A DEVICE LEVEL COMMUNICATION LIBRARY FOR THE HPJAVA PROGRAMMING LANGUAGE

Sang Boem Lim, Bryan Carpenter, Geoffrey Fox, and Han-ku Lee
{sblim, dbcarpen, gcf, hanklee}@indiana.edu

Pervasive Technology Labs at Indiana University
501 N. Morton St. Suite 224
Bloomington, IN 47404
U.S.A.

Abstract

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. This communication library supports *collective operations* on distributed arrays. The mpjdev API is a device level underlying communication library for HPJava. This library is developed to perform actual communication between processes.

The paper describes the novel issues in the implementation of device level library 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**.

Key Words

Distributed Software Systems and Applications, Compiler and Runtime Support, Parallel and Distributed Compiler, Java.

1. Introduction

HPJava [1, 2, 3] is the authors' environment for SPMD (Single Program, Multiple Data) parallel programming---especially, for SPMD programming with distributed arrays---in Java. It includes a set of syntax extension to Java for dealing with multi-dimensional distributed arrays, plus a set of communication libraries. The HPJava language and the high-level collective library Adlib [4] has been described in several earlier papers. This paper will concentrate more on the implementation of device level underlying communication library on different platforms.

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. In this paper we will illustrate how mpjdev can be naturally be implemented in terms of the primitives of the HPJava language. The mpjdev API is a low-level Java messaging platform which has the potential to be used as a common API for implementing libraries like Adlib and its relatives.

```
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 1: A parallel matrix addition.

Section 2 briefly reviews the HPJava language and includes a few illustrative programming fragments. Section 3 discussed collective communication in HPJava. Section 4 describes the low-level communication library, mpjdev. In this section, three different implementations of mpjdev are also discussed. Benchmark results for each implementation are discussed in Section 5. Conclusions are drawn together in Section 6.

2. Features of HPJava

Figure 1 is a simple HPJava program for a matrix addition. It illustrates much of the HPJava special syntax, like 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 [5, 6]).

The variables *a*, *b*, and *c* are all distributed array variables. The creation expressions on the right hand side of the initializes 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*). The class **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*.

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**.

3. High-level Communication Library

Adlib library has been described in earlier paper in depth. Here we briefly review usage of high-level communication library in HPJava programs using a simple program.

Figure 2 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.

```
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 2: Solution of Laplace equation by Jacobi relaxation.

The method **Adlib.writeHalo()** 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. 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.

Besides **writeHalo()**, Adlib includes a family of related regular collective communication operations (remaps, 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 operations. Reduction operations combine the elements of two distributed arrays 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*.

4. A low-level communication library for Java HPC

The mpjdev API is designed with the goal that it can be implemented *portably* on network platforms and

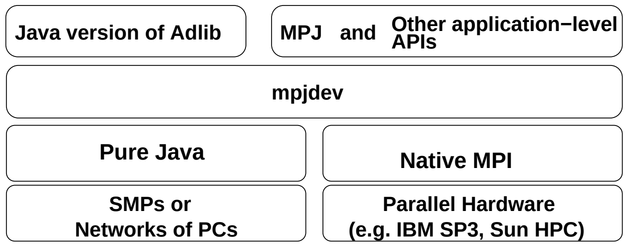

Figure 3: An HPJava communication stack.

efficiently on parallel hardware. It needs to support communication of 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.

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 [7] might be implemented on top of mpjdev. mpjdev itself may be implemented on top of Java sockets in a portable network implementation, or—on HPC platforms---through a JNI interface to a subset of MPI. The positioning of the mpjdev API is illustrated in Figure 3.

The currently specified API for mpjdev is small compared to MPI. It only includes point-to-point communications. Currently the only messaging modes 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.

Currently we have three different implementations of mpjdev: mpiJava-based, multithreaded, and LAPI-based.

4.1 mpiJava-based Implementation

The mpiJava-based implementation assumes C binding of native method calls to MPI from mpiJava [8] as basic communication protocol. This implementation can be divided into two parts: Java APIs (**Buffer** and **Comm** classes) 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** (different from the Java side Buffer class) is used for maintain message vector. This lightweight object stores a pointer to the

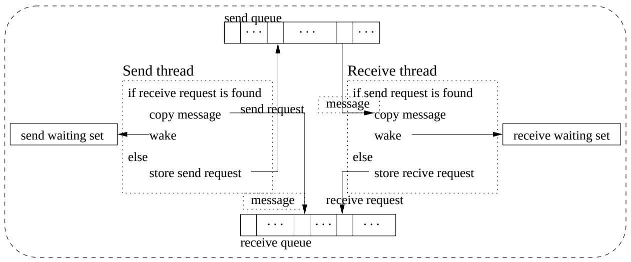

Figure 4: Multithreaded implementation.

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. 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.

4.2 Multithreaded Implementation

In this implementation, the processes of an HPJava program are mapped to the Java threads of a single JVM. This allows debugging and demonstrating 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 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 4 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.

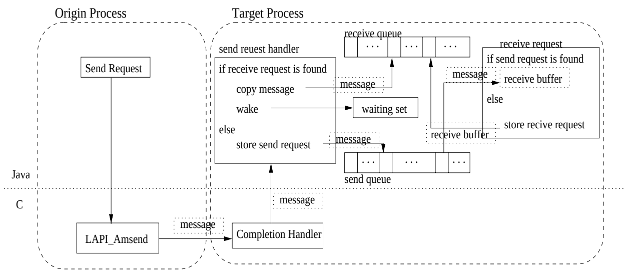

Figure 5: LAPI implementation with Active Message Call.

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 stored in this set will be blocked until 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 continues 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.

4.3 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.

Figure 5 illustrates LAPI implementation with *active message* function (**LAPI_Amsend**). The active message infrastructure 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.

This implementation stores and manages message buffer in C, like in mpiJava-based implementation. This implementation uses Java thread synchronization to implement waiting in the MPI and uses two static objects—"send queue" and "receive queue"—to maintain early arrived send and receive requests.

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 actual messages. Those information are used to identify matching send by the target process. Since the messages are sent out when active message call is made, the source process does not have to wait for completion of communication.

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 copies messages to receive buffer. It 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.

We will see in section 5.2 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 synchronization on the C side using POSIX mechanisms, but didn't have time to test this.

5. Benchmarks

The results of our benchmarks use an IBM SP3 running with four Power3 375MHz CPUs and 2GB of memory on each node. This machine uses AIX version 4.3 operating system and the IBM Developer Kit 1.3.1 (JIT) for the Java system. We are using the shared ``css0" adapter with User Space (US) communication mode for MPI setting and -O compiler command for Java. For comparison, we also have completed experiments for sequential Java, Fortran and HPF version of the HPJava programs. For the HPF version of program, it uses IBM XL HPF version 1.4 with *xlhpf95* compiler commend and -O3 and -qhot flag. And XL Fortran for AIX with -O5 flag is used for Fortran version.

Figure 6 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 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

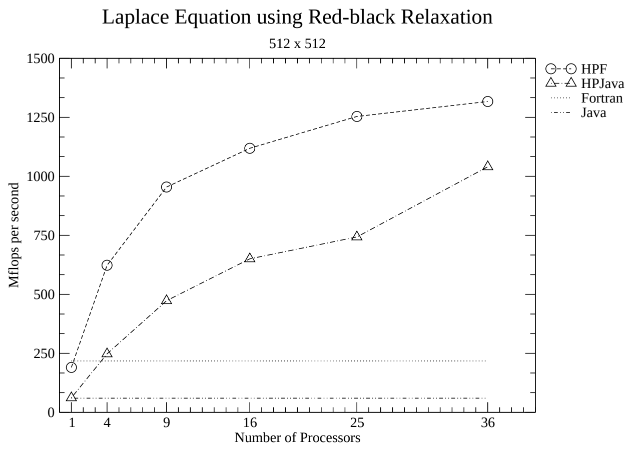

Figure 6: Red-black relaxation of two dimensional Laplace equation with size of 512 x 512

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

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

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 [3]. 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.

Speedup of HPJava for the Laplace equation is summarized in Table 1. Different sizes 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 example CFD which will be discussed in next section) will be more suitable for the parallel implementation than simple benchmarks described in this section.

5.1 HPJava with GUI

By adding pure Java version of the mpjdev to the Adlib communication library, it gives us the possibility to use

CFD				
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

Table 2: Speedup of CFD.

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 flow through an axisymmetric nozzle (Figure 7). 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. This demo also illustrates usage of Java object in our communication library. We are using **writeHalo()** method to communicate Java class object between threads. You can view this demonstration and source code at **http://www.hpjava.org/demo.html.**

For the performance test, we 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.

Speedup of HPJava is also summarized in Table 2. As we are 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.

5.2 LAPI

Figure 8 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

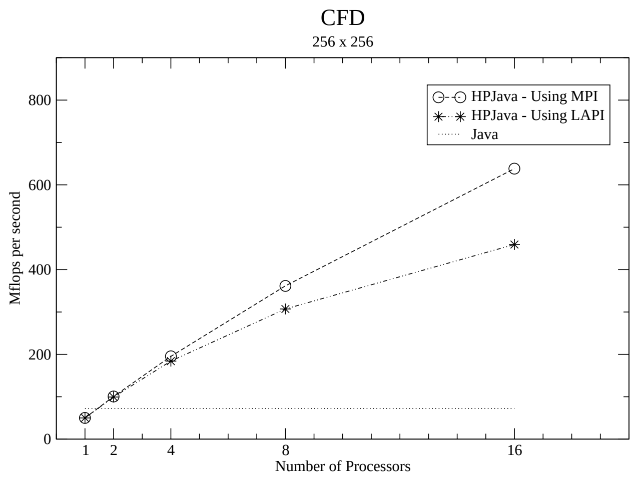

Figure 8: Comparison of the mpjdev communication library using MPI vs. LAPI.

Java Thread	POSIX Thread
57.49	10.68

Table 3: Timing for a wait and wake-up function calls on Java thread and POSIX thread in microseconds.

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 wake-up thread function calls (Table 3). 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 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 data, and thread waiting and awaking will be done in C. Implementation is a future project.

6. Conclusion

We have described how an underlying, low-level communication library for HPJava can be implemented in various platforms, plus some collective communication library primitives. We discussed format of a message and 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. As we mentioned in section 5.2, we would also like a better design for LAPI implementation of mpjdev to avoid the overheads of Java thread operation.

7. Acknowledgement

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

References

[1] B. Carpenter, G. Fox, H.-K. Lee, and S. Lim, Translation of the HPJava Language for Parallel Programming. *The 14th annual workshop on Languages and Compilers for Parallel Computing (LCPC2001),* May 2002.

[2] HPJava home page. http://www.hpjava.org.

[3] H.-K. Lee, B. Carpenter, G. Fox, and S. B. Lim, Benchmarking HPJava: Prospects for Performance. *Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR2002),* March 2002.

[4] S. B. Lim, B. Carpenter, G. Fox, and H.-K. Lee, Collective Communication for the HPJava Programming Language. *Concurrency and Computation: Practice and Experience*, 2003

[5] J. Moreira and S. Midkiff and M. Gupta, A comparison of three approaches to language, compiler and library support for multidimensional arrays in Java. *ACM 2001 Java Grande/ISCOPE Conference.* ACM Press, June 2001.

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

[7] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox, MPJ: MPI-like message passing for Java. *Concurrency: Practice and Experience*, *12*(11):1019—1038, 2000.

[8] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox, MPI for Java—Position Document and Draft API Specification. Technical Report JGF-TR-03, Java Grande Forum, November 1998.