp, int r){

  if(r == rank) {
    sendReq(offset, steps, exts, world.leadId(remGrp));
  } else {

    Block loc = src.rng(r).localBlock();

    int offsetElem = offset + src.str(r) * loc.sub_bas;
    int step       = src.str(r) * loc.sub_stp;

    Range rng = dst.rng(r);
    Triplet crds = rng.crds(loc.glb_lo, loc.glb_hi, loc.glb_stp);

    for (int i = 0, crd = crds.lo; i < crds.count; i++, crd += crds.stp){

      Block rem = rng.block3(crd, loc.glb_lo, loc.glb_hi, loc.glb_stp);

      exts[r]  = rem.count;
      steps[r] = step * rem.glb_stp;

      sendLoop(offsetElem + step * rem.glb_lo,
               remGrp.restrict(rng.dim(), crd),
               r + 1) ;
    }
  }
}
```

Figure 4.3. `sendLoop` method for `Remap`

and `write()`) are dealing with a contiguous data and the other two (`strGather()` and `strScatter()`) are dealing with non-contiguous data. The usage of `strGather()` is to write a section to the buffer from a multi-dimensional, strided patch of the source array. The behaviour of `strScatter()` is opposite of `strGather()`. It reads a section from the buffer into a multi-dimensional, strided patch of the destination array. The `isend()` and `irecv()` are used for actual communication.

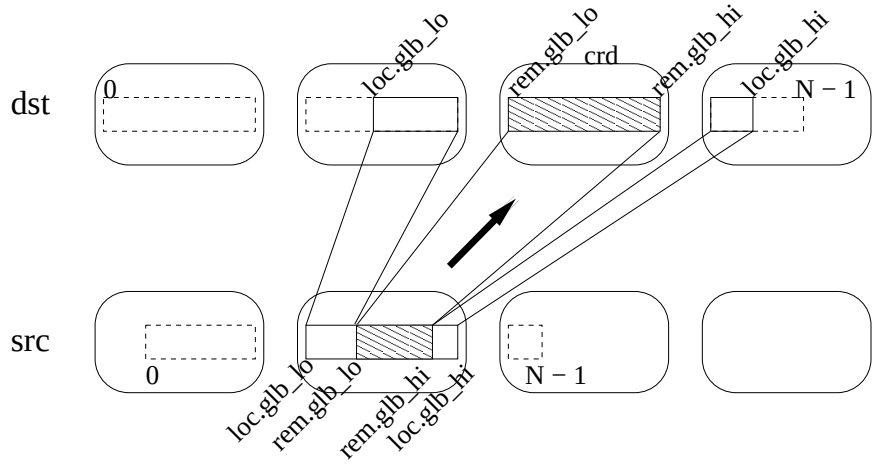

Figure 4.4. Illustration of `sendLoop` operation for `remap`

4.3 Collective Communications

In the previous section we described the Adlib communication implementation issues with a characteristic collective operation example, `remap()`. In this section we will overview functionalities of all collective operations in Adlib. The Adlib has three main families of collective operation: regular communications, reduction operations, and irregular communications. We discuss usage and high-level API overview of Adlib methods. Complete APIs of Adlib are described in Appendix A.

4.3.1 Regular Collective Communications

We already described two characteristic example of the regular communications, `remap()` and `writeHalo()`, in depth. In this section we describe other regular collective communications.

The method `shift()` is a communication schedule for shifting the elements of a distributed array along one of its dimensions, placing the result in another array. In general we have the signatures:

```
void shift(T [[-]] destination, T [[-]] source,
           int shiftAmount)
```

and

```
void shift(T # destination, T # source,
           int shiftAmount, int dimension)
```

where the variable T runs over all primitive types and Object, and the notation T `#` means a multiarray of arbitrary rank, with elements of type T. The first form applies only for one dimensional multiarrays. The second form applies to multiarrays of any rank. The `shiftAmount` argument, which may be negative, specifies the amount and direction of the shift. In the second form the `dimension` argument is in the range $0, \ldots, R-1$ where R is the rank of the arrays: it selects the array dimension in which the shift occurs. The source and destination arrays must have the same shape, and they must also be *identically aligned*. By design, `shift()` implements a simpler pattern of communication than general `remap()`. The alignment relation allows for a more efficient implementation. The library incorporates runtime checks on alignment relations between arguments, where these are required.

The `shift()` operation does not copy values from `source` that would go past the edge of `destination`, and at the other extreme of the range elements of `destination` that are not targetted by elements from `source` are unchanged from their input value. The related operation `cshift()` is essentially identical to `shift()` except that it implements a circular shift, rather than an "edge-off" shift.

Finally we mention the function `broadcast()`, which is actually a simplified form of `remap()`. There are two signatures:

```
T broadcast(T [[]] source)
```

and

```
T broadcast(T source, Group root)
```

The first form takes rank-0 distributed array as argument and broadcasts the element value to all processes of the active process group. Typically it is used with a scalar section to broadcast an element of a general array to all members of the active process group, as here:

```
int [[-,-]] a = new int [[x, y]] ;

int n = 3 + Adlib.broadcast(a [[10, 10]]) ;
```

The second form of `broadcast()` just takes an ordinary Java value as the source. This value should be defined on the process or group of processes identified by `root`. It is broadcast to all members of the active process group.

4.3.2 Reductions

Reduction operations take one or more distributed arrays as input. They combine the elements to produce one or more scalar values, or arrays of lower rank. Adlib provides a large set of reduction operations, supporting the many kinds of reduction available as "intrinsic functions" in Fortran. Here we mention only a few of the simplest reductions. One difference between reduction operations and other collective operations is reduction operations do not support Java `Object` type.

The `maxval()` operation simply returns the maximum of all elements of an array. It has prototypes

t `maxval (`t `# a)`

where t now runs over all Java numeric types—that is, all Java primitive types except `boolean`. The result is broadcast to the active process group, and returned by the function. Other reduction operations with similar interfaces are `minval()`, `sum()` and `product()`. Of these `minval()` is minimum value, `sum()` adds the elements of `a` in an unspecified order, and `product()` multiplies them.

The boolean reductions:

```
boolean any   (boolean # a)
boolean all   (boolean # a)
int     count (boolean # a)
```

behave in a similar way. The method `any()` returns true if any element of `a` is true. The method `all()` returns true if all elements of `a` are true. The method `count()` returns a count of the number of true elements in `a`.

The method `dotProduct()` is also logically a reduction, but it takes two one-dimensional arrays as arguments and returns their scalar product—the sum of pairwise products of elements. The situation with element types is complicated because the types of the two arguments needn't be identical. If they are different, standard Java binary numeric promotions are applied—for example if the dot product of an `int` array with a `float` array is a `float` value. The prototypes are

t_3 `dotProduct(`t_1 `# a,` t_2 `# b)`

and

```
boolean dotProduct(boolean # a, boolean # b)
```

If either of t_1 or t_2 is a floating point type (float or double) the result type, t_3, is double). Otherwise the result type t_3 is long. The argument multiarrays must have the same shape and must be aligned. The result is broadcast to all members of the active process group. The second form takes boolean as the arguments and returns the logical "or" of all the pairwise logical "ands" of elements.

The methods `maxloc()` and `minloc()` return respectively the maximum and minimum values of all elements of an array—similar to `maxval()` and `minval()`—but also output the index tuple in the array at which the extreme value was found. The prototypical forms are:

```
t maxval(int [] loc, t # a)
t minval(int [] loc, t # a)
```

where `loc` is an ordinary Java array of length R, the rank of `a`. On exit it contains the (broadcast) global subscripts of the extreme value.

For each of the simple reductions that combine all elements of an array into a single value, there is a corresponding "dimensional reduction", which just reduces along a selected dimension of the input array, and outputs an array of values of rank one less than the input. The method `maxvalDim()`, for example, has the form:

```
void maxvalDim(t # res, t # a, int dimension)
```

The `dimension` argument takes a value, d, in the range $0, \ldots, R-1$ where R is the rank of `a`. The result array, `res`, must have rank $R-1$. It must be aligned with the input array `a`, *with replicated alignment* in the dth dimension of `a`. In other words, the distribution groups of `a` and `res` must be the same, and

```
res.rng(i).isAligned(a.rng(i))
```

for $i < d$, and

```
res.rng(i).isAligned(a.rng(i + 1))
```

for $d \leq i < R-1$. The reductions `minvalDim()`, `sumDim()`, `productDim()`, `anyDim()`, `allDim()`, `countDim()` are defined in a similar way. The `maxloc()` and `minloc()` reductions have "dimensional" forms:

```
void maxlocDim(t # res, int # loc, t # a, int dimension)
void minlocDim(t # res, int # loc, t # a, int dimension)
```

where the array `loc` has the same rank and alignment as `res` (since the reduction is in a single dimension, only one index value—for the specified dimension—needs to be returned per extreme value). Currently there is no "`Dim`" form of `dotProduct()`.

Finally all the numeric simple reductions and dimensional reductions all have "masked" variants. These take an extra boolean array aligned with the source array. For example

```
t maxval(t # a, boolean # mask)
```

ignores all elements of `a` for which the corresponding element of `mask` is false.

One omission in the current version of the library is a facility for user-defined reduction operations. It also omits various arithmetic reductions that might seem natural in Java, such as bitwise, and, or, and xor. There is no fundamental reason for these omissions, and this might change in future releases.

4.3.3 Irregular Collective Communications

Adlib has some support for irregular communications in the form of collective `gather()` and `scatter()` operations. The simplest form of the gather operation for one-dimensional arrays has prototypes

```
void gather(T [[-]] destination, T [[-]] source, int [[-]] subscripts) ;
```

The `subscripts` array should have the same shape as, and be aligned with, the `destination` array. In pseudocode, the `gather` operation is equivalent to

```
for all i in {0,...,N-1} in parallel do
  destination [i] = source [subscripts [i]] ;
```

where N is the size of the `destination` (and `subscripts`) array. If we are implementing a parallel algorithm that involves a stage like

```
for all i in {0,...,N-1} in parallel do
  a [i] = b [fun(i)] ;
```

where *fun* is an arbitrary function, it can be expressed in HPJava as

```
int [[-]] tmp = new int [[x]] on p ;
on(p)
  overall(i = x for :)
    tmp [i] = fun(i) ;

Adlib.gather(a, b, tmp) ;
```

where `p` and `x` are the distribution group and range of `a`. The source array may have a completely unrelated mapping.

The one-dimensional case generalizes to give a complicated family of prototypes for multidimensional arrays:

```
void gather(T # destination, T [[-]] source,
            int # subscripts)
void gather(T # destination, T [[-,-]] source,
            int # subscripts0, int # subscripts1)
void gather(T # destination, T [[-,-,-]] source,
            int # subscripts0, int # subscripts1, int # subscripts2)
```

Currently the highest rank of source array with a `gather()` method is 3. The source and destination arrays can have different ranks (just as they have unrelated distribution formats). But the destination and subscript arrays are all the same shape, and all are aligned with one another. The *number* of subscript array arguments is equal to the rank of the source array. If the rank of the destination array is R, the general behaviour of these methods is:

```
for all i_0 in {0,...,N_0 - 1} in parallel do
  ...
    for all i_{R-1} in {0,...,N_{R-1} - 1} in parallel do
      destination [i_0, ..., i_{R-1}] = source [subscripts0 [i_0, ..., i_{R-1}],
                                                subscripts1 [i_0, ..., i_{R-1}],
                                                ...] ;
```

where $(N_0, \ldots, N_{R-1})$ is the shape of `destination` array.

The basic `scatter` function has very similar prototypes, but the names `source` and `destination` are switched. For example the one-dimensional case is

```
void scatter(T [[-]] source, T [[-]] destination,
             int [[-]] subscripts) ;
```

```
WriteHalo   writeHalo = new WriteHalo(a) ;
MaxvalFloat maxval    = new MaxvalFloat(r) ;

do {
  writeHalo.execute() ;

  overall(i = x for 1 : N - 2)
    overall(j = y for 1 : N - 2) {
      float newA = 0.25 * (a [i - 1, j] + a [i + 1, j] +
                           a [i, j - 1] + a [i, j + 1]) ;

        r [i, j] = Math.abs(newA - a [i, j]) ;
        b [i, j] = newA ;
      }

  HPutil.copy(a, b) ; // Jacobi relaxation.
} while(maxval.execute() > EPS)) ;
```

Figure 4.5. Jacobi relaxation, re-using communication schedules.

and it behaves like

```
for all i in {0,...,N-1} in parallel do
    destination [subscripts [i]] = source [i] ;
```

Currently the HPJava version of Adlib does not support combining scatters, although these could be added in later releases.

All the `gather()` and `scatter()` operations take an optional final argument which is a boolean array acting as a mask, e.g.

```
void gather(T [[-]] destination, T [[-]] source,
            int [[-]] subscripts, boolean [[-]] mask)
```

The mask should have the same shape as and be aligned with the subscript arrays. Assignment to a destination element is conditional on the value of the element of the mask associated with the subscripts.

4.4 Schedules

The collective communication methods introduced in the last few sections involve two phases: an *inspector* phase in which the arguments are analyzed to determine what communications and local copies will be needed to complete the operation, and an *executor* phase in which the schedule of these data transfers is actually performed. In iterative algorithms, it often happens that exactly the same communication pattern is repeated many times over. In this case it is wasteful to repeat the inspector phase in every iteration, because the data transfer schedule will be the same every time.

Adlib provides a class of *schedule objects* for each of its communication functions. The classes generally have the same names as the static methods, with the first letter capitalized (the name may also be extended with a result type). Each class has a series of constructors with arguments identical to the instances of the function. Every schedule class has one public method with no arguments called `execute`, which executes the schedule.

Using `WriteHalo` and `Maxval` schedules, the main loop of the Jacobi relaxation program from section 3.2, Figure 3.4 could be rewritten as in Figure 4.5.

CHAPTER 5

A LOW-LEVEL COMMUNICATION LIBRARY FOR JAVA HPC

In this chapter we describe the low-level communication library, mpjdev, we introduced for HPJava. This library is developed with HPJava in mind, but it is a standalone library and could be used by other systems. We start this chapter with brief introduction of mpjdev. Detailed information on the mpjdev buffer APIs and communication APIs follows. We will describe different types of buffer operations dealing with both contiguous and non-contiguous data, and various communication methods like blocking and non-blocking communications. A following section discusses how actual data is stored into the message buffer. We also describe issues in four different implementations. An implementation based on native MPI, a pure-Java multithreaded implementation, and platform specific LAPI implementation are already developed. There is also one proposed *Jini* implementation.

5.1 Goals and Requirements

The mpjdev API is designed with the goal that it can be implemented *portably* on network platforms and *efficiently* on parallel hardware. Unlike MPI which is intended for the application developer, mpjdev is meant for library developers. Application level communication libraries like the Java version of Adlib (or MPJ [13]) may be implemented on top of mpjdev. The mpjdev API itself might be implemented on top of Java sockets in a portable network implementation, or—on HPC platforms—through a JNI (Java Native Interface) to a subset of MPI. The positioning of the mpjdev API is illustrated in Figure 5.1. Currently not all the communication stack in this figure is implemented. The Java version of Adlib, the pure Java implementation on SMPs, and native the MPI implementation are

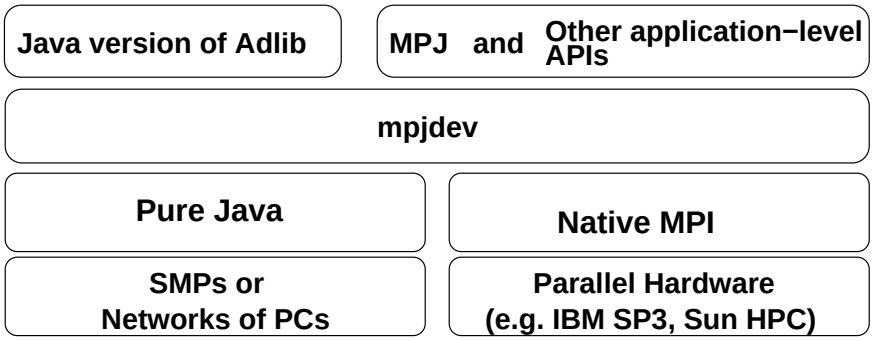

Figure 5.1. An HPJava communication stack.

developed and included in the current HPJava or mpiJava releases. The rest of the stack may be filled in the future. Detailed API information is given in section 5.2.

An important requirement is to support communication of all intrinsic Java types, including primitive types, and objects. It should transfer data between the Java program and the network while keeping the overheads of the Java Native Interface as low as practical. From the development of our earlier successful library mpiJava, we learned communication overheads are key factor of performance. For the mpjdev library, one important decision is made to reduce communication overhead. Usually communication protocols are type specific—different type of data should be sent separately. To avoid many small sends, we maintain all the data of the mpjdev as the Java `byte []` array for pure Java versions or C `char []` array for JNI-based versions. This means all the different primitive types of Java can be stored into the one buffer and sent together instead of using many small separate sends. The Java class types are treated as special case. We can send both primitive types and class types together in one buffer but data may end up in two different messages, one for primitive data and the other for serialized Java objects. To support Java objects efficiently, mpjdev maintains serialized Java objects as a separate Java `byte []` array.

Currently there are three different implementations. The initial version of mpjdev was targeted to HPC platforms, through a JNI interface to a subset of MPI. For SMPs, and for debugging on a single processor, we later implemented a pure-Java, multithreaded version. This version assumes SPMD processes are mapped to Java threads. We also developed a more optimized and system-specific mpjdev built on the IBM SP system using

the Low-level Application Programming Interface (LAPI). This chapter also describes a proposed pure-Java version of mpjdev that uses a Java sockets. This would provide a more portable network implementation of HPJava (without layering on, say, MPICH or LAM).

Our mpjdev layer is similar to *abstract device interface* (ADI) of MPICH. This is used as a lower level communications layer in the MPICH implementation of the MPI. This interface handles just the communication between processes. Message information is divided into two parts: the message *envelope* and the *data*. The message envelope is relatively small and contains message information like tag, communicator, and length.

There are various differences between mpjdev and the ADI. One is that while mpjdev stores message information in the same buffer with the data and send together, the ADI message envelope maintain own buffer and the data of ADI may or may not be delivered at same time. Another is that mpjdev is more suitable to handle different types of data in a message. The ADI does not have particularly good ways to handle different data types in the same buffer.

5.2 General APIs

There are two parts of the mpjdev API. One part deals with communication, and the other part deals with the message buffer.

The currently specified communication API for mpjdev is small compared to MPI. It only includes point-to-point communications. The sophisticated data types of MPI are omitted. This is a fairly major change relative to MPI, but for now it seems hard to make progress in Java while pursuing the HPC ideal of messaging with "zero-copying"—something the derived data types of MPI were designed to facilitate. Avoiding internal copying of message buffers would require changes to the implementation of some of the most popular Java Virtual Machines. This is outside our control.

So in mpjdev we revert to a less demanding scheme in which data is locally copied *at least once*—between the Java program's memory space and a system-managed message-buffer. There is explicit packing and unpacking of buffers—a similar strategy to *new I/O* package of the Sun JDK version 1.4—but we provide a specialized set of gather/scatter buffer operation to better support HPC applications. Much of the complexity in the mpjdev API is then associated with packing and unpacking of message buffers. Because we want to make sure

```
public class Buffer {
  public static final int SECTION_OVERHEAD = 8 ;

  public void ensureCapacity(int newCapacity) { ... }

  public void restoreCapacity() { ... }

  public void free() { ... }
  . . .
}
```

Figure 5.2. The public interface of `Buffer` class.

that usually data need be copied locally *at most once*, we provide a flexible suite of operations for copying data to and from the buffers. These include assorted gather- and scatter-style operations.

5.2.1 Message Buffer API

The message buffer described by class `Buffer` is used for explicit packing and unpacking of messages. The sender creates a communication buffer object of type `Buffer`. Internally this buffer maintains a vector of bytes reflecting the wire format of the message.

This class is a base class for several concrete classes described below. Constructors for those classes specify a fixed initial capacity. The effective public interface of the `Buffer` class itself is given in Figure 5.2. We can increase the buffer capacity by calling `ensureCapacity()`. This method will temporarily increase capacity to `newCapacity` for extra space and will clear previous data from the buffer. The method `restoreCapacity()` is called after to one or more calls to `ensureCapacity()`. It restores the buffer capacity to its original value and frees extra storage that was temporarily allocated. It also clears data from the buffer. The method `free()` is used to free the `Buffer` object. This method is important for implementations of `Buffer` based on native methods. In those implementations, the message vector may be maintained as a dynamically allocated C array that needs an explicit `free()` call to recover its storage. In pure-Java implementations this job will be done by the Java garbage

```
public class WriteBuffer {
  public WriteBuffer(int capacity) { ... }

  public void clear() { ... }

  public void write(t [] source, int srcOff, int numEls) { ... }

  public void gather(t [] source, int numEls, int offs,
                     int [] indexes) { ... }

  public void strGather(t [] source, int srcOff, int rank, int exts,
                        int strs, int [] indexes) { ... }
  . . .
}
```

Figure 5.3. The public interface of `WriteBuffer` class.

collector—calling `free()` is optional. The constant `SECTION_OVERHEAD` defines some extra space needed on each message section. This will be explained in depth in section 5.3.

The `Buffer` class has two subclasses for primitive-type data: `WriteBuffer` for packing and `ReadBuffer` for unpacking. Packing and unpacking of messages that include `Object` types is handled by two special subclasses of `WriteBuffer` and `ReadBuffer`: `ObjectWriteBuffer` and `ObjectReadBuffer`.

The effective public interface of the `WriteBuffer` and `ReadBuffer` classes are represented in Figure 5.3 and Figure 5.4. When constructors `WriteBuffer()` or `ReadBuffer()` create an object they allocate a message vector with size of `capacity`. The `write()`, `gather()`, and `strGather()` are used for packing data. The `read()`, `scatter()`, and `strScatter()` are used for unpacking data. In Figure 5.3, the symbol t runs over all Java primitive types—e.g. there are actually 8 different `write()` methods corresponding to the 8 different primitive types in Java. Each class has three main kinds of method to deal with contiguous and non-contiguous data. Two of those pack and unpack methods (`read()` and `write()`) deal with a contiguous data and the other four (`gather()`, `strGather()`, `scatter()`, and `strScatter()`) deal with non-contiguous data.

```
public class ReadBuffer {
  public ReadBuffer(int capacity) { ... }

  public void reset() { ... }

  public void read(t [] dest, int dstOff, int numEls) { ... }

  public void scatter(t [] dest, int numEls, int offs,
                      int [] indexes) { ... }

  public void strScatter(t [] dest, int dstOff, int rank, int exts,
                         int strs, int [] indexes) { ... }
  . . .
}
```

Figure 5.4. The public interface of `ReadBuffer` class.

In the `WriteBuffer` class, we can make write buffer vector empty using `clear()` method. Any later data is then stored from the beginning of vector. The programmer can re-read a vector from the start by calling method `reset()` in the `ReadBuffer` class.

To support object serialization technology for the Java `Object` type, we need more functionalities. Two special classes, `ObjectWriteBuffer` (Figure 5.5) and `ObjectReadBuffer` (Figure 5.6), have additional pack and unpack methods for `Object` arrays. Notice these classes also inherit methods for primitive types. This allows to store `Object` type with primitive types in the buffer vector. These classes store serialized data into the separate Java `byte []` array. Overridden `clear()` and `free()` methods do some extra work for dealing with the serialized byte array. Both methods flush and close a serialization stream. For `clear()` it will deallocate the stream object. The programmer must call the `flush()` method before sending the message. It prepares the byte array with the serialized Java objects, ready for sending.

```
public class ObjectWriteBuffer extends WriteBuffer {
  public ObjectWriteBuffer(int capacity) { ... }

  public void clear() { ... }

  public void free() { ... }

  public void flush() { ... }

  public void write(Object [] source, int srcOff, int numEls) { ... }

  public void gather(Object [] source, int numEls, int offs,
                     int [] indexes) { ... }

  public void strGather(Object [] source, int srcOff, int rank, int exts,
                        int strs, int [] indexes) { ... }
  . . .
}
```

Figure 5.5. The public interface of `ObjectWriteBuffer` class.

5.2.2 Communication API

In MPI there is a rich set of communication modes. Point-to-point communication and collective communication are two main communication modes of MPI. Point-to-point communication support blocking and non-blocking communication modes. Blocking communication mode includes one blocking mode receive, `MPI_RECV`, and four different send communication modes. Blocking send communication modes include standard mode, `MPI_SEND`, synchronous mode, `MPI_SSEND`, ready mode, `MPI_RSEND`, and buffered mode, `MPI_BSEND`. Non-blocking communication mode also uses one receive, `MPI_IRECV` and the same four modes as blocking sends: standard, `MPI_ISEND`, synchronous, `MPI_ISSEND`, ready, `MPI_IRSEND`, and buffered, `MPI_IBSEND`. Collective communication also includes various communication modes. It has characteristic collective modes like broadcast, `MPI_BCAST`, gather, `MPI_GATHER`, and scatter, `MPI_SCATER`. Global reduction operations are also included in collective communication.

```
public class ObjectReadBuffer extends ReadBuffer{
  public ObjectReadBuffer(int capacity) { ... }

  public void reset() { ... }

  public void free() { ... }

  public void read(Object [] dest, int dstOff, int numEls) { ... }

  public void scatter(Object [] dest, int numEls, int offs,
                      int [] indexes) { ... }

  public void strScatter(Object [] dest, int dstOff, int rank, int exts,
                         int strs, int [] indexes) { ... }
  . . .
}
```

Figure 5.6. The public interface of `ObjectReadBuffer` class.

The mpjdev API is much simpler. It only includes point-to-point communications. Currently the only messaging modes for mpjdev are standard blocking mode (like `MPI_SEND`, `MPI_RECV`) and standard non-blocking mode (like `MPI_ISEND`, `MPI_IRECV`), together with a couple of "wait" primitives.

The communicator class, `Comm`, is very similar to the one in MPI but it has a reduced number of functionalities. It has communication methods like `send()`, `recv()`, `isend()`, and `irecv()`, and defines constants `ANY_SOURCE`, and `ANY_TAG` as static variables.

Figure 5.7 shows the public interface of `Comm` class.

We can get the number of processes that are spanned by this communicator by calling `size()` (similar to `MPI_COMM_SIZE`). Current id of process relative to this communicator is returned by `id()` (similar to `MPI_COMM_RANK`).

The two methods `send()` and `recv()` are blocking communication modes. These two methods block until the communication finishes. The method `send()` sends a message containing the contents of `buf` to the destination described by `dest` and message tag value `tag`.

```
public class Comm {
  public void size() { ... }

  public void id() { ... }

  public void dup() { ... }

  public void create(int [] ids) { ... }

  public void free() { ... }

  public void send(Buffer buf, int dest, int tag) { ... }

  public Status recv(Buffer buf, int src, int tag) { ... }

  public Request isend(Buffer buf, int dest, int tag) { ... }

  public Request irecv(Buffer buf, int dest, int tag) { ... }

  public static String [] init(String[] args) { ... }

  public static void finish() { ... }

  . . .
}
```

Figure 5.7. The public interface of mpjdev `Comm` class.

The method `recv()` receives a message from matching source described by `src` with matching tag value `tag` and copies contents of message to the receive buffer, `buf`. The receiver may use wildcard value `ANY_SOURCE` for `src` and `ANY_TAG` for `tag` instead specifying `src` and `tag` values. These indicate that a receiver accepts any source and/or tag of send. The `Comm` class also has the initial communicator, `WORLD`, like `MPI_COMM_WORLD` in MPI and other utility methods. The capacity of receive buffer must be large enough to accept these contents. Initializes the `source` and `tag` fields of the returned `Status` class which describes a completed communication.

The functionalities of `send()` and `recv()` methods are same as standard mode point-to-point communication of MPI (`MPI_SEND` and `MPI_RECV`). A `recv()` will be blocked until the

send if posted. A `send()` will be blocked until the message have been safely stored away. Internal buffering is not guaranteed in `send()`, and the message may be copied directly into the matching receive buffer. If no `recv()` is posted, `send()` is allowed to block indefinitely, depending on the availability of internal buffering in the implementation. The programmer must allow for this—this is a low-level API for experts.

The other two communication methods `isend()` and `irecv()` are non-blocking versions of `send()` and `recv`. These are equivalent to `MPI_ISEND` and `MPI_IRECV` in MPI. Unlike blocking send, a non-blocking send returns immediately after its call and does not wait for completion. To complete the communication a separate send complete call (like `iwait()` and `iwaitany()` methods in the `Request` class) is needed. A non-blocking receive also work similarly. The `wait()` operations block exactly as for the blocking versions of `send()` and `recv()` (e.g. the `wait()` operation for an `isend()` is allowed to block indefinitely if no matching receive is posted).

The method `dup()` creates a new communicator the spanning the same set of processes, but with a distinct communication context. We can also create a new communicator spanning a selected set of processes selected using the `create()` method. The ids of array `ids` contains a list of ids relative to this communicator. Processes that are outside of the group will get a null result. The new communicator has a distinct communication context.

By calling the `free()` method, we can destroy this communicator (like `MPI_COMM_FREE` in MPI). This method is called usually when this communicator is no longer in use. It frees any resources that used by this communicator.

We should call static `init()` method once before calling any other methods in communicator. This static method initializes mpjdev and makes it ready to use. The static method `finish()` (which is equivalent of `MPI_FINALIZE`) is the last method should be called in mpjdev.

The other important class is `Request` (Figure 5.8). This class is used for non-blocking communications to ensure completion of non-blocking send and receive. We wait for a single non-blocking communication to complete by calling `iwait()` method. This method returns when the operation identified by the current class is complete. The other method `iwaitany()` waits for one non-blocking communication from a set of requests `reqs` to complete. This method returns when one of the operations associated with the active requests in the array

```
public class Request {
  public Status iwait() { ... }

  public Status iwaitany(Request [] reqs) { ... }
  . . .
}
```

Figure 5.8. The public interface of `Request` class.

`reqs` has completed. After completion of `iwait()` or `iwaitany()` call, the `source` and `tag` fields of the returned status object are initialized. One more field, `index`, is initialized for `iwaitway()` method. This field indicates the index of the selected request in the `reqs` array.

5.3 Message Format

This section describes the message format used by mpjdev. The specification here doesn't define how a message vector which contained in the `Buffer` object is stored internally—for example it may be as a Java `byte []` array or it may be as a C `char []` array, accessed through native methods. But this section does define the organization of data in the buffer. For example it includes formula for computing the required *buffer capacity*, as a function of the set of Java data that is to be written to the buffer. It is the responsibility of the user to ensure that sufficient space is available in the buffer to hold the desired message. Trying to write too much data to a buffer causes an exception to be thrown. Likewise, trying to receive a message into a buffer that is too small will cause an exception to be thrown. These features are (arguably) in the spirit of MPI.

A message is divided into two main parts. The *primary payload* is used to store message elements of primitive type. The *secondary payload* is intended to hold the data from object elements in the message (although other uses for the secondary payload are conceivable). The size of the primary payload is limited by the fixed capacity of the buffer, as discussed above. The size of the secondary payload, if it is non-empty, is likely to be determined "dynamically"—for example as objects are written to the buffer.

The message starts with a short *primary header*, defining an *encoding scheme* used in headers and primary payload, and the total number of data bytes in the primary payload.

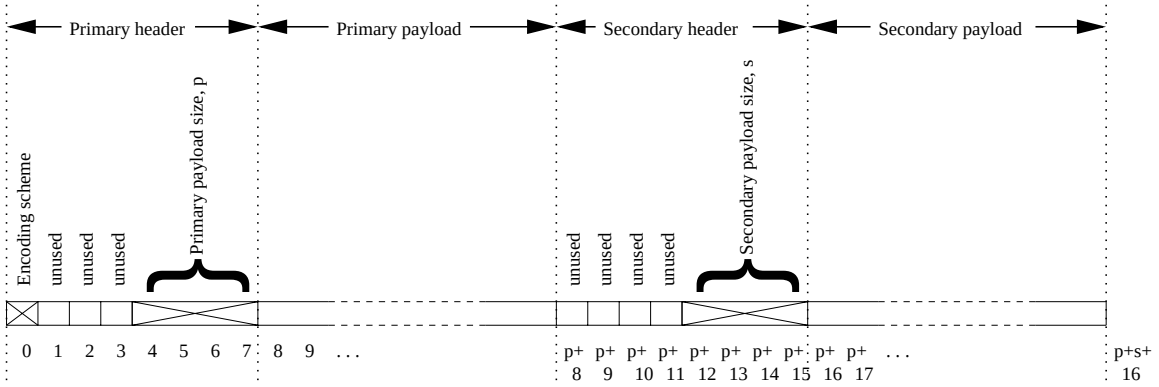

Figure 5.9. Overall layout of logical message.

Only one byte is allocated in the message to describe the encoding scheme: currently the only encoding schemes supported or envisaged are *big-ending* and *little-endian*. This is to allow for native implementations of the buffer operations, which (unlike standard Java read/write operations) may use either byte order.

A message is divided into zero or more *sections*. Each section contains a fixed number of elements of homogeneous type. The elements in a section will all have identical *primitive* Java type, or they will all have `Object` type (in the latter case the exact classes of the objects need *not* be homogeneous within the section).

Each section has a short header in the primary payload, specifying the type of the elements, and the number of elements in the section. For sections with primitive type, the header is followed by the actual data. For sections with object type, the header is the only representation of the section appearing in the primary payload—the actual data will go in the secondary payload.

After the primary payload there is a *secondary header*. The secondary header defines the number of bytes of data in the secondary payload.

The secondary header is followed in the logical message by the secondary payload. This section says nothing about the *layout* of the secondary payload. In practice this layout will be determined by the Java Object Serialization specification.

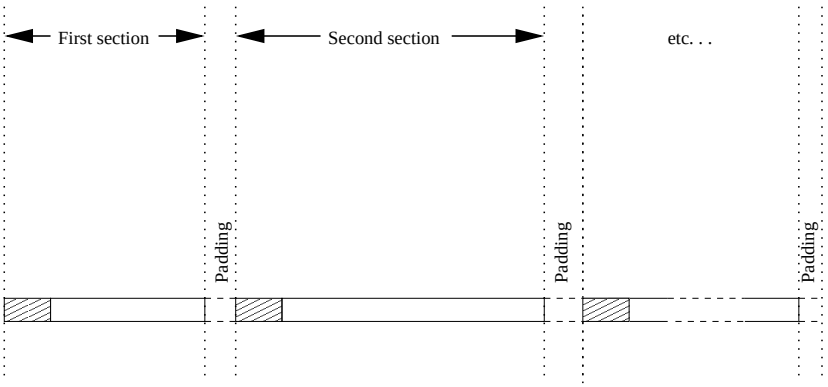

Figure 5.10. Layout of primary payload

5.3.1 Layout details

Further details of the message layout are schematized in Figures 5.9 through 5.11. Figure 5.9 shows the overall layout of the logical message. As explained above, there are four parts: a primary header, a primary payload, a secondary header, and a secondary payload. The figure also shows the layout of the headers.

The first byte of the primary header specifies the encoding of numeric types used throughout the two main headers and the primary payload. Currently there are two possible values for this byte: 0 indicates big-endian, 1 indicates little-endian.

The next three bytes are unused. The second 4-byte word of the the header contains the total length of the primary payload in bytes (because of the way sections are padded, this will always be a multiple of 8). Like all numbers in the parts of the message discussed here, this word is encoded according to the scheme selected by the first byte of the header.

The format of the primary payload will be described below. The secondary header only contains the length of the secondary payload in bytes. For uniformity with the primary header, this information is stored in the second 4-byte word. The first 4 bytes of the header are unused.

Note that all basic units of the message start on 8-byte boundaries. This is to avoid possible word alignment problems on some computer architectures, if the buffer operations are implemented natively.

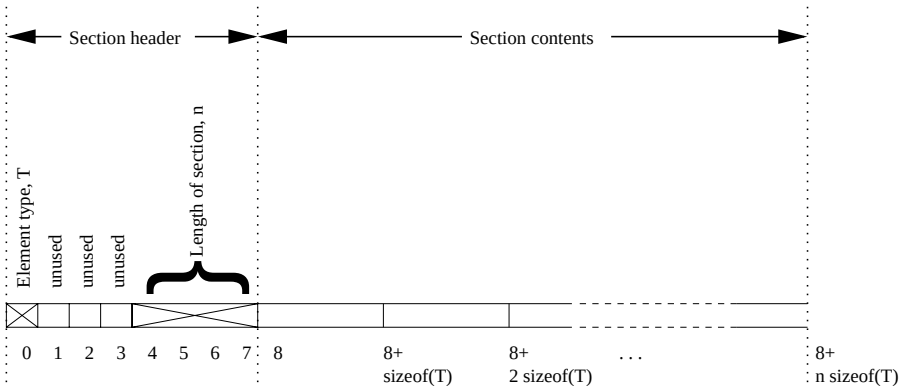

Figure 5.11. Layout of one section in the primary payload

Now we discuss the format of the primary payload. Figure 5.10 shows that it is divided into a series of zero or more sections. These sections will be separated by padding, if necessary, to ensure that the sections all start on 8-byte boundaries. If the size of a section in bytes is b, the amount of padding following the section is $8\lceil b/8 \rceil - b$ bytes.

Figure 5.11 shows the layout of an individual section. The first byte defines the type of the elements held in the section according to the scheme:

value	Java type, T	sizeof(T)
0	byte	1
1	char	2
2	short	2
3	boolean	1
4	int	4
5	long	8
6	float	4
7	double	8
8	Object	0

The space occupied by an individual element in the section is defined in terms of the function `sizeof()`. This is defined to return 0 for object types, reflecting that the data content of objects is not stored in the primary payload.

The empty message consists of 16 bytes—the primary header defining an encoding in its first byte, and containing zero in its second word, and the secondary header containing zero in its second word.

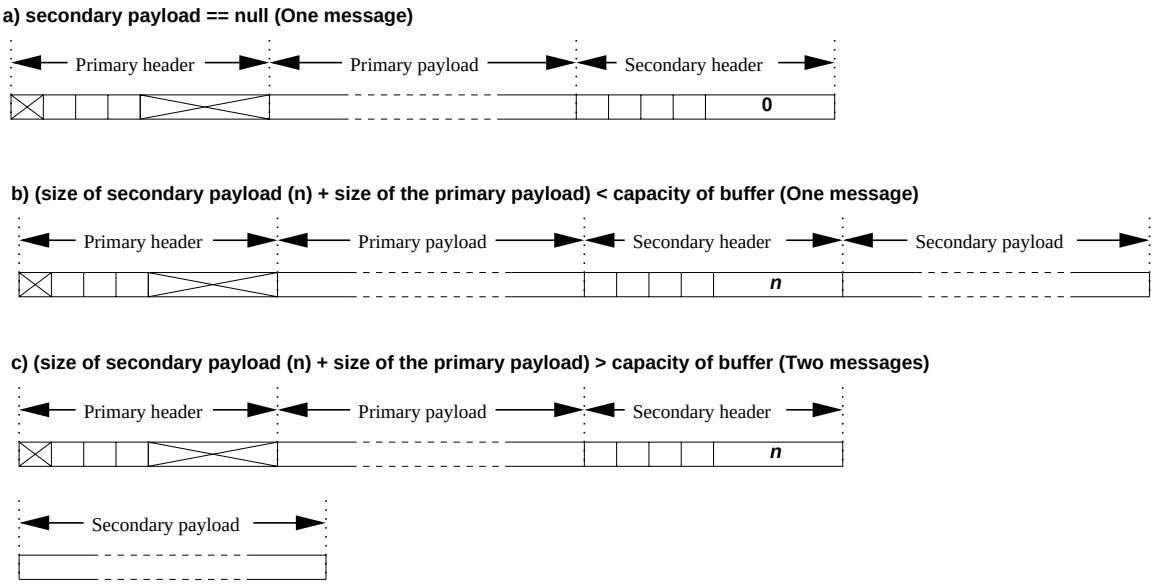

Figure 5.12. Layout of send messages. a) When secondary payload is empty. b) When secondary payload is not empty and the sum of secondary payload size and the size of the primary payload are less than the capacity of the buffer object. c) When secondary payload is not empty and the sum of secondary payload size and the size of the primary payload are greater than the capacity of the buffer object.

The smallest contentful message, containing one primitive element, consists of 32 bytes: the primary header defining an encoding and specifying a size of 16 for the primary payload; an 8-byte section header specifying an element type and a length of 1; the section contents—one primitive element padded out if necessary to 8 bytes; and the secondary header specifying an empty secondary payload.

For larger messages the use of buffer space soon becomes more economical. We assume that the overheads in buffer size for short messages will be hidden by other message overheads, including the processing cost for message startup, and network latency.

5.4 Communication Protocol

This section discusses a protocol for sending a logical message on top of message-oriented transport like MPI, and discusses the physical messages that are sent. On top of a stream-oriented transport like TCP, some of these considerations may be less relevant. Figure 5.12 illustrates layout of possible send messages.

When a buffer object is created, a capacity is defined for that buffer. This capacity is the available space for the primary payload buffer (in practise an extra 16 bytes will be allocated to allow for the primary header and the secondary header, but the user need not be concerned with this).

Through write operations on the buffer, the user will initialize the primary payload. These operations may also define a non-empty secondary payload, presumably held in separate storage.

When the time to send the message comes, the primary and secondary headers are written to the buffer. If the secondary payload is empty, a single message is sent containing the primary payload along with the primary header and secondary header, the latter specifying a size of zero.

In this case (when it is known in advance that the secondary payload will be empty—for example because only primitive elements are expected) it is required that the receiver accepts the message into a buffer object with a capacity equal to *at least* the *actual size of the primary payload*.

If the secondary payload *is not empty*, it may either be concatenated to the same message, or sent as a second message (in which case the first message still contains the primary header, the primary buffer and the secondary header). If there are two messages, the receiver allocates an array large enough to hold the secondary payload after the first message (containing the secondary header) has been received.

If the secondary payload is not empty, but the sum of its size and the size of the primary payload are less than the capacity of the buffer object at the sending end, the two may be concatenated together in a single message.

This implies that *if it is possible that the secondary payload may not be empty*, the receiver should accept the message into a buffer object with a capacity equal to *at least* the *capacity of the buffer object* used to send the message. Otherwise a concatenated message might overflow the receive buffer. Note this is a stronger requirement than in the case where all message elements are primitive (and there is no secondary payload).

Typical message-based transports allow the receiver to determine the actual number of bytes received in a message (in MPI, through the `MPI_Status` object). Moreover, by examining the primary and secondary header contained in the received message, the receiver

can compute the expected size of the complete, logical message. If this is greater than the number of bytes in the physical message, a second message must be expected.

The simplest safe policy is probably to try always to accept a message into a buffer object with *identical capacity* to the buffer used to send the message. This policy can be relaxed if the message is known only to involve primitive elements, but care should be taken.

5.4.1 Encoding

A "reader-makes-right" strategy (like CORBA IIOP, for example) is adopted to handle encoding. The writer outputs a message using the endianess most natural to its local processor or software platform. The reader must be prepared to accept either encoding, and reverse byte orders if the encoding in the message does not agree with its local native ordering. If the source and destination have same endianess (which is common), no reversal in necessary, whatever the endianess actually is.

5.5 Implementations

Currently we have three different implementations of mpjdev: mpiJava-based, multithreaded, and LAPI-based. We have also proposed an implementation based on Jini. These implementation have their own characteristic signatures. The mpiJava-based implementation uses MPI as communication protocol via JNI calls. This implementation is platform dependent due to use of JNI. Multithreaded implementation assumes HPspmd "processes" are implemented as Java threads, and uses Java thread synchronization mechanisms for communication. This provides maximum portability by using pure Java. LAPI implementation is specific to one system, the IBM SP. Design goals of the proposed Jini implementation are that the system should be as easy to install on distributed systems as we can reasonably make it, and that it be sufficiently robust to be usable in an Internet environment. It could use Java sockets for underlying communication.

5.5.1 mpiJava-based Implementation

The mpiJava-based implementation assumes C binding of native method calls to MPI from mpiJava (see section 3.4.1) as basic communication protocol. This implementation

can be divided into two parts: Java APIs (`Buffer` and `Comm` classes) which are described in the previous sections and C native methods that construct the message vector and perform communication. The Java-side methods of communicator class `Comm` and message buffer class `Buffer` are used to call native methods via JNI. The C stubs that bind the mpjdev communication class to the underlying native MPI implementation are created using JNI, which Java can call and thus pass parameters to and from a native API.

To optimize the performance of this version, we maintain the message buffer inside the C code. Instances of a C `struct` type (lightweight object) `Buffer` (differnt from the Java side Buffer class) is used for maintain message vector. This lightweight object stores a pointer to the message vectors and other information that is used for operation on the message vector: information like original capacity, current capacity, and current write and read position of vector.

Message elements of all data types other than `Object` are stored as C `char []` array. This architectural decision means actual communication takes place only with `MPI_BYTE` data type. Before sending, we extract `char []` array from the C object and store the total number of data bytes of the array into the Primary header (see section 5.3). This size information is used to make sure capacity of receive side vector is large enough to hold incoming data. For elements of `Object` type, the serialized data are stored into a Java `byte []` array. We can send this array by copying into the existing message vector if it has space to hold serialized data array, or by using separate send if the original message vector is not large enough.

In latter case there will be two different sends from the sender side. The receiver side will have to expect a second message.

5.5.2 Multithreaded Implementation

In this implementation, the processes of an HPJava program are mapped to the Java threads of a single JVM. This allows to debug and demonstrate HPJava programs without facing the ordeal of installing MPI or running on a network. A single JVM is used to debug programs before launching them on a network or parallel computer. If an HPJava program is written for execution on distributed memory parallel computers then it may be possible to run the program in this implementation and have behave the same way. As a by-product, it also means we can run HPJava programs on *shared memory parallel computers*. These kinds

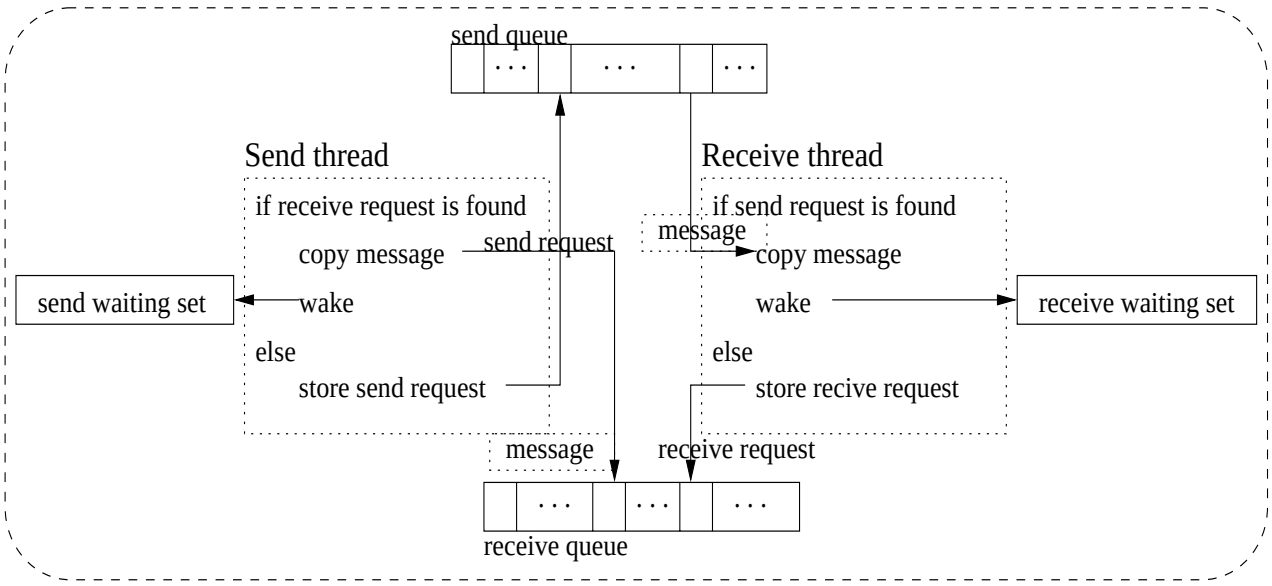

Figure 5.13. Multithreaded implementation.

of machines are quite widely available today—sold, for example, as high-end UNIX servers. Because the Java threads of modern JVMs are usually executed in parallel on this kind of machine, it is possible to get quit reasonable parallel speedup running HPJava programs in the multithreaded mode.

Figure 5.13 illustrates multithreaded implementation. In this implementation, communications are involved between Java threads. The set of all threads is stored as an array. Each index in this array represents node id.

Two different static queues send and receive queue are maintained to store early arrival of send and receive requests. Each thread also maintains a wait set in the `Request` class. Communication of any thread that is stroed in this set will be blocked untile complete transaction. If tasks of a non-blocking send or receive are not completed by the time to call completion method of non-blocking communication, like `iwait()` or `iwaitany()`, that particular send or receive is stored into the wait set and is blocked for its completion.

Current version of `send()` and `recv()` methods are implemented using `isend()` and `irecv()` with `iwait()` method call. When a send request is created by the send thread, it looks for a matching receive request in the receive queue. If a matching receive request is exist, it copies data from the send buffer to the receive request buffer. It also checks if

any other thread is iwait-ing on "matching receive" and removes all requests from wait set, and signal the waiting thread. This signal makes the waiting thread awake and continue its operation. The send request will be added into the send queue if it fails to find matching receive queue. A receive request work similarly as the send request. The receive request searches the send queue instead of receive queue for a matching request.

5.5.3 LAPI Implementation

The Low-level Application Programming Interface (LAPI) is a low level communication interface for the IBM Scalable Powerparallel (SP) supercomputer Switch. This switch provides scalable high performance communication between SP nodes. The LAPI is a non-standard application programming interface and provides efficient one-side communication performance between tasks on IBM SP system. LAPI offers better message passing performance than MPI on small and medium size messages. However users must write many extra lines of low-level communication calls on their applications. The target group of LAPI is power programmers who need performance improvement more than code portability.

LAPI functions can be divided into three different characteristic groups. The first of these is a basic *active message* infrastructure that allows programmers to write and install their own set of handlers that are invoked and executed in a target process on behalf of the process originating the active message. The active messages play important role in implementation of our underlying communication. In addition to the active message, the LAPI provides a Remote Memory Copy (RMC) interface—including a `PUT` operation that copies data from the address space of the origin process into the address space of the target process and a `GET` operation which is opposite of the `PUT` operation. LAPI also provides a set of control functions for the initialization and termination of the LAPI layer.

The active message function (`LAPI_Amsend`) is a basic programming mechanism that provides a one-sided communications model. It is a non-blocking call on the origin process that causes a specified active message handler to be invoked and executed in the address space of the target process. The basic communication model of the active message function is given in Figure 5.14. Different steps of communication functions are involved within an active message call. Each active message includes the address of a user-specified handler and it may optionally bring a user header and data from the origination process (step 1). Once

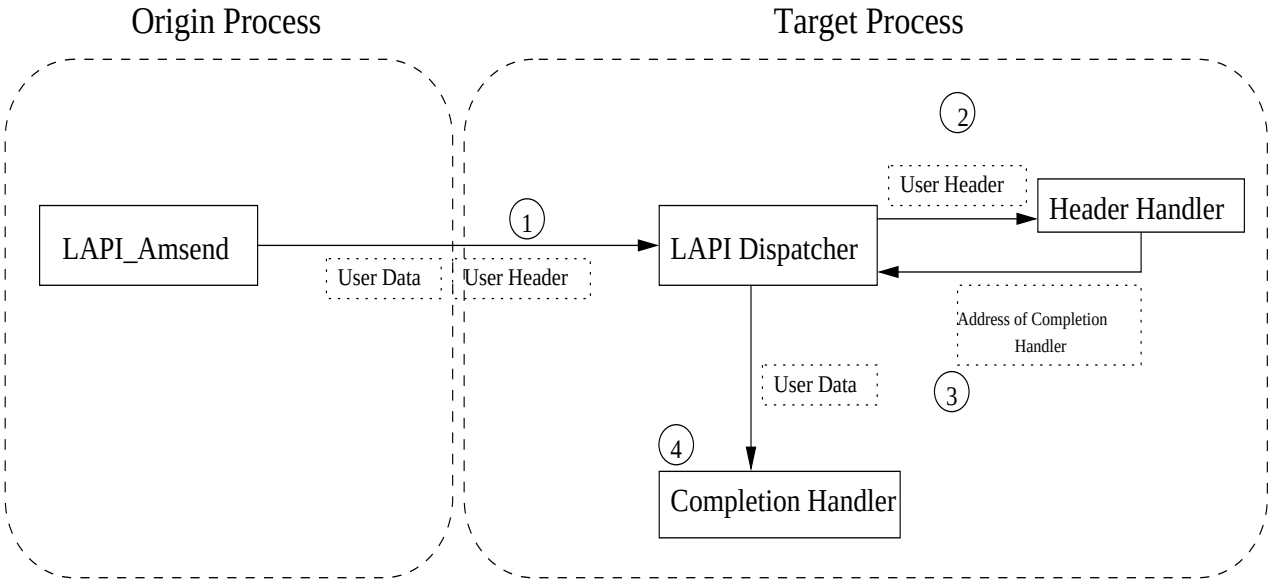

Figure 5.14. LAPI Active Message Call.

the first packet of the message arrives at the target process. The LAPI dispatcher, which is the part of the LAPI that deals with the arrival of message and invocation of handlers, invokes a *header handler* (step 2). A header handler is responsible for the address of where to copy the arriving data, the address of the optional *completion handler*, and a pointer to the parameter that is to passed to the completion hander (step 3). The completion handler is executed after the last packet of the message has been received and copied into a buffer (step 4).

To ensure reusability of buffer on completion of data transfer, the active message uses three counters. Two counters (*origin counter* and *completion counter*) are located in the address space of sending task and one counter (*target counter*) is located in the target task's address space. The origin counter is incremented when it is safe for the origin task to update the origin buffer. After the message has arrived at the target task, the target counter is incremented. Completion of the message transfer updates the completion counter. We can simulate blocking communication by waiting for the completion counter to increment.

5.5.3.1 Implementation issues

We produced two different implementations of mpjdev using LAPI. The first one uses the active message function (`LAPI_Amsend`) togather with a `GET` operation (`LAPI_Get`). It will send only message information accessery to identify the matching send on the receive side with `LAPI_Amsend`. After a matching send is identified by the receiver, the receiver will get actual messages with `LAPI_Get`. The advantage of this is that we can reduce number of message copies. But this uses an extra communication to get messages.

The second implement uses only `LAPI_Amsend` as communication. It will send actual messages along with message information. In this implementation the advantage is that we can reduce an extra communication to get message. But this increase number of message copies.

There are some common design issues in both implementations. Both implementations store and manage message buffer in C, like in mpiJava-based implementation. They both use Java thread synchronization to implement waiting in the MPI. Both use two static objects—"send queue" and "receive queue"—to maintain early arrived send and receive requests.

Figure 5.15 illustrates LAPI implementation with `LAPI_Amsend` and `LAPI_Get` communication calls. When source process receives a send request, it issues active message to target process. This active message contains message information like length, source and destination id, tag, and address of actual messages. Those information are used to identify matching send by the target process. Actual buffer data will remain with the sender until the target process gets the messages. This means sender thread must block until completion of data send when completion method (`iwait()` or `iwaitany()`) is called.

After the initial active message arrives at the target process, it calls the completion handler. In this handler, all the active message information are extracted and passed to the JVM by calling Java static method from JNI. In this static method, the posted receive queue is searched to see if receive has already been posted with matching the message description. If a matching receive is found, target issues `GET` operation to get actual messages from the source. To complete the transaction, the target must notify to source to wake any waiting thread. This is done by a second active message call to the source. It also wakes any user

thread that is waiting for this receive by issuing a local notify signal. If there is no matching receive, it will store all the information into the send queue for later use.

A receive request on the target process behaves similarly to the target side of a sending active message call. The difference is it searches send queue instead of receive queue. And it stores to the receive queue when a matching send is not found.

Figure 5.16 illustrates LAPI implementation with single `LAPI_Amsend` communication call. Architecture of this implementation is simpler than the previous. Because messages are sent out when active message call is made, the source process does not have to wait for completion of communication. This decision eliminate call backs to the source from the target. The target does not have to perform any `GET` operation any more. However, this implementation must perform extra message copies that do not exist in the previous implementation. Whenever we see a transaction of a message from the Figure 5.16, whole messages are copied into different storage.

Even though the simpler implementation has extra message copy operations, it is faster with our problem sizes (up to float array size of 1024 x 1024). We will see in section 6.4 that our LAPI implementation was not faster than the SP MPI implementation. We believe this was due to reliance on Java-side thread synchronization, which appears to be slow. We believe that this problem could be overcome by doing thread synchonization on the C side using POSIX mechanisms, but didn't have time to test this.

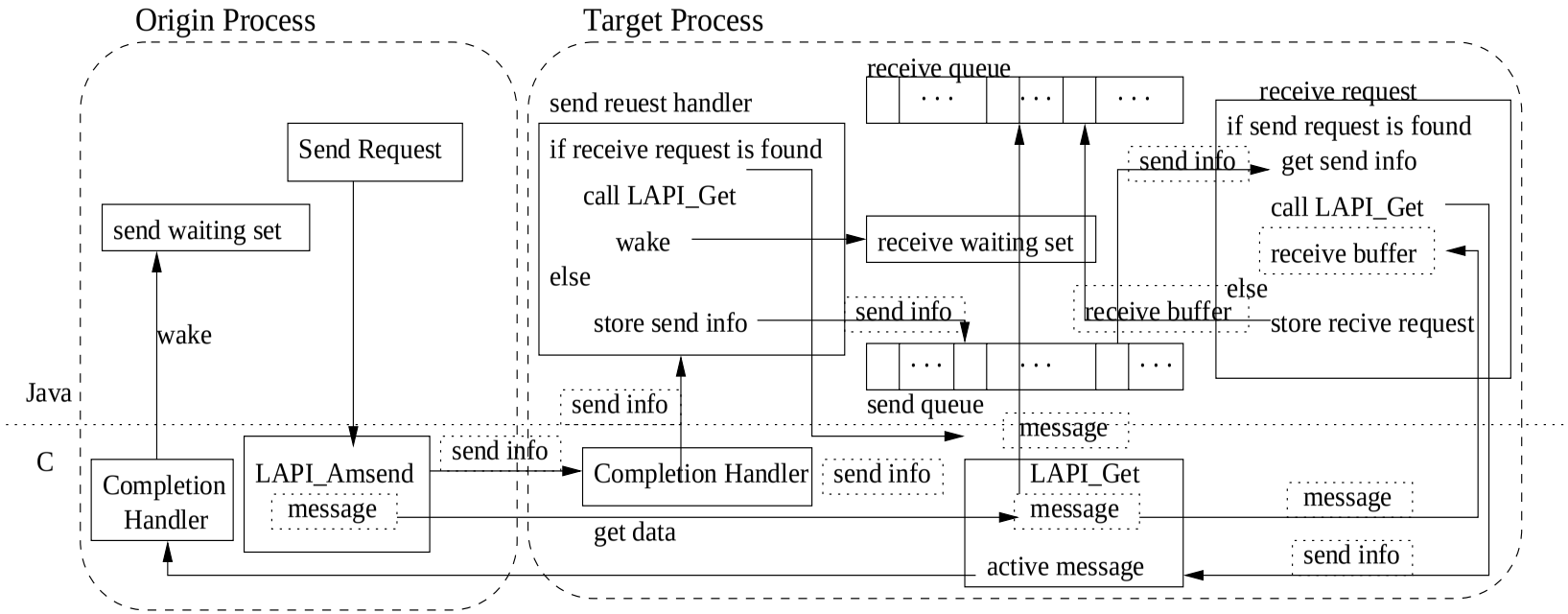

Figure 5.15. LAPI implementation with Active Message Call and `GET` operation.

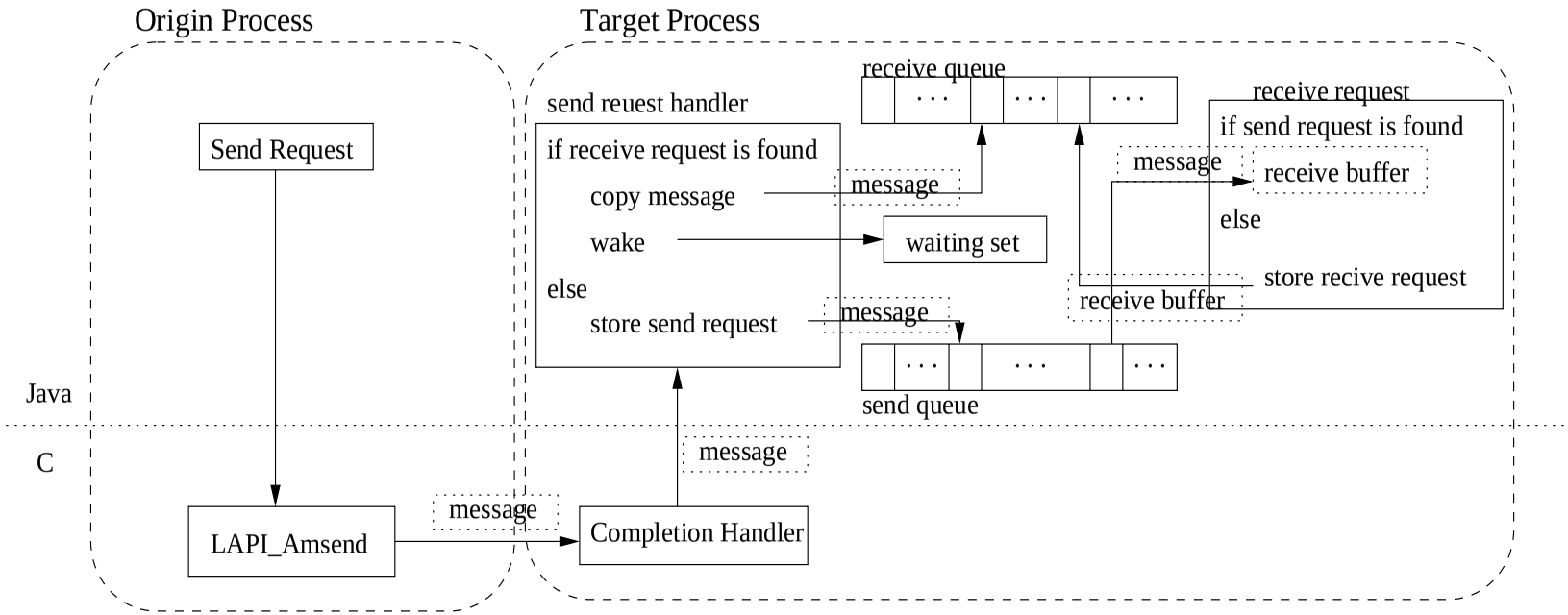

Figure 5.16. LAPI implementation with Active Message Call.

5.5.4 Jini Implementation

Current distributed implementations of our mpjdev rely on the availability of a platform-specific native underlying communication interfaces, like MPI and LAPI, for the target computer (although we have the pure Java multithreaded implementation, this implementation is not useful for distributed programming). While this is a reasonable basis in many cases, the approach has some disadvantages. For one thing the two-stage installation procedure—get and build a native underlying communication interface then install and match Java wrappers—can be tedious and discouraging to potential users. The situation has improved, and mpjdev now runs with several combinations of JVM and MPI implementation, but portability is still a problem.

The authors of [9] have outlined a design for a *pure-Java* reference implementation for MPJ (see section 2.1.2). Design goals were that the system should be as easy to install on distributed systems as one can reasonably make it, and that it be sufficiently robust to be usable in an Internet environment. This proposed design can be naturally adapted to the mpjdev library. A system like this may be an important component in a future HPJava system.

The paper referenced above suggests that Jini may be a natural foundation for meeting the requirements. Jini is Sun's Java architecture for making services available over a network. It is built on top of the Java Remote Method Invocation (RMI) mechanism. The main additional features are a set of protocols and basic services for "spontaneous" discovery of new services, and a framework for detecting and handling *partial failures* in the distributed environment.

A Jini lookup service is typically discovered through multicast on a well-known port. The discovered registry is a unified first point of contact for all kinds of device, service, and client on the network. This model of discovery and lookup is somewhat distinct from the more global concept of discovery in, say, the CORBA trading services, HP's e-speak [32], or JXTA. The Jini version is a lightweight protocol, especially suitable for initial binding of clients and services within multicast range.

The ideas of Jini run deeper than the lookup services. Jini completes a vision of *distributed programming* started by RMI. In this vision *partial failure* is a defining characteristic, distin-

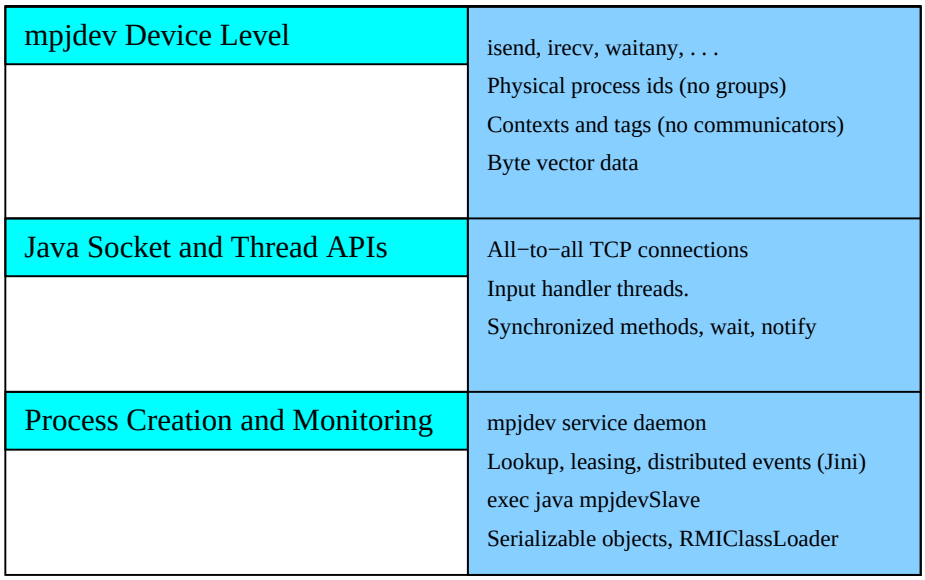

Figure 5.17. Layers of a proposed mpjdev reference implementation

guishing distributed programming from the textbook discipline of concurrent programming [53]. So in Jini remote objects and RMI replace ordinary Java objects and methods; garbage collection for recovery of memory is replaced by a *leasing* model for recovery of distributed resources; the events of, say, AWT or JavaBeans are replaced by the distributed events of Jini. Finally, the synchronized methods of Java are mirrored in the nested transactions of the Jini model. Concurrent programming exactly identical to scalable parallel programming, but we need analogous sets of abstractions for the parallel case.

The installation script can start a daemon on the local machine by registering a persistent activatable object with the `rmid` daemon. The mpjdev daemons automatically advertise their presence through the Jini lookup services. The Jini paradigms of leasing and distributed events are used to detect failures and reclaim resources in the event of failure. These observations lead us to believe that a pure Java distributed reference implementation of mpjdev should probably use Jini technology [7, 24] to facilitate location of remote mpjdev daemons and to provide a framework for the required fault-tolerance.

A possible architecture is sketched in Figure 5.17. The base layer—process creation and monitoring—incorporates initial negotiation with the mpjdev daemon, and low-level services

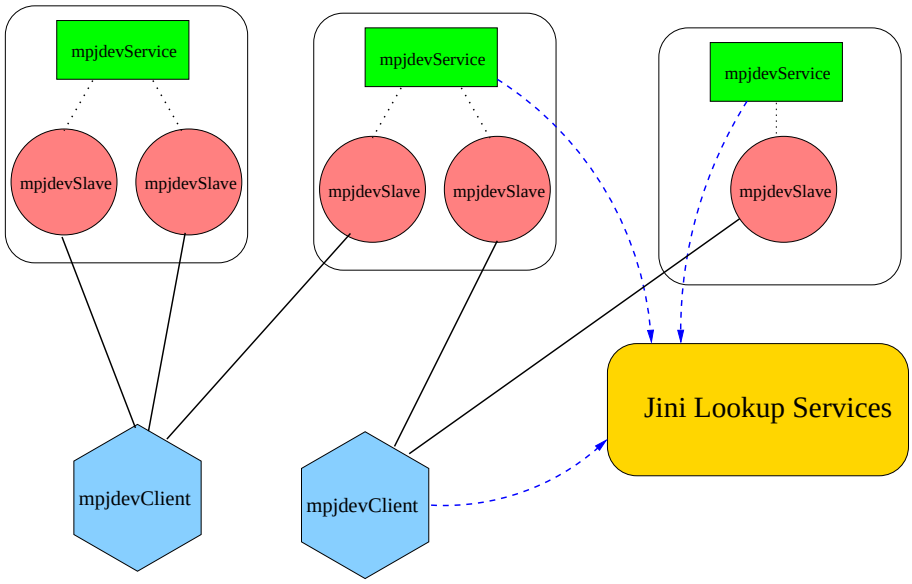

Figure 5.18. Independent clients may find `mpjdevService` daemons through the Jini lookup service. Each daemon may spawn several slaves.

provided by this daemon, including clean termination and routing of output streams (Figure 5.18).

One emphasis for the future work will be on researching links between parallel message-passing programming and Jini-like systems. Moreover we will also investigate newer ideas coming from projects like JXTA.

CHAPTER 6

APPLICATIONS AND PERFORMANCE

In this chapter we illustrate usage of HPJava with some applications. Performance and usage of three different implementations of mpjdev are discussed with applications. We start by describing a full multigrid solver application. Benchmark results of simple kernel applications that use MPI-based mpjdev follow. We present a graphical simulation of 2D inviscid flow through an axisymmetric nozzle with GUI in Section 6.3. This example illustrates usage of the multithreaded implementation of mpjdev. LAPI implementation of mpjdev communication library is also evaluated. Finally, we compare our mpiJava based Adlib communication library with C/MPI version to find most time consuming part of communication. This data can be used for further optimization.

6.1 Environments

We have implemented several HPspmd style parallel applications using HPJava. These applications are included in the HPJava software package (available from `www.hpjava.org`). In this chapter we consider a full application of HPJava, and also evaluate the performance of some of simpler kernel applications written in HPJava like red-black relaxation and diffusion equation.

For each application, we have developed both sequential and parallel programs to compare the performance. The sequential programs were written in Fortran 95 and Java and parallel programs were written in HPF and HPJava. For better performance, all sequential and parallel Fortran and Java codes were compiled using -O5 or -O3 with -qhot or -O (i.e. maximum optimization) flag.

The system environment for SP3 runs were as follows:

- System: IBM SP3 supercomputing system with AIX 4.3.3 operating system and 42 nodes.
- CPU: A node has Four processors (Power3 375 MHz) and 2 gigabytes of shared memory.
- Network MPI Settings: Shared "css0" adapter with User Space(US) communication mode.
- Java VM: IBM 's JIT
- Java Compiler: IBM J2RE 1.3.1

6.2 Partial Differential Equations

Partial differential equations (PDE's) are one of the most fundamental applications of mathematics. They describe phenomena of the physical, natural and social science such as fluids, gravitational and electromagnetic fields, and the human body. They also play an important role in fields like aircraft simulation, computer graphics, and weather prediction.

There exist many partial differential equations, but from the view point of mathematics, three important examples are Laplace's equation, diffusion equation, and wave equation. The general linear 2 dimensional equation of the PDE's is can be written as follows

$$a\frac{\partial^2 V}{\partial x^2} + 2b\frac{\partial^2 V}{\partial x \partial y} + c\frac{\partial^2 V}{\partial y^2} + d\frac{\partial V}{\partial x} + e\frac{\partial V}{\partial y} + fV + g = 0 \tag{6.1}$$

where it is Laplace equation if $b^2 < ac$, Wave equation if $b^2 > ac$, and Diffusion/Schrödinger equation if $b^2 = ac$.

In this paper we will perform benchmarks on the Laplace's equation and diffusion equation.

Elliptic equation like Laplace equation is commonly found in multidimensional steady state problems. We can write the simplest form of elliptic equation (Poisson equation) by simplifying the equation (6.1) as follows

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} = \rho_{x,y} \tag{6.2}$$

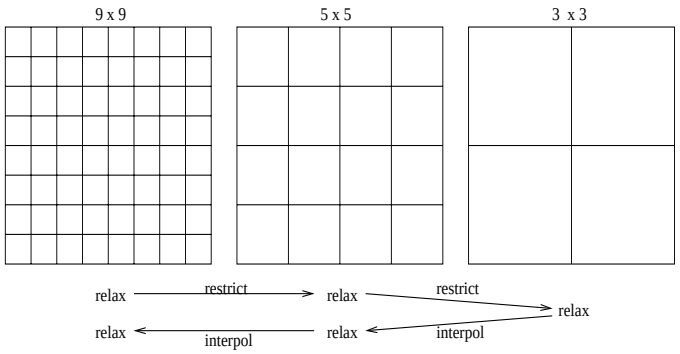

Figure 6.1. An example of multigrid iteration.

This equation can also be rewritten in finite-difference form using the function $u(x, y)$ and its representation values at the discrete set of points

$$x_j = x_0 + j\Delta \quad and \quad y_l = y_0 + l\Delta \tag{6.3}$$

with the indices $j = 0, 1, 2, ..., J$ and $l = 0, 1, 2, ..., L$. Finite-difference representation (equation (6.4)) of equation 6.2 and its equivalent format (equation (6.5)) which can be written as a system of linear equations in matrix form can be described as follows

$$\frac{u_{j+1,l} - 2u_{j,l} + u_{j-1,l}}{\Delta^2} + \frac{u_{j,l+1} - 2u_{j,l} + u_{j,l-1}}{\Delta^2} = \rho_{j,l} \tag{6.4}$$

$$u_{j+1,l} + u_{j-1,l} + u_{j,l+1} + u_{j,l-1} - 4u_{j,l} = \Delta^2 \rho_{j,l} \tag{6.5}$$

6.2.1 An Application

In this section we will discuss a full application of HPJava—a multigrid solver. The particular solver was adapted from an existing Fortran program (called PDE2), taken from the Genesis parallel benchmark suite [5]. The whole of this program has been ported to HPJava (it is about 800 lines), but in this section we will only consider a few critical routines.

The *Multigrid* [10] method is a fast algorithm for solution of linear and nonlinear problems using *restrict* and *interpolate* operations between current grids (*fine grid*) and restricted grids (*coarse grid*). As applied to basic relaxation methods for PDEs, it hugely accelerates elimination of the residual by restricting a smoothed version of the error term to a coarse grid, computing a correction term on the coarse grid, then interpolating this correction

```
static void relax(int itmax, int npf, double[[-,-]] uf, double[[-,-]] ff) {

  Range xf = ff.rng(0), yf = ff.rng(1);

  for(int it = 1; it <= itmax * 2; it++) {
    Adlib.writeHalo(uf);

    overall(i = xf for 1 : npf - 2)
      overall(j = yf for 1 + (i‘ + it) % 2 : npf - 2 : 2) {

        uf [i, j] = 0.25 * (ff [i, j] + uf [i - 1, j] + uf [i + 1, j] +
                                        uf [i, j - 1] + uf [i, j + 1]);
     }
  }
}
```

Figure 6.2. Red black relaxation on array uf.

back to the original fine grid where it is used to improve the original approximation to the solution. Multigrid methods can be developed as a *V-cycle* method for simple linear iterative methods. As we can see in Figure 6.1, there are three characteristic phases in a V-cycle method; *pre-relaxation*, *multigrid*, and *post-relaxation*. The pre-relaxation phase makes the error smooth by performing a relaxation method. The multigrid phase restricts the current problem to a subset of the grid points and solves a restricted problem for the correction term. The post-relaxation phase perform some steps of the relaxation method again after interpolating results back to the original grid.

As an example we take red-black relaxation for the Laplace equation as our relaxation method. The *relax* operation, the *restrict* operation, and the *interpolate* operation are three critical parts of a solution of 2D Poisson equation using a multigrid method. Domain decomposition, which assigns part of the grid to each processor, is used for parallelization. Boundary values of the subgrids are exchanged with nearest neighbors before each calculation. Red-black relaxation on array `uf` is illustrated in Figure 6.2. In this red-black relaxation, the values of the subgrids boundary are exchanged by using the `writeHalo` method. We assume that all distributed arrays in our examples were created with ghost regions.

```
static void restr(int           npc, int          npf, double[[-,-]] fc,
                  double[[-,-]] uf , double[[-,-]] ff ) {

  // uf, ff are input values on fine grid, fc is output.
  Range xf = ff.rng(0), ff = ff.rng(1);

  float [[-,-]] tf = new float [[xf, yf]] ;

  int nc = npc - 1, nf = npf - 1;

  Adlib.writeHalo(uf);

  overall(i = xf for 2 : nf - 2 : 2)
    overall(j = yf for 2 : nf - 2 : 2)
      tf [i, j] += 2.0 * (ff [i, j] - 4.0 * uf [i, j] +
                          uf [i - 1, j] + uf [i + 1, j] +
                          uf [i, j - 1] + uf [i, j + 1]);

  Adlib.remap(fc [[1 : nc - 1, 1 : nc - 1]],
              tf [[2 : nf - 2 : 2, 2 : nf - 2 : 2]]);
}
```

Figure 6.3. Restrict operation.

The implementation makes use of the integer global loop index `i‘` (read "i-primed") associated with distributed index `i`. This value is used in computing the lower bound of the inner overall. The modulo 2 operation including `i‘` and `it`, in conjunction with the loop stride of 2, ensures that sites of a single parity are updated in a given iteration `it`, and that this parity is swapped in successive iterations. This is a short way to write the algorithm, although it might be more efficient to split into several `overall` constructs dealing with sites of different party.

The *restrict* operation (Figure 6.3) computes the residual and restricts it to a coarser grid. The residual is computed only at points of the fine grid matching points in the coarse grid (hence the strides of 2 in the overall constructs). A work-array, `tf`, aligned with `uf`, is used to hold the residual values. We then use a *remap* operation to copy these values to the coarse grid.

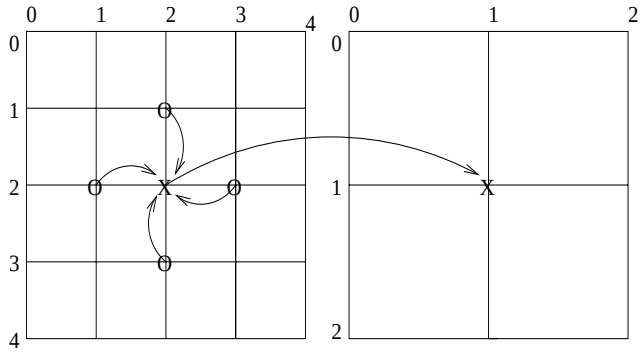

Figure 6.4. Illustration of restrict operation

The important `remap` operation copies one distributed array to another of the same shape which may have unrelated distribution format. We have made no assumptions of any relationship between the distribution of the fine grid array `uf` and the coarse grid array `uc`. In our code a subset of the elements of the work-array `tf` is copied into a subset of the array in coarse grid. Behavior of the restrict operation is illustrated in Figure 6.4.

The interpolate operation (Figure 6.5) is the opposite of restriction. It sends the correction computed on the coarse grid to the finer grid, and corrects current values on that grid. The `remap` and the array section expression are used to copy correction, `uc`, from coarse grid to matching point of a work-array, `tf`, on the fine grid. After the copying, we need to update boundary values using the `writeHalo` to get most up-to-date values. By the two nested overall constructs, it corrects current values of grid with the work-array `tf`. The first overall deals with fine grid points on horizontal links of the coarse grid and the second deals with those of vertical links—this is sufficient because it turns out that the correction is only needed on fine grid sites of a single parity. Behavior of the interpolate operation is illustrated in Figure 6.6.

6.2.2 Evaluation

Before attempting benchmark the full multigrid application, we experiment with simple kernel applications like Laplace equation and diffusion equation.

Figure 6.7 show result of four different versions (HPJava, sequential Java, HPF and Fortran) of red-black relaxation of the two dimensional Laplace equation with size of 512 by

```
static void interp(int npf, double[[-,-]] uc, double[[-,-]] uf) {

  // uc is correction, uf is output
  Range xf = uf.rng(0), yf = uf.rng(1);

  int nf = npf - 1;

  float [[-,-]] tf = new float [[xf, yf]] ;

  Adlib.remap(tf [[0 : nf : 2, 0 : nf : 2]], uc);

  Adlib.writeHalo(tf);

  overall(i = xf for 1 : nf - 1 : 2)
    overall(j = yf for 2 : nf - 2 : 2)
      uf [i, j] += 0.5 * (tf [i - 1, j] + tf [i + 1, j]);

  overall(i = xf for 2 : nf - 2 : 2)
    overall(j = yf for 1 : nf - 1 : 2)
      uf [i, j] += 0.5 * (tf [i, j - 1] + tf [i, j + 1]);
}
```

Figure 6.5. Interpolate operation.

512. In our runs HPJava can out-perform sequential Java by up to 17 times. On 36 processors HPJava can get about 79% of the performance of HPF. It is not very bad performance for the initial benchmark result without any serious optimization. Performance of the HPJava will be increased by applying optimization strategies as described in a previous paper [37]. Scaling behavior of HPJava is slightly better then HPF, though this mainly reflects the low performance of a single Java node compared to Fortran. We do not believe that the current communication library of HPJava is faster than the HPF library because our communication library is built on top of the portability layers, mpjdev and MPI, while IBM HPF is likely to use a platform specific communication library. But future versions of Adlib could be optimized for the platform.

Complete performance results of red-black relaxation of the Laplace equation are given in Table 6.1.

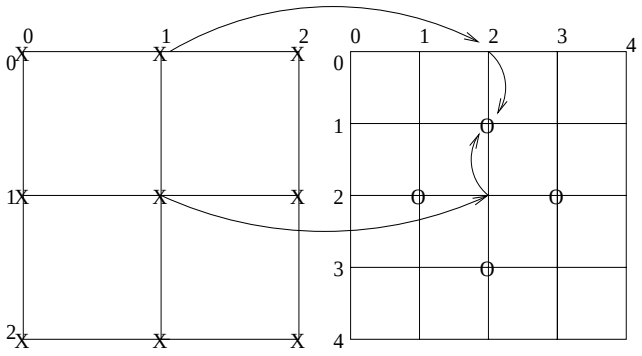

Figure 6.6. Illustration of interpolate operation

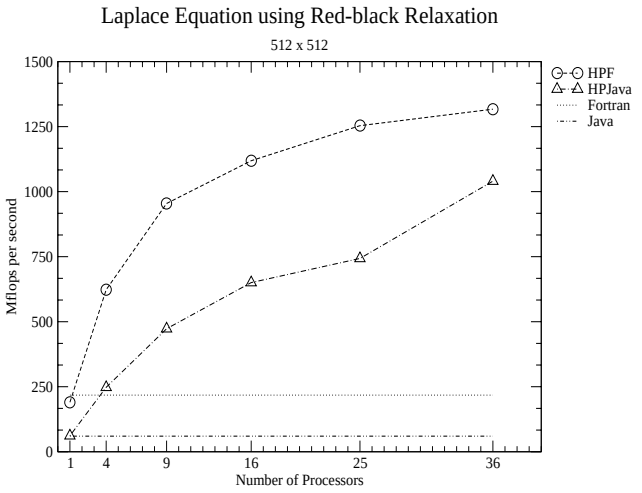

Figure 6.7. Red-black relaxation of two dimensional Laplace equation with size of 512^2.

We see similar behavior on large size of three dimensional Diffusion equation benchmark (Figure 6.8). In general we expect 3 dimensional problems will be more amenable to parallelism, because of the large problem size.

On a small problem size the three dimensional Diffusion equation benchmark (Figure 6.9) we can see the speed of sequential Fortran is about 4-5 times faster then Java. Benchmarking results from [11] do not see this kind of result on other platforms—a factor of 2 or less is common. Either IBM version of Fortran is very good or we are using an old Java compiler (JDK 1.3.1).

Table 6.1. Red-black relaxation performance. All speeds in MFLOPS.

256^2						
Processors	1	4	9	16	25	36
HPF	263.33	358.42	420.23	430.11	441.89	410.93
HPJava	69.12	184.33	258.06	322.58	430.12	430.10
Fortran	224.40					
Java	73.59					

512^2						
Processors	1	4	9	16	25	36
HPF	190.32	622.99	954.49	1118.71	1253.49	1316.96
HPJava	61.44	247.72	472.91	650.26	743.15	1040.40
Fortran	217.66					
Java	59.98					

1024^2						
Processors	1	4	9	16	25	36
HPF	104.66	430.27	1558.93	2153.58	2901.34	3238.71
HPJava	62.36	274.86	549.73	835.59	1228.81	1606.90
Fortran	149.11					
Java	58.73					

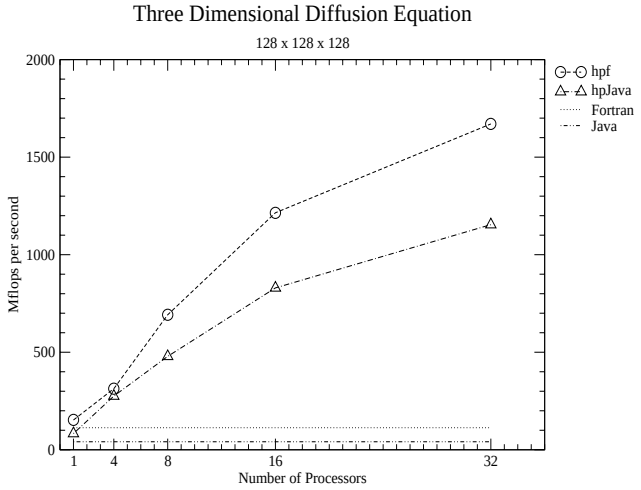

Figure 6.8. Three dimensional Diffusion equation with size of 128^3.

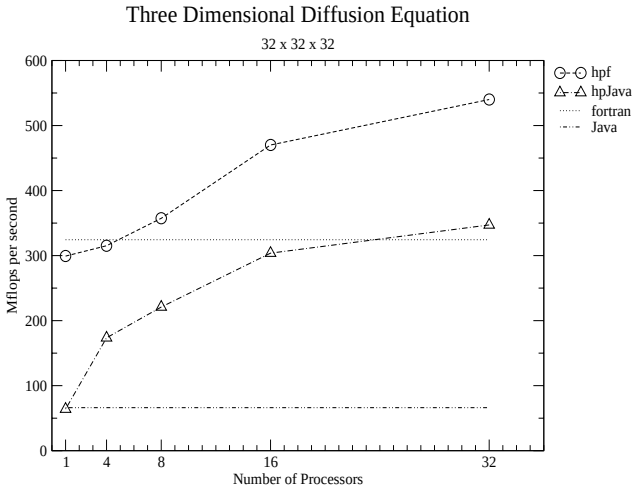

Figure 6.9. Three dimensional Diffusion equation with size of 32^3.

Table 6.2. Three dimensional Diffusion equation performance. All speeds in MFLOPS.

32^3						
Processors	1	2	4	8	16	32
HPF	299.26	306.56	315.18	357.35	470.02	540.00
HPJava	63.95	101.25	173.57	220.91	303.75	347.14
Fortran	113.00					
Java	66.50					

64^3						
Processors	1	2	4	8	16	32
HPF	274.12	333.53	502.92	531.32	685.07	854.56
HPJava	77.60	129.21	233.15	376.31	579.72	691.92
Fortran	113.00					
Java	66.50					

128^3						
Processors	1	2	4	8	16	32
HPF	152.55	185.15	313.16	692.01	1214.97	1670.38
HPJava	83.15	149.53	275.28	478.81	829.65	1154.06
Fortran	113.00					
Java	66.50					

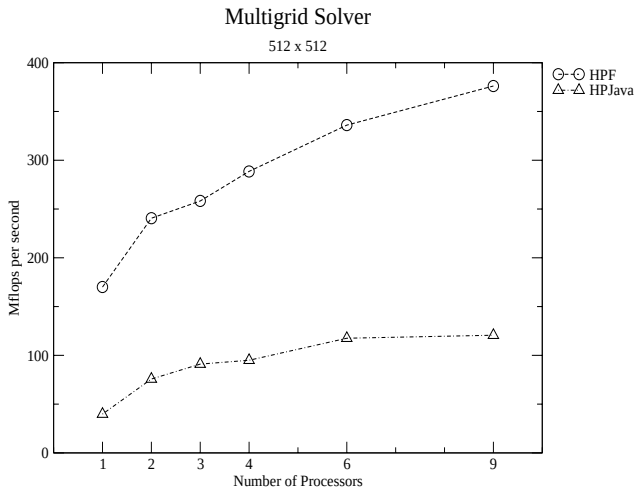

Figure 6.10. Multigrid solver with size of 512^2.

Table 6.3. Multigrid solver with size of 512^2. All speeds in MFLOPS.

512^2						
Processors	1	2	3	4	6	9
HPF	170.02	240.59	258.23	288.56	336.03	376.09
HPJava	39.77	75.70	91.02	94.97	117.54	123.15

Complete performance results of three dimensional Diffusion equation are given in Table 6.2.

Finally, we consider benchmark results on our original problem, the multigrid solver, in Figure 6.10 and Table 6.3. For the complete multigrid algorithm, speedup is relatively modest. This seems to be due to the complex pattern of communication in this algorithm. Neither the HPJava translation scheme or the Adlib implementation are yet optimized. We expect there is plenty of low hanging fruit in terms of opportunities for improving them.

Speedup of HPJava for the various applications is summarized in Table 6.4. Different size of problems are measured on different numbers of processors. For the reference value, we are using the result of the single-processor HPJava version. As we can see on the table we are getting up to 25.77 times speedup on Laplace equation using 36 processors with problem size of 1024^2. Many realistic applications with more computation for each grid point (for

Table 6.4. Speedup of HPJava benchmarks as compared with 1 processor HPJava.

Multigrid Solver					
Processors	2	3	4	6	9
512^2	1.90	2.29	2.39	2.96	3.03

2D Laplace Equation					
Processors	4	9	16	25	36
256^2	2.67	3.73	4.67	6.22	6.22
512^2	4.03	7.70	10.58	12.09	16.93
1024^2	4.41	8.82	13.40	19.71	25.77

3D Diffusion Equation					
Processors	2	4	8	16	32
32^3	1.58	2.72	3.45	4.75	5.43
64^3	1.67	3.00	4.85	7.47	8.92
128^3	1.80	3.31	5.76	9.98	13.88

example CFD which will be discussed in next section) will be more suitable for the parallel implementation than the Laplace equation and simple benchmarks described in this section.

6.3 HPJava with GUI

In this section we will illustrate how our HPJava can be used with a Java graphical user interface. The Java multithreaded implementation of mpjdev makes it possible for HPJava to cooperate with Java AWT. We ported the mpjdev layer to communicate between the threads of a *single* Java Virtual Machine (see section 5.5.2). The threads cooperate in solving a problem by communicating through our communication library, Adlib, with pure Java version of the mpjdev. In this version of the implementation we followed the mpjdev buffer specification which is described in Chapter 5. By adding pure Java version of the mpjdev to the Adlib communication library, it gives us the possibility to use the Java AWT and other Java graphical packages to support a GUI and visualize graphical output of the parallel application. Visualization of the collected data is a critical element in providing developers or educators with the needed insight into the system under study.

For test and demonstration of multithreaded version of mpjdev, we implemented computational fluid dynamics (CFD) code using HPJava which simulates 2 dimensional inviscid

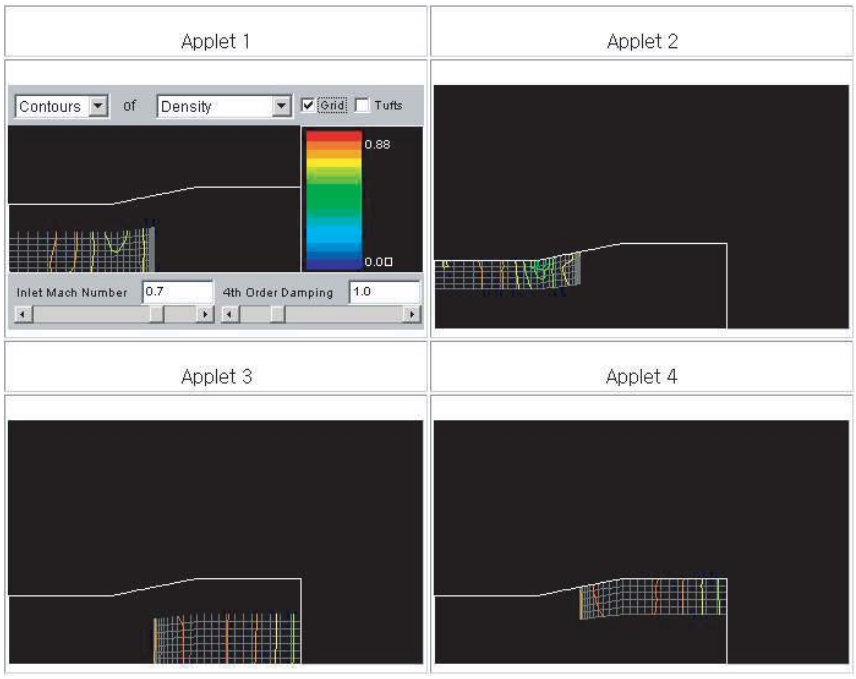

Figure 6.11. A 2 dimensional inviscid flow simulation.

flow through an axisymmetric nozzle(Figure 6.11). The simulation yields contour plots of all flow variables, including velocity components, pressure, Mach number, density and entropy, and temperature. The plots show the location of any shock wave that would reside in the nozzle. Also, the code finds the steady state solution to the 2 dimensional Euler equations, seen below.

$$\frac{\partial U}{\partial t} + \frac{\partial E}{\partial x} + \frac{\partial F}{\partial y} = \alpha H \tag{6.6}$$

Here $U = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ e_t \end{pmatrix}$, $E = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ (e_t + p)u \end{pmatrix}$, and $F = \begin{pmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ (e_t + p)v \end{pmatrix}$.

The source vector H is zero for this case.

The demo consists of 4 independent Java applets communicating through the Adlib communication library which is layered on top of mpjdev. Applet 1 is handling all events and broadcasting control variables to other applets. Each applet has the responsibility to draw its own portion of the data set into the screen, as we can see in the figure. That

Table 6.5. CFD speedup of HPJava benchmarks as compared with 1 processor HPJava.

CFD					
Processors	2	3	4	6	9
97 by 25	1.75	2.33	2.73	4.06	4.91

Processors	2	4	8	16
128^2	1.84	3.34	5.43	7.96
256^2	2.01	3.90	7.23	12.75

this demo also illustrates usage of Java object in our communication library. We are using `writeHalo()` method to communicate Java class object between threads.

This unusual interpretation of parallel computing, in which several applets in a single Web browser cooperate on a scientific computation, is for demonstration purpose only. The HPJava simulation code can also be run on a collection of virtual machines distributed across heterogeneous platforms like the native MPI of MPICH, SunHPC-MPI, and IBM POE (see next section).

You can view this demonstration and source code at

`http://www.hpjava.org/demo.html`

6.3.1 Evaluation

We also removed the graphic part of the CFD code and did performance tests on the computational part only. For this we also changed a 2 dimensional Java object distributed array into a 3 dimensional `double` distributed array and stored fields of the Java object into the collapsed 3rd dimension of `double` array. This change was to improve performance, because if we are using Java object to communicate between processors, there is an object serialization overhead which is not required for primitive data types. Also we are using HPC implementation of underlying communication to run the code on an SP.

Figure 6.12 shows result of two different versions (HPJava, sequential Java) of CFD with size of 256 by 256. Speedup of HPJava is also summarized in Table 6.5. As we mentioned earlier, we are measuring different size of problems on different number of processors and using the result of the single-processor HPJava version for the reference value. As we are

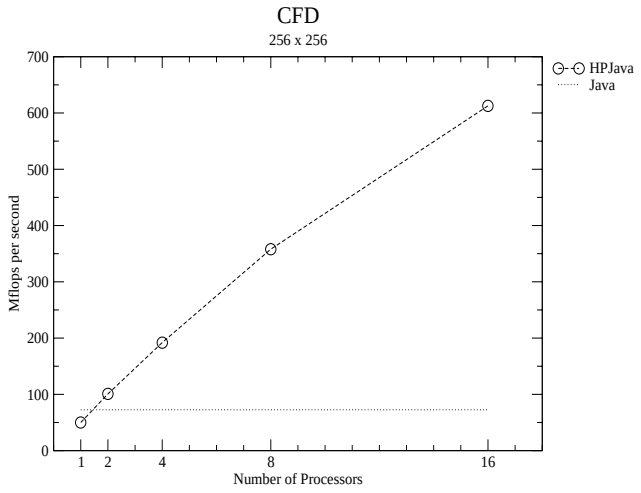

Figure 6.12. CFD with size of 256^2.

expected speed up of CFD is more scalable then partial differential equation examples which described in the previous section. As we can see on the table we are getting up to 12.75 times speedup (4.68 times speed up on Laplace equation) using 16 processors with problem size of 256^2.

Complete performance results from the CFD code are given in Table 6.6.

6.4 LAPI

Figure 6.13 and Table 6.7 show same benchmark results of an implementation of underlying communication library using LAPI. As we can see from the figure, the results of the sample benchmark indicate, unfortunately, that LAPI version of library is slower then MPI version. After careful investigation of the time consuming parts of the library, we found that current version of Java thread synchronization is not implemented with high performance. The Java thread consumes more then five times a long as POSIX thread, to perform wait and awake thread function calls (Table 6.8). This result suggests we should look for a new architectural design for mpjdev using LAPI. In this section we will not discuss in detail the new architecture design. However, we briefly introduce our thoughts. To eliminate major problem of current design, we consider using POSIX threads by calling JNI to the C instead of Java threads. This would force us to move any synchronized data from the Java

Table 6.6. CFD performance. All speeds in MFLOPS.

97 x 25						
Processors	1	2	3	4	6	9
HPJava	49.09	85.74	114.31	133.79	199.58	241.08
Java	72.03					

128^2					
Processors	1	2	4	8	16
HPJava	51.37	94.75	171.40	278.84	408.86
Java	72.42				

256^2					
Processors	1	2	4	8	16
HPJava	50.04	100.73	195.30	361.72	638.24
Java	72.56				

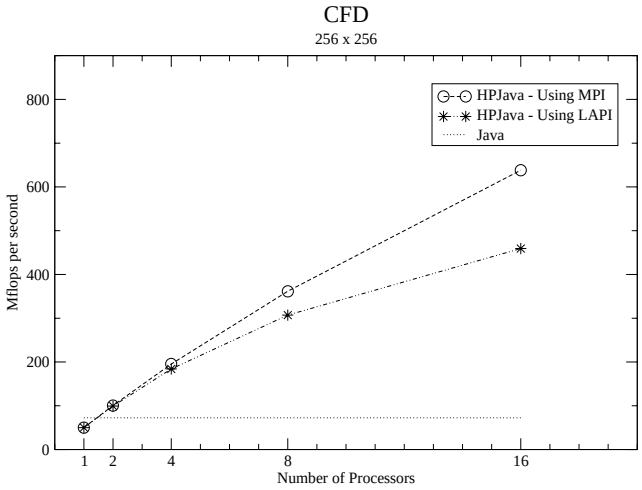

Figure 6.13. Comparison of the mpjdev communication library using MPI vs. LAPI.

Table 6.7. Comparison of the mpjdev communication library using MPI vs. LAPI. All speeds in MFLOPS.

CFD					
Processors	1	2	4	8	16
HPJava/MPI	50.04	100.73	195.30	361.72	638.24
HPJava/LAPI	50.08	99.44	184.09	307.12	459.22
Java	72.56				

Table 6.8. Timing for a wait and wake-up function calls on JAVA thread and POSIX Thread in microseconds.

Java Thread	POSIX Thread
57.49	10.68

Table 6.9. Latency of Laplace equation Communication Library per One Iteration and C/MPI `sendrecv()` function in microseconds.

`Adlib.writeHalo()`	Preparations	Communications	C/MPI (1368 bytes)
300.00	100.00	200.00	44.97

to the C side. In this design, work for the Java side of the mpjdev is to call C functions via JNI. All the actual communication and data processing parts including maintain send and receive queue, protection of any shared datas, and thread waiting and awaking will be done in C. Implementation is a future project.

6.5 Communication Evaluation

In this section we timed each part of an Adlib communication call to compare underlying communication latency with C/MPI and to find most time consuming part of the operation. This data can be used for further optimization of Adlib communication calls.

We divided HPJava communication into two parts: actual communication calls like `isend()`, `irecv()`, and `iwaitany()`, and communication preparations. In preparation parts, we include a high-level Adlib collective communication schedule like the method `remap()` and `writeHalo()`, and a message packing and unpacking parts of mpjdev. We measured timing of communication and preparation parts in a Laplace equation solver on 9 processors, and also measured `sendrecv()` function call using C/MPI with 1368 bytes on 2 processors (Table 6.9).

We use 9 processors to measure timing because this is the smallest number of processors with most time consuming communication pattern. Figure 6.14 illustrates communication patterns of `Adlib.writeHalo()` method among 9 processors with a 3 by 3 processes grid. This figure indicates that processor 4 is the most communication-based processor with 4

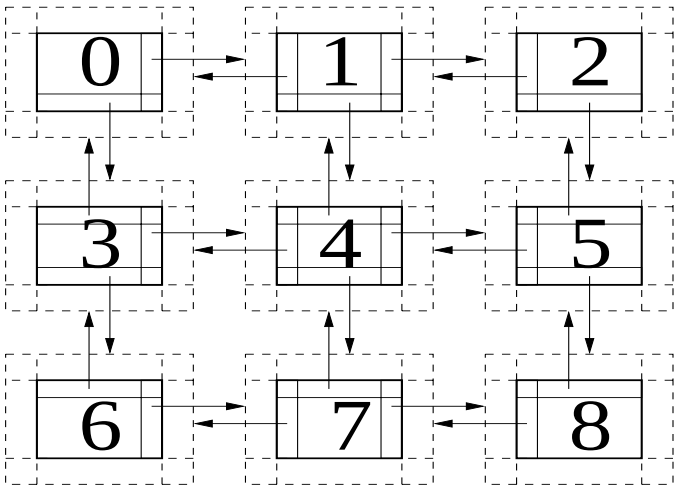

Figure 6.14. `writeHalo()` communication patterns on 9 processors.

pairs of send and receive. Since the size of the problem is 512^2, the processor 4 performed each send and receive with 171 double values (1368 bytes).

According to Table 6.9 the communication in `writeHalo()` takes about 200 microseconds. Meanwhile 4 pairs of send and receive communications are taking about 180 microseconds with C/MPI. So our communication library performs very close to the C/MPI version with marginal overhead. Overhead in our communication is due to the language binding and some extra work like finding the Java class and store communication result values during the JNI call. As we can see in the Table 6.9, one third (100 microseconds) of the total `writeHalo()` method timing (300 microseconds) is consumed by the preparation of communication. It is bit high but it is not such a bad performance for the initial implementation. Useful optimization can be done on this part in the future.

We see similar behavior in the CFD benchmark (Table 6.10). This is done on 9 processors with processes grid of 9 by 1 and problem size of 97 by 25. In this processes grid two send and receive communication is occurred on each processors, where first and last processor which only one send and receive is happening. We have about 20 microseconds of communication latency which is about same as previous case. About one fourth of total time is spent on preparation. This reduction of preparation time is due to the smaller problem size.

Table 6.10. Latency of CFD Communication Library per One Iteration and C/MPI `sendrecv()` function in microseconds.

`Adlib.writeHalo()`	Preparations	Communications	C/MPI (1728 bytes)
181.57	42.1	139.47	60.00

The above data indicates that further optimization is needed on *preparation* part of Adlib. In the future we also may adopt a platform specific communication library (for example, LAPI on AIX) instead of using MPI to reduce actual communication latency as discussed in section 5.5.3.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

We have explored enabling parallel, high-performance computation—in particular development of scientific software in the network-aware programming language, Java. Traditionally, this kind of computing was done in Fortran. Arguably, Fortran is becoming a marginalized language, with limited economic incentive for vendors to produce modern development environments, optimizing compilers for new hardware, or other kinds of associated software expected by today's programmers. Java looks like a promising alternative for the future.

In this dissertation, we have discussed motivations and features of HPJava, a new HPspmd programming environment. HPJava is an environment for SPMD parallel programming—especially, for SPMD programming with distributed arrays. HPJava has a lower-level programming model than HPF. Interfacing to other parallel-programming paradigms is more natural than in HPF. Both data parallel code and direct SPMD library calls are allowed in the same program. Various features and new concepts of HPJava were discussed using simple examples.

We have discussed in detail the design and development of high-level and low-level runtime libraries for HPJava—these are essentially communication libraries. The Adlib API is presented as high-level communication library. This API is intended as an example of an application level communication library suitable for data parallel programming in Java. This library fully supports Java object types, as part of the basic data types. We discussed implementation issues of collective communications in depth. The API and usage of three

different types of collective communications were also presented. Current implementation of this library is based on low-level communication library called mpjdev.

Java introduces implementation issues for message-passing APIs that do not occur in conventional programming languages. One important issue is how to transfer data between the Java program and the network while reducing overheads of the Java Native Interface. As contribution toward new low-level APIs, we developed a low-level Java API for HPC message passing, called mpjdev. The mpjdev API is a device level communication library. This library is developed with HPJava in mind, but it is a standalone library and could be used by other systems. We discussed message buffer and communication APIs of mpjdev and also format of a message. We also discussed three different implementations of mpjdev: mpiJava-based, multithreaded, and LAPI-based.

To evaluate current communication libraries, we did various performance tests. We developed small kernel level applications and a full application for performance test. We got reasonable performance on simple applications without any serious optimization. We also evaluated a communication performance of the high- and low-level libraries for future optimization.

7.2 Future Work

7.2.1 HPJava

The initial release of HPJava was made on April 1, 2003. It is freely available from `www.hpjava.org`. This release includes complete HPJava translator, two implementations of communication libraries (mpiJava-based and multithreaded), test suites, and all the applications described in this dissertation. In the future, further optimization of the HPJava translator is needed.

7.2.2 Communication Libraries

Some HPJava benchmark results were described in Chapter 6. We get good performance the simple problems like Laplace equation with the initial HPJava implementation. Results for the multigrid solver indicate further optimization for HPJava translation scheme and the Adlib communication library is desirable.

The evaluation of communication libraries (see section 6.5) indicates that further optimization of the preparation operations in Adlib would be useful. As we mentioned in section 6.4, we would also like a better design for LAPI implementation of mpjdev to avoid the overheads of Java thread operation.

We also need to develop portable network platforms based underlying communication library. As we discussed in Section 5.5.4, it may done by Jini-based implementation. The goals of this implementation are the system should be as easy to install on distributed systems as one can reasonably make it, and that it be sufficiently robust to be usable in an Internet environment.

APPENDIX A

THE JAVA ADLIB API

This appendix defines the Adlib communication library. An overview of Adlib was given in Chapter 5. Currently all communication functions in Adlib take the form of collective transformations on distributed arrays. These transformations are implemented in terms of *communication schedules*. Each kind of transformation has an associated class of schedules. Specific instances of these schedules, involving particular data arrays and particular parameters, are created as instances of the classes concerned. *Executing* a schedule initiates the communications required to perform the transformation. A single schedule may be executed many times, repeating the same communication pattern.

A.1 General Features

Adlib includes a family of related regular collective communication operations, a set of collective gather and scatter operations for more irregular communications, and a set of reduction operations based on the corresponding Fortran 90 array intrinsics. Reduction operations take one or more distributed arrays as input. They combine the elements to produce one or more scalar values, or arrays of lower rank. Adlib also provides a few I/O operations.

Only two public member parts, constructor(s) and an `execute()` method, are described in this chapter. All the input and output arrays and any parameters of the transformation are passed to the constructor. During execution of constructor, all send messages, receive messages, and internal copy operations implied by execution of the schedule are enumerated and stored in light-weight data structures. The `execute()` method nearly always involves communication. It should of course be treated as a collective operation, executed by all members of the active process group.

In this appendix we are using type variables T, t, and a notation like `Name`T, `Name`t for schedule names. The variable T runs over all primitive types and Java object types. The variable t typically runs over all primitive types other than boolean. Since each data type of Java initially has its own Adlib schedule class, we attach the type variable to the name of method to represent scope of method. A method `Name`T means that this particular method has all primitive type classes and Java object type class (e.g. `NameFloat`, `NameDouble`, ..., `NameObject`). A method `Name`t means that this particular method has all primitive type classes other than boolean type.

Below we briefly discuss various terms and notations used, following the subheadings used in the schedule definitions.

A.2 Glossary of Terms

Array Shape

As in Fortran, the *shape* of an array, a, is defined as the vector of extents of its ranges, ie (`a.rng(0).size()`, ..., `a.rng(R-1).size()`), where R is the rank of the array. Implicitly, if two array have the same shape, they also have the same rank.

Alignment

An array, `a` is *aligned with* an array `b` if they are distributed over the equivalent process groups and their ranges are all equivalent:

$$\begin{array}{lcl} \texttt{a.grp()} & \approx & \texttt{b.grp()} \\ \texttt{a.rng(0)} & \approx & \texttt{b.rng(0)} \\ & \vdots & \\ \texttt{a.rng(R-1)} & \approx & \texttt{b.rng(R-1)} \end{array}$$

Informally, two groups or two ranges are equivalent if they are *structurally equivalent*. The informal meaning of array alignment is that corresponding elements of the two arrays are stored on the same process, or replicated over the same group of processes.

The array, `a` is aligned with `b` *with replicated-alignment* in some dimensions if the groups are equivalent, and the ranges of `a` can be paired with equivalent ranges of `b` by omitting the ranges of `b` associated with the specified dimensions.

Containment

An array, `a`, is *fully contained* at a particular point in program execution if it is distributed over a group contained in the active process group:

$$\texttt{a.grp()} \subseteq \texttt{apg}$$

Informally this means that all copies of all elements of the array are available within the set of processes sharing the current thread of control.

Effect

In describing the effect of schedules, array subscripting notation will after be used informally. In this context, the subscripting should always be understood in terms of global subscripts to abstract global arrays, without reference to the distributed nature of the actual arrays.

Value Restrictions

These are simply restrictions on the input values of data, such as constraints ensuring values used as subscripts are in the required bounds.

Type Restrictions

Schedules that perform arithmetic operations or comparisons will impose further restrictions on the types of the array elements.

Shape Restrictions

Restrictions on the shape of the array arguments, such as the requirement that a particular pair of arrays passed to the constructor should have the same shape.

Alignment Restrictions

Many of the schedules in the library assume some alignment relations (see section A.2) between their array arguments. For example, it is required that the source array for a `Shift`

is aligned with the destination array. Historically, an essential feature of the shift operation is that it can be implemented very efficiently by simple nearest neighbour communications. The library could easily have been defined to implement shift without the alignment constraint, but then implementation would be essentially the same as the more complex `Remap` operation, and the main point of providing `Shift` is that it is a simpler, relatively light-weight operation. If versions of the library functions without alignment restrictions are needed, they can always be constructed by combining the constrained operation with `Remap` operations.

Containment Restrictions

Containment restrictions are needed to ensure that copies of array elements are available inside the group of processes that execute a schedule. Access to elements stored outside the active process group is not allowed by the collective communication paradigm currently implemented by Adlib.

Overlap Restrictions

In general the library does not allow in-place updates. No array written by a communication schedule should overlap with an array read by the schedule. The sections on individual to schedules give the specific restrictions.

Replicated Data

By definition, an array is replicated over a particular process dimension if the dimension appears in its distination group but not its signature (ie, the array has no range distributed over the dimension concerned).

As a rule it is good practise for programmers to maintain the same values in all copies of an element of a replicated array. If all arrays input to the communication schedules meet this requirement, it is guaranteed that those output do. This is not an absolute requirement on arrays passed to schedules, and the sections on individual schedules discuss the effect of defaulting on this rule.

A.3 Remap

A `remap()` operation is a communication schedule for copying the elements of one distributed array to another. The `remap()` method takes two distributed array arguments—a source array and a destination. The source and destination must have the same size, shape and same element-types, but no relation between the mapping of the two arrays is required. If the destination array has *replicated* mapping, the `remap()` operation will broadcast source values to all copies of the destination array elements.

The `remap()` method is a static member of the `Adlib` class. This operation can be applied to various ranks and type of array. Any section of an array with any allowed distribution format can be used. Supported element types include Java *primitive* and `Object` types. A general signature of the `remap()` function is

`void remap (`T `# destination,` T `# source)`

where the variable T runs over all primitive types and `Object`, and the notation T `#` means a multiarray of arbitrary rank, with elements of type T.

The `remap` method is implemented by a schedule object with HPspmd class `Remap`T. Each primitive type and `Object` has its own schedule class. This class has a constructor with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `Remap`T class is

```
public class RemapT implements HPspmd {
  public RemapT (T # destination, T # source) { ... }

  public void execute () { ... }
  . . .
}
```

The source array is the `source` and the destination array is `destination`.

A.3.1 Effect and Restrictions

Effect: Copy the values of the elements in the source array to the corresponding elements in the destination.

Type restrictions: The elements of the source and destination arrays must have the same type if primitive. If the element types are `Object`, subtypes of all objects referenced by elements of the source array must be assignable to elements of the destination array.

Shape restrictions: The source and destination arrays must have the same shape.

Containment restrictions: The source and destination arrays must be fully contained in the active process group.

Overlap restrictions: In-place updates are not allowed. The source and destination arrays must not overlap—no element of source array must be an alias for an element of the destination array. This is only an issue if both arguments are sections of the same array.

Replicated data: If the source array has replicated mapping, the value for a particular element is taken from one of its copies. If the destination array has replicated mapping, identical values are broadcast to every copy of the elements.

A.4 Shift

A `shift()` method is a communication schedule for shifting the elements of a distributed array along one of its dimensions, placing the result in another array. The source and destination arrays must have the same shape and element-type, and they must be identically aligned. The `shift()` operation does not copy values from source array that would go past the edge of destination array, and at the other extreme of the range elements of destination that are not targeted by elements from source array are unchanged from their input value. The related operation `cshift()` is essentially identical to `shift()` except that it implements a circular shift, rather then an "edge-off" shift.

General signatures of `shift()` function are

```
void shift (T [[-]] destination, T [[-]] source, int shiftAmount)
void cshift (T [[-]] destination, T [[-]] source, int shiftAmount)
```

and

```
void shift (T # destination, T # source, int shiftAmount,
            int dimension)
void cshift (T # destination, T # source, int shiftAmount,
             int dimension)
```

where the variable T runs over all primitive types and `Object`, and the notation T `#` means a multiarray of arbitrary rank, with elements of type T. The first form applies only for one dimensional multiarrays. The second form applies to multiarrays of any rank. The `shiftAmount` argument, which may be negative, specifies the amount and direction of the shift. In the second form the `dimension` argument is in the range 0, ...,R-1 where R is the rank of the arrays.

The `shift` method is implemented by a schedule object with HPspmd class `Shift`T. Each primitive type and `Object` has its own schedule class. This class has a constructor with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `Shift`T class is

```
public class Shift$T$ implements HPspmd {
  public Shift$T$ ($T$ # destination, $T$ # source,
                int shiftAmount , int dimension, int mode) { ... }

  public void execute () { ... }
  . . .
}
```

The source array is the `source` and the destination array is `destination`. The shift mount is given by `shiftAmount`. The `dimension` argument selects the array dimension in which the shift occurs. The flag `mode` specifies the type of shift. It takes one of the values `Adlib.CYCL`, `Adlib.EDGE`, or `Adlib.NONE`.

A.4.1 Effect and Restrictions

Effect: On exit, if `mode` is `Adlib.CYCL`, `cshift()`, the value of

$$\texttt{destination}\ [x_0, \ldots, x_{\texttt{dim}}, \ldots, x_{R-1}]$$

is

$$\texttt{source}\ [x_0, \ldots, x_{\texttt{dim}} + \texttt{shift} \mod N, \ldots, x_{R-1}]$$

where N is the extent of dimension `dim`. If `mode` is `Adlib.EDGE`, `shift()`, the exit value of the `destination` element is

$$\texttt{source}\ [x_0, \ldots, x_{\texttt{dim}} + \texttt{shift}, \ldots, x_{R-1}]$$

if $x_{\texttt{dim}} + \texttt{shift}$ is in the range $0, \ldots, N-1$, or *unchanged from the entry value*, if not. If `mode` is `Adlib.NONE` executing the schedule has no effect.

Value restrictions: The value of `dimension` must be in the range 0, ...,R-1 where R is the rank of the source array.

Type restrictions: The elements of the source and destination arrays must have the same type if primitive. If the element types are `Object`, subtypes of all objects referenced by elements of the source array must be assignable to elements of the destination array.

Shape restrictions: The source and destination array must have the same shape.

Alignment restrictions: The source array *must be aligned with* the destination array.

Containment restrictions: The source and destination arrays must be fully contained in the active progess group.

Overlap restrictions: In-place updates are not allowed. The source and destination arrays must not overlap—no element of source array must be an alias for an element of the destination array. This is only an issue if both arguments are sections of the same array.

Replicated data: If the arrays have replicated mapping, values for individual copies of the destination are generally taken from the nearest copy of the corresponding source array element. The definition of "nearest" is implementation dependent. This schedule does not implement a broadcast—consistent replication of copies in the destination array depends on consistency of copies in the source array.

A.5 Skew

A `skew()` method is a communication schedule for performing a skewed shift—a shift where the shift amount is itself an array—in a particular dimension of a distributed array placing the result in another array. The source and destination must have the same shape and same element-type, and they must be identically aligned. The `skew()` operation does not copy values from source array that would go past the edge of destination array, and at the other extreme of the range elements of destination that are not targeted by elements from source array are unchanged from their input value. The related operation `cskew()` is almost identical to `skew()` except that it implements a circular shift, rather then an "edge-off" shift.

General signature of `skew()` function is

```
void skew (T # destination, T # source, T # shift,  int dimension)
void cskew (T # destination, T # source, T # shift,  int dimension)
```

where the variable T runs over all primitive types and `Object`, and the notation T `#` means a multiarray of arbitrary rank, with elements of type T. The `shift` argument is a multiarray. The elements of this array specifies the amount and direction of the shift. The shift-amount array should have rank one less than the source array. The `dimension` argument is in the range 0, ...,R-1 where R is the rank of the arrays.

The `skew` method is implemented by a schedule object with HPspmd class `Skew`T. Each primitive type and `Object` has its own schedule class. This class has a constructor with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `Skew`T class is

```
public class ShiftT implements HPspmd {
  public SkewT (T # destination, T # source, T # shift,
                int dimension   , int mode) { ... }

  public void execute () { ... }
  . . .
}
```

The source array is the `source` and the destination array is `destination`. The array of shift mounts is `shift` The `dimension` argument selects the array dimension in which the shift

occurs. The flag `mode` specifies the type of shift. It takes one of the values `Adlib.CYCL`, `Adlib.EDGE`, or `Adlib.NONE`.

A.5.1 Effect and Restrictions

Effect: The description of the exit value of destination is identical to the description after execution of a `Shift` schedule (see section A.4), except that the constant `shift` is replaced by

`shift` $[x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$

Value restrictions: The value of `dimension` must be in the range 0, ...,R-1 where R is the rank of the source array.

Type restrictions: The elements of the source and destination arrays must have the same type if primitive. If the element types are `Object`, subtypes of all objects referenced by elements of the source array must be assignable to elements of the destination array.

Shape restrictions: The source and destination array must have the same shape. The shape of the shift array must be obtained from the shape of the source array by deleting dimension `dimension`.

Alignment restrictions: The source array *must be aligned with* the destination array. The shift-amount array should be aligned with the destination array, with replicated alignment over dimension `dimension`.

Containment restrictions: The source array, the shift-amount array, and the destination array must be fully contained in the active progess group.

Overlap restrictions: In-place updates are not allowed.

Replicated data: If the arrays have replicated mapping, values for individual copies of the destination are generally taken from the nearest copy of the corresponding source array element. The definition of "nearest" is implementation dependent. This schedule does not implement a broadcast—consistent replication of copies in the destination array relies on consistency of copies of the source array.

A.6 WriteHalo

A `writeHalo()` method is a collective communication operation used to fill the *ghost cells* or *overlap regions* surrounding the "physical" segment of a distributed array. The simplest versions have prototype

`void writeHalo (`T `# source)`

where the variable T runs over all primitive types and `Object`, and the notation T `#` means a multiarray of arbitrary rank, with elements of type T. More general forms of `writeHalo` allow to specify that only a subset of the available ghost area is to be updated, or to select circular wraparound for updating ghost cells at the extreme ends of the array:

`void writeHalo (`T `# source, int [] wlo, int [] whi)`

and

`void writeHalo (`T `# source, int [] wlo, int [] whi, int [] mode)`

Simplest form defines a schedule in which whole of the array ghost region is updated using `Adlib.EDGE` mode:

```
 who [r] = source.rng(r).loExtension()
 whi [r] = source.rng(r).hiExtension()
mode [r] = Adlib.EDGE
```

In general forms, the integer vectors `wlo`, `whi`, and `mode` have length R, the rank of the argument `source`. The values `wlo` and `whi` specify the widths at upper and lower ends of the bands to be updated. The upper and lower widths in dimension r are given by `wlo [`r`]` and `whi [`r`]`. These values are non-negative, and can only be non-zero if array `source` actually has suitable ghost extensions in the dimension concerned. More specifically, if the array `source` was created using a range with ghost extensions $\texttt{wlo}_{\text{act}}$, $\texttt{whi}_{\text{act}}$ as its rth dimension, ie

$$\begin{aligned} \texttt{wlo}_{\text{act}} &= \texttt{source.rng(}r\texttt{).loExtension()} \\ \texttt{whi}_{\text{act}} &= \texttt{source.rng(}r\texttt{).hiExtension()} \end{aligned}$$

it is required that

$$\begin{aligned} \texttt{whi}\ [r] &\leq \texttt{whi}_{\text{act}} \\ \texttt{wlo}\ [r] &\leq \texttt{wlo}_{\text{act}} \end{aligned}$$

The `writeHalo` method is implemented by a schedule object with HPspmd class `WriteHalo`T. Each primitive type and `Object` has its own schedule class. This class has two constructors with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `WriteHalo`T class is

```
public class WriteHaloT implements HPspmd {
  public WriteHaloT (T # source) { ... }

  public WriteHaloT (T # source, int [] wlo, int [] whi,
                     int [] mode) { ... }

  public void execute () { ... }
  . . .
}
```

The source array is the `source`. In the second form the values `wlo` and `whi` specify the widths at upper and lower ends of the bands to be updated. The flag `mode` specifies the type of shift. It takes one of the values `Adlib.CYCL`, `Adlib.EDGE`, or `Adlib.NONE`.

A.6.1 Effect and Restrictions

Effect: We distinguish between the locally held *physical segment* of an array and the surrounding *ghost region*, which is used to cache local copies of remote elements. The effect of this operation is to overwrite a portion of the ghost region—a halo of extent defined by the `wlo`, `whi` vectors of the constructor—with values from processes holding the corresponding elements in their physical segments. The operation is visualized in figure 3.6. If the value of the `mode` element for a dimension is `EDGE`, ghost cells past the extreme ends of the array range are not updated by the the write-halo operation. If the value is `CYCL`, those cells are updated assuming cyclic wraparound. If the value is `NONE`, there is no updating at all of the ghost cells associated with this dimension.

Containment restrictions: The source array must be fully contained in the active progess group.

Replicated data: If the array has replicated mapping, values for individual copies of the ghost cell are generally taken from the nearest copy of the corresponding physical array element. The definition of "nearest" is implementation dependent. This schedule does not implement a broadcast—consistent replication of copies in the destination array relies on consistency of copies of the source array.

A.7 Gather

A `gather()` operation is a communication schedule for collecting an arbitrary set of values from one distributed array (the source array) into the elements of another (the destination array). The selected set of elements is defined by a vector of subscript arrays, with an optional mask array. General signatures of the `gather()` function are

```
void gather (T # destination, T [[-]] source,
             int # subscripts)

void gather (T # destination, T [[-,-]] source,
             int # subscripts0, int # subscripts1)

void gather (T # destination, T [[-,-,-]] source,
             int # subscripts0, int # subscripts1, int # subscripts2)
```

and

```
void gather (T # destination, T [[-]] source,
             int # subscripts, boolean # mask)

void gather (T # destination, T [[-,-]] source,
             int # subscripts0, int # subscripts1, boolean # mask)

void gather (T # destination, T [[-,-,-]] source,
             int # subscripts0, int # subscripts1, int # subscripts2,
             boolean # mask)
```

where the variable T runs over all primitive types and `Object`, and the notation T `#` means a multiarray of arbitrary rank, with elements of type T. Currently the highest rank of source array with a `gather()` method is 3. The source and destination arrays can have different ranks. But the destination and subscript arrays are all the same shape, and all are aligned with one another. The number of subscript array arguments is equal to the rank of the source array. The second set of signatures take an extra boolean array aligned with the subscript array. Assignment to a destination element is conditional on the value of the element of the mask associated with the subscripts.

The `gather` method is implemented by a schedule object with HPspmd class `Gather`T. Each primitive type and `Object` has its own schedule class. This class has six constructors with arguments identical to the method above, and has one public method with no arguments

called `execute()`, which executes the schedule. The effective public interface of the `Gather`T class is

```
public class GatherT implements HPspmd {
  public GatherT (T # destination, T [[-]] source,
                  T # subscripts) { ... }

  public GatherT (T # destination, T [[-,-]] source,
                  T # subscripts0, T # subscripts1) { ... }

  public GatherT (T # destination, T [[-,-,-]] source,
                  T # subscripts0, T # subscripts1, T # subscripts2)
                  { ... }

  public GatherT (T # destination, T [[-]] source,
                  T # subscripts, boolean # mask) { ... }

  public GatherT (T # destination, T [[-,-]] source,
                  T # subscripts0, T # subscripts1,
                  boolean # mask) { ... }

  public GatherT (T # destination, T [[-,-,-]] source,
                  T # subscripts0, T # subscripts1, T # subscripts2,
                  boolean # mask) { ... }

  public void execute () { ... }
  . . .
}
```

A.7.1 Effect and Restrictions

Effect:

```
for all i0 in {0,...,N0 - 1} in parallel do
  ...
    for all iR-1 in {0,...,NR-1 - 1} in parallel do
      if (mask [i0, ..., iR-1])
        destination[i0, ..., iR-1] = source[subscripts0 [i0, ..., iR-1],
                                            subscripts1 [i0, ..., iR-1],
                                            ...] ;
```

where $(N_0, \ldots, N_{R-1})$ is the shape of `destination` array. If `mask` is absent, the assignment is unconditional.

Value restrictions: All elements of the rth subscript array must be in the range 0, ...,N-1 where N is the extent of the source array in its rth dimension.

Type restrictions: The elements of the source and destination arrays must have the same type if primitive. If the element types are `Object`, subtypes of all objects referenced by elements of the source array must be assignable to elements of the destination array.

Shape restrictions: The destination array, all subscript arrays, and the mask array, if defined, must have the same shape.

Alignment restrictions: All subscript arrays and the mask array, if defined, must be aligned with the destination array.

Containment restrictions: The source and subscript arrays, the mask array, if defined, and the destination array must be fully contained in the active progess group.

Overlap restrictions: In-place updates are not allowed. The source and destination arrays must not overlap—no element of source array must be an alias for an elements of destination array. This is only an issue if both arguments are sections of the same array.

Replicated data: If the source array has replicated mapping, the value for a particular element is taken from one of its copies. If the destination array has replicated mapping, identical values are broadcast to every copy of the elements.

A.8 Scatter

A `scatter()` operation is a communication schedule for scattering values from one distributed array (the source array) into the elements of another (the destination array). The selected set of elements is defined by a vector of subscript arrays, with an optional mask array. General signatures of the `scatter()` function are

```
void scatter (T # source, T [[-]] destination,
              int # subscripts)

void scatter (T # source, T [[-,-]] destination,
              int # subscripts0, int # subscripts1)

void scatter (T # source, T [[-,-,-]] destination,
              int # subscripts0, int # subscripts1, int # subscripts2)
```

and

```
void scatter (T # source, T [[-]] destination,
              int # subscripts, boolean # mask)

void scatter (T # source, T [[-,-]] destination,
              int # subscripts0, int # subscripts1, boolean # mask)

void scatter (T # source, T [[-,-,-]] destination,
              int # subscripts0, int # subscripts1, int # subscripts2,
              boolean # mask)
```

where the variable T runs over all primitive types and `Object`, and the notation T `#` means a multiarray of arbitrary rank, with elements of type T. Currently the highest rank of destination array with a `scatter()` method is 3. The source and destination arrays can have different ranks. But the destination and subscript arrays are all the same shape, and all are aligned with one another. The number of subscript array arguments is equal to the rank of the destination array. The second set of signatures take an extra boolean array aligned with the subscript array. Assignment to a destination element is conditional on the value of the element of the mask associated with the subscripts.

The `scatter` method is implemented by a schedule object with HPspmd class `Scatter`T. Each primitive type and `Object` has its own schedule class. This class has six constructors with arguments identical to the method above, and has one public method with no arguments

called `execute()`, which executes the schedule. The effective public interface of the Scatter$T$ class is

```
public class Scatter$T$ implements HPspmd {
  public Scatter$T$ ($T$ # destination, $T$ [[-]] source,
                $T$ # subscripts) { ... }

  public Scatter$T$ ($T$ # destination, $T$ [[-,-]] source,
                $T$ # subscripts0, $T$ # subscripts1) { ... }

  public Scatter$T$ ($T$ # destination, $T$ [[-,-,-]] source,
                $T$ # subscripts0, $T$ # subscripts1, $T$ # subscripts2)
                { ... }

  public Scatter$T$ ($T$ # destination, $T$ [[-]] source,
                $T$ # subscripts, boolean # mask) { ... }

  public Scatter$T$ ($T$ # destination, $T$ [[-,-]] source,
                $T$ # subscripts0, $T$ # subscripts1,
                boolean # mask) { ... }

  public Scatter$T$ ($T$ # destination, $T$ [[-,-,-]] source,
                $T$ # subscripts0, $T$ # subscripts1, $T$ # subscripts2,
                boolean # mask) { ... }

  public void execute () { ... }
  . . .
}
```

A.8.1 Effect and Restrictions

Effect:

```
for all i_0 in {0,...,N_0 - 1} in parallel do
  ...
    for all i_{R-1} in {0,...,N_{R-1} - 1} in parallel do
      if (mask [i_0, ..., i_{R-1}])
        destination[subscripts0 [i_0, ..., i_{R-1}],
                    subscripts1 [i_0, ..., i_{R-1}],
                    ...]                          = source [i_0, ..., i_{R-1}]
```

where $(N_0, \ldots, N_{R-1})$ is the shape of `destination` array. If `mask` is absent, the for loop is unconditional.

Value restrictions: All elements of the rth subscript array must be in the range 0, ...,N-1 where N is the extent of the source array in its rth dimension.

Type restrictions: The elements of the source and destination arrays must have the same type if primitive. If the element types are `Object`, subtypes of all objects referenced by elements of the source array must be assignable to elements of the destination array.

Shape restrictions: The destination array, all subscript arrays, and the mask array, if defined, must have the same shape.

Alignment restrictions: All subscript arrays and the mask array, if defined, must be aligned with the destination array.

Containment restrictions: The source and subscript arrays, the mask array, if defined, and the destination array must be fully contained in the active progess group.

Overlap restrictions: In-place updates are not allowed. The source and destination arrays must not overlap—no element of source array must be an alias for an elements of destination array. This is only an issue if both arguments are sections of the same array.

Replicated data: If the source array has replicated mapping, the value for a particular element is taken from one of its copies. If the destination array has replicated mapping, identical values are broadcast to every copy of the elements.

A.9 Sum

A `sum()` method is a reduction operation for adding together all elements of a distributed array. It has two prototypes

t `sum (`t `# source)`

and

t `sum (`t `# source, boolean # mask)`

where the variable t runs over all primitive types other than boolean, and the notation t `#` means a multiarray of arbitrary rank, with elements of type t. The second form takes an extra boolean array aligned with the source array and ignores all elements of `source` for which the corresponding element of `mask` is false.

The `sum` method is implemented by a schedule object with HPspmd class `Sum`t. Each primitive types other than boolean has its own schedule class. This class has two constructors with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `Sum`t class is

```
public class Sumt implements HPspmd {
  public Sumt (t # source) { ... }

  public Sumt (t # source, boolean # mask){ ... }

  public t execute () { ... }
  . . .
}
```

The source array is `source`. It will have elements of type t. The mask array is `mask`.

A.9.1 Effect and Restrictions

Effect: If mask is not present, executing the schedule adds together all elements of the array. If mask is present, executing the schedule adds together all elements of the array for which the corresponding element of the mask array is non-zero. The addition is performed in an unspecified order. It has same effect as without the `mask` if all values of the `mask` are true. If all elements of the `mask` are false, it will return zero. It has

same effect as all elements of source array are zero without the `mask`. The result value is broadcast to all members of the active process group.

Shape restrictions: The mask array if present must have the same shape as the source array.

Alignment restrictions: The mask array if present must be aligned with the source array.

Containment restrictions: The source array and the mask array if present must be fully contained in the active progess group.

Replicated data: If the source or mask array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

A.10 SumDim

A `sumDim()` method is a reduction operation for summing the elements of a distributed array along one of its dimensions, yielding a reduced array with the rank one less than the source. It has two prototypes

`void sumDim (`t `# res,` t `# source, int dimension)`

and

`void sumDim (`t `# res,` t `# source, boolean # mask, int dimension)`

where the variable t runs over all primitive types other than boolean, and the notation t `#` means a multiarray of arbitrary rank, with elements of type t. The result of this operation is written in `res`. The reduction occurs in dimension `dimension`. The second form takes an extra boolean array aligned with the source array and ignores all elements of `source` for which the corresponding element of `mask` is false.

The `sumDim` method is implemented by a schedule object with HPspmd class `SumDim`t. Each primitive types other than boolean has its own schedule class. This class has two constructors with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `SumDim`t class is

```
public class SumDimt implements HPspmd {
  public SumDimt (t # res, t # source, int dimension) { ... }

  public SumDimt (t # res, t # source, boolean # mask,
                  int dimension){ ... }

  public void execute () { ... }
  . . .
}
```

The source array is `source` and the result array is `res`. They will both have elements of type t. The reduction occurs in dimension `dimension`. The mask array is `mask`.

A.10.1 Effect and Restrictions

Effect: On exit, if `mask` is not present, the value of

$$\texttt{res}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is

$$\sum_{x_{\texttt{dim}}=0}^{N-1} \texttt{source}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

where N is the extent of the source array in dimension `dim`. The sum is performed in an unspecified order. If `mask` is present, the exit value is

$$\sum_{\substack{x_{\texttt{dim}}=0 \\ \texttt{mask}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, \\ x_{\texttt{dim+1}}, \ldots, x_{R-1}] \neq false}}^{N-1} \texttt{source}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

It has same effect as without the `mask` if all values of the `mask` are true. If all elements of the `mask` are false, `res` will store zero. It has same effect as if all elements of source array are zero without the `mask`.

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension `dim` of the source array.

Value restrictions: The value of `dimension` must be in the range 0, ...,R-1 where R is the rank of the source array.

Shape restrictions: The mask array if present must have the same shape as the source array. The shape of the result array must be obtained from the shape of the source array be deleting dimension `dimension`.

Alignment restrictions: The mask array if present must be aligned with the source array. The result array must be aligned to the source array, with replicated alignment in dimension `dimension`.

Containment restrictions: The source array, mask array if present, and the result array must be fully contained in the active progess group.

Overlap restrictions: In-place updates are not allowed.

Replicated data: If the source array or mask array if present has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

A.11 Product

A `product()` method is a reduction operation for multiplying together all elements of a distributed array. It has two prototypes

t `product (`t `# source)`

and

t `product (`t `# source, boolean # mask)`

where the variable t runs over all primitive types other than boolean, and the notation t `#` means a multiarray of arbitrary rank, with elements of type t. The second form takes an extra boolean array aligned with the source array and ignores all elements of `source` for which the corresponding element of `mask` is false.

The `product` method is implemented by a schedule object with HPspmd class `Product`t. Each primitive types other than boolean has its own schedule class. This class has two constructors with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `Product`t class is

```
public class Productt implements HPspmd {
  public Productt (t # source) { ... }

  public Productt (t # source, boolean # mask) { ... }

  public t execute () { ... }
  . . .
}
```

The source array is `source`. It will have elements of type t. The mask array is `mask`.

A.11.1 Effect and Restrictions

Effect: If `mask` argument is not present, executing the schedule multiplies together all elements of the array. If `mask` argument is present, executing the schedule multiplies together all elements of the array for which the corresponding element of the mask array is non-zero. The multiplication is performed in an unspecified order. It has same

effect as without the `mask` if all values of the `mask` are true. If all elements of the `mask` are false, it will return one. The result value is broadcast to all members of the active process group.

Shape restrictions: The mask array if present must have the same shape as the source array.

Alignment restrictions: The mask array if present must be aligned with the source array.

Containment restrictions: The source array and the mask array if present must be fully contained in the active progess group.

Replicated data: If the source or mask array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

A.12 ProductDim

A `productDim()` method is a reduction operation for multiplying together the elements of a distributed array along one of its dimensions, yielding a reduced array with the rank one less than the source. It has two prototypes

`void productDim (`t `# res,` t `# source, int dimension)`

and

`void productDim (`t `# res,` t `# source, boolean # mask, int dimension)`

where the variable t runs over all primitive types other than boolean, and the notation t `#` means a multiarray of arbitrary rank, with elements of type t. The result of this operation is written in `res`. The reduction occurs in dimension `dimension`. The second form takes an extra boolean array aligned with the source array and ignores all elements of `source` for which the corresponding element of `mask` is false.

The `productDim` method is implemented by a schedule object with HPspmd class `ProductDim`t. Each primitive types other than boolean has its own schedule class. This class has two constructors with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `ProductDim`t class is

```
public class ProductDimt implements HPspmd {
  public ProductDimt (t # res, t # source, int dimension) { ... }

  public ProductDimt (t # res, t # source, boolean # mask,
                 int dimension){ ... }

  public void execute () { ... }
  . . .
}
```

The source array is `source` and the result array is `res`. They will both have elements of type t. The reduction occurs in dimension `dimension`. The mask array is `mask`.

A.12.1 Effect and Restrictions

Effect: On exit, if `mask` is not present, the value of

$$\texttt{res}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is

$$\prod_{x_{\texttt{dim}}=0}^{N-1} \texttt{source}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

where N is the extent of the source array in dimension `dim`. The product is performed in an unspecified order. If `mask` is present, the exit value is

$$\prod_{\substack{x_{\texttt{dim}} = 0 \\ \texttt{mask}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, \\ x_{\texttt{dim+1}}, \ldots, x_{R-1}] \neq false}} \texttt{source}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

It has same effect as without the `mask` if all values of the `mask` are true. If all elements of the `mask` are false, `res` will store one.

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension `dim` of the source array.

Value restrictions: The value of `dimension` must be in the range 0, ...,R-1 where R is the rank of the source array.

Shape restrictions: The mask array if present must have the same shape as the source array. The shape of the result array must be obtained from the shape of the source array be deleting dimension `dimension`.

Alignment restrictions: The mask array if present must be aligned with the source array. The result array must be aligned to the source array, with replicated alignment in dimension `dimension`.

Containment restrictions: The source array, mask array if present, and the result array must be fully contained in the active progess group.

Overlap restrictions: In-place updates are not allowed.

Replicated data: If the source array or mask array if present has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition

of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

A.13 Maxval

A `maxval()` method is a reduction operation for finding the largest elements of a distributed array. It has two prototypes

t `maxval (`t `# source)`

and

t `maxval (`t `# source, boolean # mask)`

where the variable t runs over all primitive types other than boolean, and the notation t `#` means a multiarray of arbitrary rank, with elements of type t. The second form takes an extra boolean array aligned with the source array and ignores all elements of `source` for which the corresponding element of `mask` is false.

The `maxval` method is implemented by a schedule object with HPspmd class `Maxval`t. Each primitive types other than boolean has its own schedule class. This class has two constructors with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `Maxval`t class is

```
public class Maxvalt implements HPspmd {
  public Maxvalt (t # source) { ... }

  public Maxvalt (t # source, boolean # mask) { ... }

  public T execute () { ... }
  . . .
}
```

The source array is `source`. It will have elements of type T. The mask array is `mask`.

A.13.1 Effect and Restrictions

Effect: If `mask` argument is not present, executing the schedule finds the largest elements of the array. If `mask` argument is present, executing the schedule finds the largest elements of the array for which the corresponding element of the mask array is non-zero. It has same effect as without the `mask` if all values of the `mask` are true. If all elements of

the `mask` are false, it will return most negative value in its type. The result value is broadcast to all members of the active process group.

Shape restrictions: The mask array if present must have the same shape as the source array.

Alignment restrictions: The mask array if present must be aligned with the source array.

Containment restrictions: The source array and the mask array if present must be fully contained in the active progess group.

Replicated data: If the source array or mask array if present has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

A.14 MaxvalDim

A `maxvalDim()` method is a reduction operation for finding the largest elements of a distributed array along one of its dimensions, yielding a reduced array with the rank one less than the source. It has two prototypes

`void maxvalDim (`t `# res,` t `# source, int dimension)`

and

`void maxvalDim (`t `# res,` t `# source, boolean # mask, int dimension)`

where the variable t runs over all primitive types other than boolean, and the notation t `#` means a multiarray of arbitrary rank, with elements of type t. The result of this operation is written in `res`. The reduction occurs in dimension `dimension`. The second form takes an extra boolean array aligned with the source array and ignores all elements of `source` for which the corresponding element of `mask` is false.

The `maxvalDim` method is implemented by a schedule object with HPspmd class `MaxvalDim`t. Each primitive types other than boolean has its own schedule class. This class has two constructors with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `MaxvalDim`t class is

```
public class MaxvalDimt implements HPspmd {
  public MaxvalDimt (t # res, t # source, int dimension) { ... }

  public MaxvalDimt (t # res, t # source, boolean # mask,
                     int dimension) { ... }

  public void execute () { ... }
  . . .
}
```

The source array is `source` and the result array is `res`. They will both have elements of type t. The reduction occurs in dimension `dimension`. The mask array is `mask`.

A.14.1 Effect and Restrictions

Effect: On exit, the value of

`res` $[x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$

is the maximum value of

`source` $[x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$

for which

$$\texttt{mask}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}] \neq false$$

over the allowed values of $x_{\texttt{dim}}$. It has same effect as without the `mask` if all values of the `mask` are true. If all elements of the `mask` are false, `res` will store most negative value in its type.

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension `dim` of the source array.

Value restrictions: The value of `dimension` must be in the range 0, ...,*R*-1 where R is the rank of the source array.

Shape restrictions: The mask array if present must have the same shape as the source array. The shape of the result array must be obtained from the shape of the source array be deleting dimension `dimension`.

Alignment restrictions: The mask array if present must be aligned with the source array. The result array must be aligned to the source array, with replicated alignment in dimension `dimension`.

Containment restrictions: The source array, mask array if present, and the result array must be fully contained in the active progess group.

Overlap restrictions: In-place updates are not allowed.

Replicated data: If the source array or mask array if present has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

A.15 Minval

A `minval()` method is a reduction operation for finding the smallest elements of a distributed array. It has two prototypes

t `minval (`t `# source)`

and

t `minval (`t `# source, boolean # mask)`

where the variable t runs over all primitive types other than boolean, and the notation t `#` means a multiarray of arbitrary rank, with elements of type t. The second form takes an extra boolean array aligned with the source array and ignores all elements of `source` for which the corresponding element of `mask` is false.

The `minval` method is implemented by a schedule object with HPspmd class `Minval`t. Each primitive types other than boolean has its own schedule class. This class has two constructors with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `Minval`t class is

```
public class Minval t implements HPspmd {
  public Minval t (t # source) { ... }

  public Minval t (t # source, boolean # mask) { ... }

  public t execute () { ... }
  . . .
}
```

The source array is `source`. It will have elements of type t. The mask array is `mask`.

A.15.1 Effect and Restrictions

Effect: If `mask` argument is not present, executing the schedule finds the smallest elements of the array. If `mask` argument is present, executing the schedule finds the smallest elements of the array for which the corresponding element of the mask array is non-zero. It has same effect as without the `mask` if all values of the `mask` are true. If all elements

of the `mask` are false, it will return most positive value in its type. The result value is broadcast to all members of the active process group.

Shape restrictions: The mask array if present must have the same shape as the source array.

Alignment restrictions: The mask array if present must be aligned with the source array.

Containment restrictions: The source array and the mask array if present must be fully contained in the active progess group.

Replicated data: If the source array or mask array if present has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

A.16 MinvalDim

A `minvalDim()` method is a reduction operation for finding the smallest elements of a distributed array along one of its dimensions, yielding a reduced array with the rank one less than the source. It has two prototypes

`void minvalDim (`*t* `# res,` *t* `# source, int dimension)`

and

`void minvalDim (`*t* `# res,` *t* `# source, boolean # mask, int dimension)`

where the variable *t* runs over all primitive types other than boolean, and the notation *t* `#` means a multiarray of arbitrary rank, with elements of type *t*. The result of this operation is written in `res`. The reduction occurs in dimension `dimension`. The second form takes an extra boolean array aligned with the source array and ignores all elements of `source` for which the corresponding element of `mask` is false.

The `minvalDim` method is implemented by a schedule object with HPspmd class `MinvalDim`*t*. Each primitive types other than boolean has its own schedule class. This class has two constructors with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `MinvalDim`*t* class is

```
public class MinvalDimt implements HPspmd {
  public MinvalDimt (t # res, t # source, int dimension) { ... }

  public MinvalDimt (t # res, t # source, boolean # mask,
                  int dimension){ ... }

  public void execute () { ... }
  . . .
}
```

The source array is `source` and the result array is `res`. They will both have elements of type *t*. The reduction occurs in dimension `dimension`. The mask array is `mask`.

A.16.1 Effect and Restrictions

Effect: On exit, the value of

`res` $[x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$

is the minimum value of

`source` $[x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$

for which

$$\texttt{mask}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}] \neq false$$

over the allowed values of $x_{\texttt{dim}}$. It has same effect as without the `mask` if all values of the `mask` are true. If all elements of the `mask` are false, `res` will store most positive value in its type.

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension `dim` of the source array.

Value restrictions: The value of `dimension` must be in the range 0, ...,*R*-1 where R is the rank of the source array.

Shape restrictions: The mask array if present must have the same shape as the source array. The shape of the result array must be obtained from the shape of the source array be deleting dimension `dimension`.

Alignment restrictions: The mask array if present must be aligned with the source array. The result array must be aligned to the source array, with replicated alignment in dimension `dimension`.

Containment restrictions: The source array, mask array if present, and the result array must be fully contained in the active progess group.

Overlap restrictions: In-place updates are not allowed.

Replicated data: If the source array or mask array if present has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

A.17 All

A `all()` method is a boolean reduction operation for computing the logical conjunction of the elements of a distributed array of boolean values. This method returns true if and only if all element of `source` are true. The prototype is

```
boolean all (boolean # source)
```

where the notation `boolean #` means a multiarray of arbitrary rank with elements of type boolean.

The `all` method is implemented by a schedule object with HPspmd class `All`. This class has two constructors with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `All` class is

```
public class All implements HPspmd {
  public All (boolean # source) { ... }

  public boolean execute () { ... }
  . . .
}
```

The source array is `source`.

A.17.1 Effect and Restrictions

Effect: Executing the schedule forms the logical conjunction (boolean *and*) of the elements of the array. The result value is broadcast to all members of the active process group.

Replicated data: If the source array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

A.18 AllDim

A `allDim()` method is a boolean reduction operation for computing the logical conjunction of the elements of a distributed array of its dimensions, yielding a reduced array with the rank one less than the source. The prototype is

```
void allDim (boolean # res, boolean # source, int dimension)
```

where the notation `boolean #` means a multiarray of arbitrary rank with elements of type boolean. The result of this operation is written in `res`. The reduction occurs in dimension `dimension`.

The `allDim` method is implemented by a schedule object with HPspmd class `AllDim`. This class has two constructors with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `AllDim` class is

```
public class AllDim implements HPspmd {
  public AllDim (boolean # res, boolean # source, int dimension) { ... }

  public void execute () { ... }
  . . .
}
```

The source array is `source` and the result array is `res`. They will both have elements of type `boolean`. The reduction occurs in dimension `dimension`.

A.18.1 Effect and Restrictions

Effect: On exit, the value of

$$\texttt{res}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is true if

$$\texttt{source}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is true for *all* allowed values of $x_{\texttt{dim}}$.

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension `dim` of the source array.

Value restrictions: The value of `dimension` must be in the range 0, ...,R-1 where R is the rank of the source array.

Shape restrictions: The shape of the result array must be obtained from the shape of the source array be deleting dimension `dimension`.

Alignment restrictions: The result array must be aligned to the source array, with replicated alignment in dimension `dimension`.

Containment restrictions: The source array and the result array must be fully contained in the active progess group.

Overlap restrictions: In-place updates are not allowed.

Replicated data: If the source array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

A.19 Any

A `any()` method is a boolean reduction operation for computing the logical disjunction of the elements of a distributed array of boolean values. This method returns true if and only if any element of `source` is true. The prototype is

```
boolean any (boolean # source)
```

where the notation `boolean #` means a multiarray of arbitrary rank with elements of type boolean.

The `any` method is implemented by a schedule object with HPspmd class `Any`. This class has two constructors with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `Any` class is

```
public class Any implements HPspmd {
  public Any (boolean # source) { ... }

  public boolean execute () { ... }
  . . .
}
```

The source array is `source`. It will have elements of type `boolean`.

A.19.1 Effect and Restrictions

Effect: Executing the schedule forms the logical conjunction (boolean *and*) of the elements of the array. The result value is broadcast to all members of the active process group.

Replicated data: If the source array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

A.20 AnyDim

A `anyDim()` method is a boolean reduction operation for computing the logical disjunction of the elements of a distributed array of its dimensions, yielding a reduced array with the rank one less than the source. The prototype is

```
void anyDim (boolean # res, boolean # source, int dimension)
```

where the notation `boolean #` means a multiarray of arbitrary rank with elements of type boolean. The result of this operation is written in `res`. The reduction occurs in dimension `dimension`.

The `anyDim` method is implemented by a schedule object with HPspmd class `AnyDim`. This class has two constructors with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `AnyDim` class is

```
public class AnyDim implements HPspmd {
  public AnyDim (boolean # res, boolean # source, int dimension) { ... }

  public void execute () { ... }
  . . .
}
```

The source array is `source` and the result array is `res`. They will both have elements of type `boolean`. The reduction occurs in dimension `dimension`.

A.20.1 Effect and Restrictions

Effect: On exit, the value of

$$\texttt{res}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is true if

$$\texttt{source}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is true for *any* allowed values of $x_{\texttt{dim}}$.

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension `dim` of the source array.

Value restrictions: The value of `dimension` must be in the range 0, ...,R-1 where R is the rank of the source array.

Shape restrictions: The shape of the result array must be obtained from the shape of the source array be deleting dimension `dimension`.

Alignment restrictions: The result array must be aligned to the source array, with replicated alignment in dimension `dimension`.

Containment restrictions: The source array and the result array must be fully contained in the active progess group.

Overlap restrictions: In-place updates are not allowed.

Replicated data: If the source array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

A.21 Count

A `count()` method is a boolean reduction operation for counting the number of true elements in a distributed array of boolean values. This method returns true if and only if count element of `source` is true. The prototype is

```
boolean count (boolean # source)
```

where the notation `boolean #` means a multiarray of arbitrary rank with elements of type boolean.

The `count` method is implemented by a schedule object with HPspmd class `Count`. This class has a constructor with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `Count` class is

```
public class Count implements HPspmd {
  public Count (boolean # source) { ... }

  public int execute () { ... }
  . . .
}
```

The source array is `source`. It will have elements of type `boolean`.

A.21.1 Effect and Restrictions

Effect: Executing the schedule returns the number of true elements of the array. The result value is broadcast to all members of the active process group.

Containment restrictions: The source array array must be fully contained in the active progess group.

Replicated data: If the source array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

A.22 CountDim

A `countDim()` method is a boolean reduction operation for counting the number of true elements of a distributed array of boolean values along one of its dimensions, yielding a reduced array with the rank one less than the source. The prototype is

```
void countDim (boolean # res, boolean # a, int dimension)
```

where the notation `boolean #` means a multiarray of arbitrary rank with elements of type boolean. The result of this operation is written in `res`. The reduction occurs in dimension `dimension`.

The `countDim` method is implemented by a schedule object with HPspmd class `CountDim`. This class has a constructor with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `CountDim` class is

```
public class CountDim implements HPspmd {
  public CountDim (boolean # res, boolean # source, int dimension) { ... }

  public void execute () { ... }
  . . .
}
```

The source array is `source` and the result array is `res`. They will both have elements of type `boolean`. The reduction occurs in dimension `dimension`.

A.22.1 Effect and Restrictions

Effect: On exit, the value of

$$\texttt{res}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is the number of true (non-zero) elements

$$\texttt{source}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

over allowed values of $x_{\texttt{dim}}$.

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension `dim` of the source array.

Value restrictions: The value of `dimension` must be in the range 0, ...,R-1 where R is the rank of the source array.

Shape restrictions: The shape of the result array must be obtained from the shape of the source array be deleting dimension `dimension`.

Alignment restrictions: The result array must be aligned to the source array, with replicated alignment in dimension `dimension`.

Containment restrictions: The source array and the result array must be fully contained in the active progess group.

Overlap restrictions: In-place updates are not allowed.

Replicated data: If the source array has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

A.23 DotProduct

A `dotProduct()` method is a reduction operation for computing the dot product of two distributed arrays. The function `dotProduct` takes two aligned arrays as arguments and returns their scalar product—the sum of pairwise products of elements. The situation with element types is complicated because the types of the two arguments need not be identical. If they are different, standard Java binary numeric promotions are applied before multiplying elements. The prototypes are

t_3 `dotProduct(`t_1 `# a,` t_2 `# b)`

and

```
boolean dotProduct(boolean # a, boolean # b)
```

If either of t_1 or t_2 is a floating point type (`float` or `double`) the result type, t_3, is `double`). Otherwise the result type t_3 is `long`. The second form takes boolean as the arguments and returns the logical "or" of all the pairwise logical "ands" of elements. The argument multiarrays must have the same shape and must be aligned. The result is broadcasts to all members of the active process group.

The `dotProduct` method is implemented by a schedule object with HPspmd class `DotProduct`t. Each pair of primitive types other than boolean has its own schedule class. This class has a constructor with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `DotProduct`t class is

```
public class DotProduct t1t2 implements HPspmd {
  public DotProduct t1t2 (t1 # source1, t2 # source2) { ... }

  public t3 execute () { ... }
  . . .
}
```

All primitive type of Java combination other than boolean type is possible for t_1 and t_2. Type of t_3 is depends on the type of arguments t_1 and t_2.

A.23.1 Effect and Restrictions

Effect: Executing the schedule multiplies together corresponding elements of the source arrays, in pairs, then adds together all pairwise products. The addition occurs in an unspecified order. Boolean dot product equivalent to masked version of 'Any' (see section A.19). The result value is broadcast to all members of the active process group.

Shape restrictions: The two source arrays must have the same shape.

Alignment restrictions: The source arrays must be aligned with one another.

Containment restrictions: The source arrays must be fully contained in the active progess group.

Replicated data: If the source has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

A.24 Maxloc

A `maxloc()` method is a reduction operation for finding the location of the largest elements of a distributed array. This method rerun the maximum value of all element of an array—similar to `maxval` (see section A.13—but also output the index tuple in the array at which the extreme value was found. The prototypical forms are

t `maxloc (int [] loc,` t `# source)`

and

t `maxloc (int [] loc,` t `# source, boolean # mask)`

where the variable t runs over all primitive types other than boolean, and the notation t `#` means a multiarray of arbitrary rank, with elements of type t. `loc` is an ordinary Java array of length R, the rank of `source`. On exit it contains the global subscripts of the extreme value. The second form takes an extra boolean array aligned with the source array and ignores all elements of `source` for which the corresponding element of `mask` is false.

The `maxloc` method is implemented by a schedule object with HPspmd class `Maxloc`t. Each primitive types other than boolean has its own schedule class. This class has two constructors with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `Maxloc`t class is

```
public class Maxloct implements HPspmd {
  public Maxloct (int [] loc, t # source) { ... }

  public Maxloct (int [] loc, t # source, boolean # mask) { ... }

  public t execute () { ... }
  . . .
}
```

The source array is `source`. It will have elements of type t. The location of the largest value stored in `loc`. Size of `loc` vector must be equal to the rank of the source array. The mask array is `mask`.

A.24.1 Effect and Restrictions

Effect: If `mask` argument is not present the value of largest element in the array is returned. If `mask` argument is present the value of largest element in the array for which the corresponding element of the mask array is non-zero is returned. The global subscripts of the first occurrence of this element are written to the vector `loc`. If the maximum value occurs more than once in the array, "first occurrence" is defined by ordering the set of global subscripts with first subscript *least* significant. The result values returned value and location are broadcast to all member of the active process group.

It has same effect as without the `mask` if all values of the `mask` are true. If all elements of the `mask` are false, it will return most negative value in its type and `loc` will store most positive value in its type.

Shape restrictions: The mask array if present must have the same shape as the source array.

Alignment restrictions: The mask array if present must be aligned with the source array.

Containment restrictions: The source array and the mask array if present must be fully contained in the active progess group.

Replicated data: If the source array or mask array if present has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

A.25 MaxlocDim

A `maxlocDim()` method is a reduction operation for searching for the largest elements of a distributed array along one of its dimensions, yielding a reduced array with the rank one less than the source. It has two prototypes

```
void maxlocDim (t # res, t # loc, t # a, int dimension)
```

and

```
void maxlocDim (t # res, t # loc, t # source, boolean # mask,
                int dimension)
```

where the variable t runs over all primitive types other than boolean, and the notation t `#` means a multiarray of arbitrary rank, with elements of type t. The result of this operation is written in `res` and location is written in `loc`. The array `loc` has the same rank and alignment as `res` (since the reduction is in a single dimension, only one index value—for the specified dimension—meeds to be returned per extreme value). The reduction occurs in dimension `dimension`. The second form takes an extra boolean array aligned with the source array and ignores all elements of `source` for which the corresponding element of `mask` is false.

The `maxlocDim` method is implemented by a schedule object with HPspmd class `MaxlocDim`t. Each primitive types other than boolean has its own schedule class. This class has two constructors with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `MaxlocDim`t class is

```
public class MaxlocDimt implements HPspmd {
  public MaxlocDimt (t # res, t # loc, t # source,
                     int dimension) { ... }

  public MaxlocDimt (t # res, t # loc, t # source, boolean # mask,
                     int dimension) { ... }

  public void execute () { ... }
  . . .
}
```

The source array is `source` and the array of maximum values is `res`. They will both have elements of type t. The array of maximum locations is `loc`. The reduction occurs in dimension `dimension`. The mask array is `mask`.

A.25.1 Effect and Restrictions

Effect: On exit, the value of

$$\texttt{res}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is the maximum value of

$$\texttt{source}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

for which

$$\texttt{mask}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}] \neq false$$

over the allowed values of $x_{\texttt{dim}}$. The value of

$$\texttt{loc}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is the smallest $x_{\texttt{dim}}$ value at which this maximum occurs. It has same effect as without the `mask` if all values of the `mask` are true. If all elements of the `mask` are false, `res` will store most negative value in its type and `loc` will store most positive value in its type.

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension `dim` of the source array.

Value restrictions: The value of `dimension` must be in the range 0, ...,R-1 where R is the rank of the source array.

Shape restrictions: The shape of the two result arrays (`res` and `loc`) must be obtained from the shape of the source array by deleting dimension `dimension`. The mask array if present must be same shape as the source array.

Alignment restrictions: The two result arrays (`res` and `loc`) must be aligned to the source array, with replicated alignment in dimension `dimension`. The mask array if present must be aligned with the source array.

Containment restrictions: The source array and mask array if present and the result array must be fully contained in the active progess group.

Overlap restrictions: In-place updates are not allowed.

Replicated data: If the source array or mask array if present has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

A.26 Minloc

A `minloc()` method is a reduction operation for finding the location of the smallest elements of a distributed array. This method returns the minimum value of all element of an array—similar to `minval` (see section A.15—but also output the index tuple in the array at which the extreme value was found. The prototypical forms are

t `minloc (int [] loc,` *t* `# source)`

and

t `minloc (int [] loc,` *t* `# source, boolean # mask)`

where the variable *t* runs over all primitive types other than boolean, and the notation *t* `#` means a multiarray of arbitrary rank, with elements of type *t*. `loc` is an ordinary Java array of length *R*, the rank of `source`. On exit it contains the global subscripts of the extreme value. The second form takes an extra boolean array aligned with the source array and ignores all elements of `source` for which the corresponding element of `mask` is false.

The `minloc` method is implemented by a schedule object with HPspmd class `Minloc`*t*. Each primitive types other than boolean has its own schedule class. This class has two constructors with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `Minloc`*t* class is

```
public class Minloct implements HPspmd {
  public Minloct (int [] loc, t # source) { ... }

  public Minloct (int [] loc, t # source, boolean # mask) { ... }

  public t execute () { ... }
  . . .
}
```

The source array is `source`. It will have elements of type *t*. The location of the largest value stored in `loc`. Size of `loc` vector must be equal to the rank of the source array. The mask array is `mask`.

A.26.1 Effect and Restrictions

Effect: If `mask` argument is not present the value of smallest element in the array is returned. If `mask` argument is present the value of smallest element in the array for which the corresponding element of the mask array is non-zero is returned. The global subscripts of the first occurrence of this element are written to the vector `loc`. If the minimum value occurs more than once in the array, "first occurrence" is defined by ordering the set of global subscripts with first subscript *least* significant. The result values returned value and location are broadcast to all member of the active process group.

It has same effect as without the `mask` if all values of the `mask` are true. If all elements of the `mask` are false, it will return most positive value in its type and `loc` will store most positive value in its type.

Shape restrictions: The mask array if present must have the same shape as the source array.

Alignment restrictions: The mask array if present must be aligned with the source array.

Containment restrictions: The source array and the mask array if present must be fully contained in the active progess group.

Replicated data: If the source array or mask array if present has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent.

A.27 MinlocDim

A `minlocDim()` method is a reduction operation for searching for the smallest elements of a distributed array along one of its dimensions, yielding a reduced array with the rank one less than the source. It has two prototypes

```
void minlocDim (t # res, t # loc, t # a, int dimension)
```

and

```
void minlocDim (t # res, t # loc, t # source, boolean # mask,
                int dimension)
```

where the variable t runs over all primitive types other than boolean, and the notation t `#` means a multiarray of arbitrary rank, with elements of type t. The result of this operation is written in `res` and location is written in `loc`. The array `loc` has the same rank and alignment as `res` (since the reduction is in a single dimension, only one index value—for the specified dimension—meeds to be returned per extreme value). The reduction occurs in dimension `dimension`. The second form takes an extra boolean array aligned with the source array and ignores all elements of `source` for which the corresponding element of `mask` is false.

The `minlocDim` method is implemented by a schedule object with HPspmd class `MinlocDim`t. Each primitive types other than boolean has its own schedule class. This class has two constructors with arguments identical to the method above, and has one public method with no arguments called `execute()`, which executes the schedule. The effective public interface of the `MinlocDim`t class is

```
public class MinlocDimt implements HPspmd {
  public MinlocDimt (t # res, t # loc, t # source,
                     int dimension) { ... }

  public MinlocDimt (t # res, t # loc, t # source, boolean # mask,
                     int dimension) { ... }

  public void execute () { ... }
  . . .
}
```

The source array is `source` and the array of maximum values is `res`. They will both have elements of type t. The array of maximum locations is `loc`. The reduction occurs in dimension `dimension`. The mask array is `mask`.

A.27.1 Effect and Restrictions

Effect: On exit, the value of

$$\texttt{res}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is the minimum value of

$$\texttt{source}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

for which

$$\texttt{mask}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}] \neq false$$

over the allowed values of $x_{\texttt{dim}}$. The value of

$$\texttt{loc}\ [x_0, \ldots, x_{\texttt{dim-1}}, x_{\texttt{dim+1}}, \ldots, x_{R-1}]$$

is the smallest $x_{\texttt{dim}}$ value at which this minimum occurs. It has same effect as without the `mask` if all values of the `mask` are true. If all elements of the `mask` are false, `res` and `loc` will store most positive value in its type.

As implied by the replicated alignment of the result array, results are broadcast in the process dimension associated with dimension `dim` of the source array.

Value restrictions: The value of `dimension` must be in the range 0, ...,R-1 where R is the rank of the source array.

Shape restrictions: The shape of the two result arrays (`res` and `loc`) must be obtained from the shape of the source array by deleting dimension `dimension`. The mask array if present must be same shape as the source array.

Alignment restrictions: The two result arrays (`res` and `loc`) must be aligned to the source array, with replicated alignment in dimension `dimension`. The mask array if present must be aligned with the source array.

Containment restrictions: The source array, mask array if present, and the result array must be fully contained in the active progess group.

Overlap restrictions: In-place updates are not allowed.

Replicated data: If the source array or mask array if present has replicated mapping, values for particular elements are generally taken from the nearest copy. The definition of "nearest" is implementation dependent. Consistent replication of copies in the result array relies on consistency of copies in the source array.

A.28 APrintf

A `aprintf` method is a I/O operation that modelled on the `C printf()` function. Its arguments are a control string and a list of distributed arrays. The effective prototypes are

`void aprintf(String control,` T `# source)`

`void aprintf(String control,` T_0 `# source0,` T_1 `# source1)`

`void aprintf(String control,` T_0 `# source0,` T_1 `# source1,` T_2 `# source2)`

For now a maximum of three input arrays is allowed. If more than one array is specified, all should have the same shape. But they can have any, unrelated distribution format.

The `aptintf` method is implemented by a class `APrintf`. The effective public interface of the `APrintf` class is

```
public class APrintf {
   public APrintf(String control, IOArg[] args) { ... }
  . . .
}
```

The control string is `control`. The multidimensional arrays are stored into the array `args`.

A.28.1 Effect and Restrictions

Effect: In a pseudocode notation, the general behaviour of `aprintf()` is like

```
for each i0 in (0,...,N0 - 1) in sequence do
  ...
    for each iR-1 in (0,...,NR-1 - 1) in sequence do
      printf(control, source0 [i0, ..., iR-1],
                      source1 [i0, ..., iR-1],
                      ...)
```

where $(N_0, \ldots, N_{R-1})$ is the shape the arrays. The integer value i_0 is interpolated into the output wherever there is a `%R0` in the control string; the value i_1 is interpolated wherever there is a `%R1`; and so on. The imaginary elemental `printf` operation outputs to `System.out` on the root process of the active process group.

Special options:

%RD: It prints the current index value. D stands for integer number like `%R0`, `%R1`, and so on. The format `%R`D interpolates the index value into the string without the need to initialize an extra array of integer.

Example:

```
float [[-,-]] a = new float [[x,y]] ;
overall(i = x for :)
  overall(j = y for :)
    a [i, j] = 10.0F * i‘ + j‘;

Adlib.aprintf("a[%R0, %R1] = %f\n", a) ;
```

```
Output:
  a [0, 0] = 0.0
  a [0, 1] = 1.0
  a [0, 2] = 2.0
  a [1, 0] = 10.0
  a [1, 1] = 11.0
  a [1, 2] = 12.0
```

%N: It provides line breaking. It behaves exactly like the `\n` escape sequence. But `%N` allow an integer modifier. It defines the frequency with which the newline is printed. If the value of the modifier is w, the new line is only printed in every wth elemental print operation.

Example: If we replace the `aprintf()` call in the previous example with

```
Adlib.aprintf("a[%R0, %R1] = %f    %3N", a) ;
```

```
Output:
  a [0, 0] = 0.0    a [0, 1] = 1.0    a [0, 2] = 2.0
  a [1, 0] = 10.0    a [1, 1] = 11.0    a [1, 2] = 12.0
```

Shape restrictions: All arrays should have the same shape.

A.29 gprintf

A `gprintf` is a I/O operation for printing a "global" `String` value. All they do is output string to `System.out` on the root process of the active process group. They have interfaces:

```
gprint(String string)

gprintln(String string)
```

The interface `gprintf()` is equivalent to `System.out.print()` and the interface `gprintln()` is eqivalent to `System.out.println()`.

REFERENCES

[1] Common Language Infrastructure (CLI), Partition III, CIL Instruction SET. Technical Report ECMA TC39/TG3, Fujitsu Software, Hewlett-Packard, Intel Corporation, International Business Machines, ISE, Microsoft Corporation, Monash University, Netscape, OpenWave, Plum Hall, Sun Microsystems.

[2] Java for Computational Science and Engineering–Simulation and Modelling. *Concurrency: Practice and Experience*, 9(6), June 1997.

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

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

[5] C.A. Addison, V.S. Getov, A.J.G. Hey, R.W. Hockney, and I.C. Wolton. *The Genesis Distributed-Memory Benchmarks*. Elsevier Science B.V., North-Holland, Amsterdam, 1993.

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

[7] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification*. 1999.

[8] 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.

[9] Mark Baker and Bryan Carpenter. MPJ: A proposed Java message-passing API and environment for high performance computing. In *International Workshop on Java for Parallel and Distributed Computing*, Cancun, Mexico, May 2000. To be presented.

[10] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial*. The Society for Industrial and Applied Mathematics (SIAM), 2000.

[11] M. Bull, L. Smith, L. Pottage, and R. Freeman. Benchmarking Java against C and Fortran for Scientific Applications. In *ACM 2001 Java Grande/ISCOPE Conference*. ACM Press, 2001.

[12] B. Carpenter, G. Zhang, and Y. Wen. NPAC PCRC runtime kernel definition. Technical Report CRPC-TR97726, Center for Research on Parallel Computation, 1997. http://www.npac.syr.edu/projects/pcrc/kernel.html.

[13] Bryan Carpenter, Vladimir Getov, Glenn Judd, Anthony Skjellum, and Geoffrey Fox. MPJ: Mpi-like message passing for java. *Concurrency: Practice and Experience*, 12(11):1019–1038, 2000.

[14] 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/.

[15] 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.

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

[17] 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.

[18] 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/projects/javelin/.

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

[20] Visual Studio .NET home page. http://msdn.microsoft.com/vstudio/nextgen/.

[21] 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.

[22] Linda G. DeMichiel, L. Ümit Yalçinalp, and Sanjeev Krishnan. Enterprise JavaBeans specification, Version 2.0. Technical report, Sun Microsystems, August 2001. http://java.sun.com/products/ejb/.

[23] 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.

[24] W. Keith Edwards. *Core Jini*. 1999.

[25] Enterprise JavaBeans home page. http://java.sun.com/products/ejb/.

[26] 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.

[27] Stephen J. Fink and Scott B. Baden. Run-time data distribution for blockstructured applications on distributed memory computers. In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, IMA Volumes in Mathmatics and its Applications. Springer-Verlag, February 1995. http://www-cse.ucsd.edu/groups/hpcl/scg/kelp.html.

[28] 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.

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

[30] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Pararallel Virtual Machine—A Users' Guids and Tutorial for Networked Parallel Computing*. Scientific and Engineering Series. MIT Press, 1994.

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

[32] Hewlett Packard Company. *E-speak Architecture Specification*, September 1999. http://www.e-speak.hp.com/.

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

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

[35] Java Message Service API home page. http://java.sun.com/products/jms/.

[36] Project JXTA home page. http://www.jxta.org.

[37] Han-Ku Lee, Bryan Carpenter, Geoffrey Fox, and Sang Boem Lim. Benchmarking hpjava: Prospects for performance. In *Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers(LCR2002)*, March 2002.

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

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

[40] 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/.

[41] 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.

[42] 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.

[43] J.E. Moreira, S.P. Midkiff, and M. Gupta. A comparision of three approaches to language, compiler and library support for multidimensional arrays in Java. In *ACM 2001 Java Grande/ISCOPE Conference*. ACM Press, June 2001.

[44] 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/.

[45] Java Numberics home page. http://math.nist.gov/javanumerics/.

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

[47] 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.

[48] Parabon Computation, Inc. home page. http://www.parabon.com.

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

[50] 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.

[51] 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/.

[52] SEIT@home home page. http://setiathome.ssl.berkeley.edu.

[53] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, 1994.

[54] Jeremiah Willcock, Andrew Lumsdaine, and Aech Robison. Using mpi with c# and the common language infrastructure. *Concurrency and Computation: Practice and Experience*, 2003.

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

[56] 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.

[57] 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.

[58] 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://aspen.csit.fsu.edu/projects/pcrc.

BIOGRAPHICAL SKETCH

Sang Boem Lim

PLACE OF BIRTH: Jeonju, Korea

DATE OF BIRTH: January 16, 1973

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

- Eastern Michigan University, Ypsilanti, Michigan, U.S.A.
- Syracuse University, Syracuse, New York, U.S.A.

DEGREE AWARDED:

- Bachelor of Science in Computer Science, 1995, Eastern Michigan University
- Master of Science in Computer Science, 1998, Syracuse University

PROFESSIONAL EXPERIENCE:

- Visiting Research Associate, Pervasive Technology Labs, Indiana University
- Research Assistant, School of Computational Science and Information Technology, Florida State University
- Research Assistant, Northeast Parallel Architecture Center, Syracuse University

PUBLICATIONS:

- Sang Boem Lim, Bryan Carpenter, Geoffrey Fox, and Han-Ku Lee. Collective Communication for the HPJava Programming Language. To appear *Concurrency and Computation, Practice and Experience*, 2003

- Han-Ku Lee, Bryan Carpenter, Geoffrey Fox, Sang Boem Lim. HPJava: Programming Support for High-Performance Grid-Enabled Applications. The 2003 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA03), June 2003
- Han-Ku Lee, Bryan Carpenter, Geoffrey Fox, Sang Boem Lim. HPJava: Efficient Compilation and Performance for HPC. The Seventh World Multiconference on Systems, Cybernetics, and Informatics (SCI 2003), July 2003.
- Han-Ku Lee, Bryan Carpenter, Geoffrey Fox, Sang Boem Lim. Benchmarking HPJava: Prospects for Performance. *Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR2002)*, March 2002.
- Bryan Carpenter, Geoffrey Fox, Han-Ku Lee and Sang Lim. Translation of the HPJava Language for Parallel Programming. *The 14th annual workshop on Languages and Compilers for Parallel Computing(LCPC2001)*, May 2001.
- 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.
- 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.
- 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.
- Bryan Carpenter and Sang Boem Lim. *A Low-level Java API for HPC Message Passing*. February 27, 2002.
- Bryan Carpenter, Han-Ku Lee, Sang Boem Lim, Geoffrey Fox, and Guansong Zhang, . *Parallel Programming in HPJava*. Draft of May 2001.
  http://www.hpjava.org/

- Bryan Carpenter, Geoffrey Fox, Sung-Hoon Ko and Sang Lim. *mpiJava 1.2: API Specification*. October 1999.
  http://www.hpjava.org/mpiJava.html