Collective communication for the HPJava programming language

Sang Boem Lim1,2, Bryan Carpenter1, Geoffrey Fox1 and Han-ku Lee3,*†

1Pervasive Technology Labs, Indiana University, Bloomington, IN 47404, U.S.A.
2Florida State University, Tallahassee, FL 32306, U.S.A.
3School of Internet and Multimedia Engineering, Konkuk University, Seoul 143-701, Korea

SUMMARY
This paper addresses functionality and implementation of a HPJava version of the Adlib collective communication library for data parallel programming. We begin by illustrating typical use of the library, through an example multigrid application. Then we describe implementation issues for the high-level library. At a software engineering level, we illustrate how the primitives of the HPJava language assist in writing library methods whose implementation can be largely independent of the distribution format of the argument arrays. We also describe a low-level API called mpjdev, which handles basic communication underlying the Adlib implementation. Finally we present some benchmark results, and some conclusions. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: HPspmd programming model; HPJava; Adlib; Java

1. INTRODUCTION

HPJava [1–3] is the authors’ environment for parallel programming—especially data-parallel scientific programming—in Java. It includes a set of syntax extensions to Java for dealing with multi-dimensional distributed arrays, plus a set of communication libraries. The HPJava language has been described in several earlier papers. So has our message passing library mpiJava [4], which can be used for low-level single program, multiple data (SPMD) programming in HPJava. This paper will concentrate more on

*Correspondence to: Han-ku Lee, School of Internet and Multimedia Engineering, Konkuk University, Seoul 143-701, Korea.
†E-mail: hlee@konkuk.ac.kr

Contract/grant sponsor: National Science Foundation Division of Advanced Computational Infrastructure and Research; contract/grant number: 9872125

Received 16 January 2003
Revised 2 September 2003
Accepted 14 October 2003
the high-level collective library that is currently the preferred mechanism for communication in HPJava programs.

This library, called Adlib [5,6], was originally implemented in C++ and used in compilation of High Performance Fortran (HPF) programs. In HPJava it is made available as an application level library. In this paper we will illustrate how Adlib can be used as an application-level communication library in various example programs, and describe how it can naturally be implemented in terms of the primitives of the HPJava language. We also introduce a low-level Java messaging platform called mpjdev, which could potentially be used as a common API for implementing libraries like Adlib, mpiJava and their relatives.

Section 2 briefly reviews the HPJava language and includes a few illustrative programming fragments. As further motivation for use of this kind of collective communication in HPJava, Section 3 describes a larger scale multigrid application using HPJava and Adlib. Section 4 discusses various issues in the object-oriented Java implementation of the Adlib communication schedules. Section 5 describes features of the lower-level API mpjdev. Some benchmark results are discussed in Section 6. Conclusions are drawn together in Section 7.

1.1. Background and related work

HPJava implements a model for parallel programming which we call the HPspmd model. This is supposed to be a hybrid of HPF-like and MPI-like features.

The HPF language is about a decade old. To date it has not had the hoped-for impact on the practice of parallel programming. One possible reason for this is that it is difficult to write compilers for HPF: the compiler is responsible for details of parallelization, and insertion of communications, and it proved harder than expected to do these things automatically. Another possible reason may be that the programming model of HPF is quite rigid. For efficiency real programmers often want to break out of the ‘single-threaded’ semantics of HPF, and do lower-level programming. HPF allows this, but it is relatively clumsy.

Meanwhile the explicit SPMD programming style has been popular among developers of high-level parallel programming environments and libraries. Many notable libraries have been developed for the SPMD framework, including Kelp [7], CHAOS/PARTI [8], and Global Arrays [9]. These typically provide high-level collective or one-sided communication primitives for operating on distributed data. In other libraries, all aspects of data localization and transfer are managed by the library. It is left to the library to work out whether or not a communication is implied. In effect the library is operating at the same level as a language like HPF. One such library is A++/P++ [10].

While the library-based SPMD approach to data-parallel programming is quite popular and successful, it misses out on the uniformity and elegance promised by HPF. There are no compile-time or compiler-generated checks on the use of distributed arrays because libraries manage these arrays. The HPspmd model implemented by HPJava attempts to address some of these issues. It is a model of explicit SPMD programming supported by specific syntax for representing HPF-like distributed arrays. The programmer is responsible for explicitly specifying communications through suitable libraries; but the compiler can do various sanity checks on the usage of distributed arrays. Further details of how this works follow later. The benefits we claim for the HPspmd approach are: compilers or translators for
HPJava are much easier to write than for HPF; specification of communication libraries is cleaner than for SPMD libraries without any language support for distributed arrays; and the language is flexible—it allows one to put efficient, low-level SPMD code ‘in-line’ at any point.

Because of the way the distribution format of arrays is represented in the HPspmd model, it is convenient to use an object-oriented language like Java as base language. Many other arguments for use of Java can be found elsewhere in this issue.

2. THE HPJAVA LANGUAGE

HPJava is an environment for parallel programming, especially suitable for programming massively parallel, distributed memory computers. HPJava is a strict extension of Java—it incorporates all of Java as a subset. For dealing with distributed arrays, it adds some pre-defined classes and some extra syntax. Through the HPspmd programming model of HPJava, we aim to provide a hybrid of the data parallel and the low-level SPMD approaches. Therefore HPF-like distributed arrays appear as language primitives.

In the spirit of distributed memory SPMD programming, a design decision is made that all access to non-local array elements should go through calls to library functions in the source program. These library calls must be placed in the original HPJava program by the programmer. This requirement may seem strange to people expecting to program in high-level parallel languages like HPF, but it should not seem unnatural to programmers accustomed to writing parallel programs with MPI or other SPMD libraries. The exact communication library used is not part of the HPJava language design. An appropriate communication library might perform collective operations on whole distributed arrays (like the one described in this paper), or it might provide some kind of get and put functions for access to remote blocks of a distributed array, similar to the ones provided in the Global Array Toolkit [9], say.

A subscripting syntax can be used to directly access local elements of distributed arrays. A well-defined set of rules—automatically checked by the HPJava translator‡—ensures that references to these elements can only be made by processes that hold copies of the elements concerned.

Figure 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 which are named through ‘grids’ objects. The class Proc2 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 Proc2 extends the special base class Group which has a privileged status in the HPJava language. An object that inherits from 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.

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 to the constructor of BlockRange represent the total

‡The HPJava extended syntax is currently implemented by providing a translator from the extended language to standard Java.
size of the range and the target process dimension. Thus, a `Range` instance `x` has `M` elements distributed over the first dimension of `p` (`p.dim(0)`) and `y` has `N` elements distributed over the second dimension of `p` (`p.dim(1)`).

The variables `a`, `b`, and `c` are all `distributed array` variables. The distributed array is the most important feature HPJava adds to Java. A distributed array is a collective object 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 a `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, multi-dimensional array—a valuable adjunct to Java in its own right, as many people have noted [11,12]).

The constructors 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`.

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`.

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.

Figure 2 is a HPJava program for the Laplace program that uses ghost regions. It illustrates the use of the standard library class `ExtBlockRange` to create arrays with ghost extensions. The distributed range class `ExtBlockRange` is a library class derived from the special class `Range`, distributing
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 [x, y] a = new float [x, y] ;
        ... initialize edge values in 'a'
        float [x, y] 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 the Laplace equation by Jacobi relaxation.

with block distribution format with ghost extensions. In this case, the extensions are of width 1 on
either side of the locally held ‘physical’ segment. Figure 3 illustrates this situation.

From the point of view of this paper the most important feature of this example is the appearance
of the function Adlib.writeHalo(). This is a collective communication operation 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 4 illustrates the effect of executing the writeHalo
function. 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.

The final component of the basic HPJava syntax that we will discuss here is support for 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 subsequently be accessed either through the parent
array or through the array section—HPJava sections behave something like array pointers in Fortran,
Figure 3. Example of a distributed array with ghost regions.

Figure 4. Illustration of the effect of executing the writeHalo function.
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 also be used in array 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 paper.

3. AN APPLICATION

In the last section we introduced the basic HPJava syntax with a couple of simple examples. 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 [13]. The whole of this program has been ported to HPJava (it is about 800 lines), but in this article we will only consider a few critical routines. The point is to further illustrate the power of HPJava collective communication operations. We will see that writeHalo and one other operation (remap) suffice to code our routines compactly and quite efficiently, provided these operations can operate on arbitrary distributed arrays, including distributed array sections. In the next section we will discuss implementation of the collectives.

The Multigrid [14] 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 term 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 5, 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. The post-relaxation phase performs 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 main parts of a solution of two-dimensional Poisson equation using a multigrid method. Domain decomposition, which assigns part of the grid to each process, is used for parallelization. Boundary values of the subgrids are exchanged with nearest neighbors after each calculation. This operation is illustrated in Figure 6. In this red-black relaxation, the values of the subgrids boundaries are exchanged by using the writeHalo method. We assume that all distributed arrays in our examples were created with ghost regions.

The implementation makes use of the expression \(\hat{i}\) (read as ‘i-primed’). This is new syntax. In HPJava distributed indexes like \(i\) are a special kind of symbol—they do not in themselves have a Java value associated with them. To get the integer global loop index associated with the distributed index, the ‘primed’ operator must be applied. In this example this value is used to compute the lower
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 5. An example of multigrid iteration.

Figure 6. Red-black relaxation on array uf.

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 moreover this
parity is swapped in successive iterations.

The restrict operation (Figure 7) computes the residual and restricts it to a coarser grid. The residual
is computed only at points of fine grid with matching points in the course 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.

The important remap operation allows one to copy one distributed array to another of the same
shape which may have unrelated distribution format. Note that we have made no assumptions of any
relationship between the distribution of the fine grid array uf and the coarse grid array fc. In our code
a subset of the elements of the work-array tf is copied into a subset of the array in the coarse grid.
The behavior of the restrict operation is illustrated in Figure 8.
static void restr(int npc, int npf, double fc[[-,-]], double uf[[-,-]],
        double ff[[-,-]], double tf[[-,-]]) {

    Range xf = ff.rng(0), ff = ff.rng(1);
    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 7. The restrict operation.

Figure 8. Illustration of the restrict operation.

The interpolate operation (Figure 9) is opposite of the 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 the correction, uc, from the coarse grid to matching points of a work-array, tf, in the fine grid. After the copying, we need to update boundary values using writeHalo to get most up-to-date values. By the two nested overall constructs, it corrects current values of the 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 on vertical links—this is sufficient because it turns out that the correction is only needed on fine grid sites of a single parity. The behavior of the interpolate operation is illustrated in Figure 10.
static void interp(int npf, double[][] uc,
            double[][] uf, double [][] tf) {
    Range xf = uf.rng(0), yf = uf.rng(1);
    int nf = npf - 1;
    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 9. The interpolate operation.

Figure 10. Illustration of the interpolate operation.

The examples here rely on the two high-level Adlib communication schedules that deal explicitly with distributed arrays: the remap and writeHalo methods. The remap operation can be applied to various ranks and types 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 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 also summarize these in the shorthand interface:

```java
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 (i.e. libraries implemented in standard Java). This more concise signature does not incorporate the constraint that dst and src have the same rank—that would have to be tested at run-time.)

Adlib was developed as a C++ library to support HPF translation in the Parallel Compiler Runtime Consortium (PCRC) [6] and earlier projects [5]. Early versions of HPJava used a Java Nature Interface (JNI) wrapper interface to the C++ kernel of the PCRC library. The new version described here is pure Java, and is extended it to support Java object types, and to target Java communication platforms.

Besides remap and writeHalo, Adlib includes a family of related regular collective communication operations (shifts, skews, transposes, and so on). It incorporates 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.

4. 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. For example, it is essentially equivalent to the operation called Regular Section Copy Sched in [15]. In this section we want to illustrate how this kind of operation can be implemented in terms of the particular Range and Group hierarchies of HPJava (complemented by a 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 perform 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 paradigm was pioneered in the CHAOS/PARTI libraries [8]. If a communication pattern is to be executed only once, simple wrapper functions can be made available to construct a schedule, execute it, and 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 associated data structures are just those associated with schedule construction.

The constructor and public method of the remap schedule for distributed arrays of float element can be described as follows:
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 11. API of the class BlockMessSchedule.

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

The remap schedule combines two functionalities: it reorganizes data in the way indicated by the distribution formats of the source and destination arrays. 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 lightweight 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 11. 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().

The implementation details of BlockMessSchedule will not be discussed in greater detail here because they are not particularly specific to our HPJava system, and the principles are fairly well known (see, for example, [15]).

However, we do wish to describe in a little 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 by each process. 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 is illustrated in Figure 12, and some of the relevant virtual functions are displayed in the API of Figure 13. The most relevant methods optionally take arguments that allow one to specify a contiguous or strided subrange of interest. The `Triplet` and `Block` classes represent simple struct-like objects holding a few `int` fields 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 subroutines called `sendLoop` and `recvLoop` that enumerate messages to be sent and received, respectively. Figure 14 sketches the implementation of `sendLoop`. This is a recursive function—it implements a multi-dimensional loop over the `rank dimensions` of the arrays. It is initially called with `r = 0`. There is little point going into full detail of the algorithm here, but 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 15 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 class is invoked.
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 13. Partial API of the class Range.

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.subbas;
        int step = src.str(r) * loc.substp;
        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 14. The sendLoop method for Remap.
The variables src and dst represent the distributed array arguments. The inquiries rng() and grp() extract the range and group objects of these arrays.

For further information on the Adlib implementation, we refer the reader to the open source of the library—available from www.hpjava.org—and to [16].

Of the collective communication schedules currently implemented in Adlib (see next subsection), all except WriteHalo share with Remap this property that their implementation code does not explicitly depend on the distribution format of the arrays\(^8\). All rely heavily on the methods and inquiries of the Range and Group classes, which abstract the distribution format of arrays. The specific API of the Range and Group classes has evolved through C++ and Java versions of Adlib over a fairly lengthy period of development. Some other distributed array libraries (e.g., A++/P++ [10]) provide broadly comparable APIs for accessing layout information, but our specific factorization of this information into distributed range and group objects is peculiar. It compactly supports the distribution formats considered by the High Performance Fortran Forum.

In the HPJava version currently under development, 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 next section. To perform the actual communication and to deal with preparation of the data, it uses methods of the mpjdev like isend(), irecv(), read(), write(), strGather(), and strScatter(). The isend() and irecv() are used for actual communication. 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() and write()) deal with a contiguous data and the other two (strGather() and strScatter()) are dealing with non-contiguous data.

\(^8\)The schedule WriteHalo depends on the presence ghost regions, which are only useful for block-like distribution formats. This introduces some explicit dependency on distribution format into the implementation.
4.1. Other schedules in Adlib

We have already described two characteristic examples of the regular communications, remap() and writeHalo(). In this subsection we summarize other collective communication schedules in Adlib.

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 signature

\[
\text{void shift}(T \ # \ destination, \ T \ # \ source, \\
\quad \text{int shiftAmount, int dimension})
\]

where the variable \( T \) runs over all primitive types and \text{Object}, and the notation \( T \ # \) means a multi-array of arbitrary rank, with elements of type \( T \). The \text{shiftAmount} argument, which may be negative, specifies the amount and direction of the shift. The \text{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 function broadcast() 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 a rank-zero 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. 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 \( \text{root} \). It is broadcast to all members of the active process group.

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 that reduction operations do not support Java Object type.

The \text{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 \text{boolean}. The result is broadcast to the active process group, and returned by the function. Other reduction operations with similar interfaces are \text{minval()}, \text{sum()}, and \text{product()}. Of these \text{minval()} is minimum value, \text{sum()} adds the elements of \( a \) in an unspecified order, and \text{product()} multiplies them.

The \text{dotProduct()} operation 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 need not be identical. If they are different, standard Java binary numeric promotions are applied—for example, if the dot product of an \text{int} array with a \text{float} array is a \text{float} value. The prototypes are

\[
t_3 \ dotProduct(t_1 \ # \ a, \ t_2 \ # \ b)
\]
If either \( t_1 \) 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.

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

\[
\texttt{void gather(T \[\ldots\] destination, T \[\ldots\] source, int \[\ldots\] subscripts)} ;
\]

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

\[
\begin{align*}
\text{for all } & i \text{ in } \{0,\ldots,N-1\} \text{ in parallel do} \\
\text{destination}[i] & = \text{source}[\text{subscripts}[i]] ;
\end{align*}
\]

where \( N \) is the size of the \texttt{destination} (and \texttt{subscripts}) array.

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

\[
\texttt{void scatter(T \[\ldots\] source, T \[\ldots\] destination, int \[\ldots\] subscripts)} ;
\]

and it behaves like

\[
\begin{align*}
\text{for all } & i \text{ in } \{0,\ldots,N-1\} \text{ in parallel do} \\
\text{destination}[\text{subscripts}[i]] & = \text{source}[i] ;
\end{align*}
\]

Higher dimensional arrays (with additional \texttt{subscript} arguments) are also supported. Currently the HPIJava version of Adlib does not support combining scatters, although these could be added in later releases.

The complete API of Adlib is described in [16].

5. mpjdev

The mpjdev API is designed with the goal that it can be implemented \textit{portably} on network platforms and \textit{efficiently} on parallel hardware. Unlike MPI which is intended for the application developer, mpjdev is meant for library developers. The mpjdev API itself might be implemented on top of Java sockets in a portable network implementation, or—on HPC platforms—through a JNI to a subset of MPI. The positioning of the mpjdev API is illustrated in Figure 16. Currently not all the communication stack in this figure is implemented. The Java version of Adlib, the pure Java implementation on SMPs, and the native MPI implementation are developed and included in the current HPJava or mpiJava releases. The rest of the stack may be filled in the future.

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 JNI as low as practical. From the development of our earlier successful library mpiJava, we learned that communication overheads are a key factor of performance. For the mpjdev library, one important decision is made to reduce communication overhead. Usually communication
protocols are type specific—different types of data should be sent separately. To avoid many small sends, we maintain all the data of the mpjdev as the Java byte [] array or C char [] array depending on the implementation. 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 a 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.

5.1. General API

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 datatypes 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 datatypes of MPI were designed to facilitate. Avoiding internal copying of message buffers would require changes to the implementation of some of the most popular JVMs. This is outside our control.

Therefore 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 operations to better support HPC applications. Much of the complexity in the mpjdev API is then associated with packing and unpacking.

---

In MPI, the type or types of data held in a message buffer are described by special opaque objects called datatypes. A compound ‘derived datatype’ object can describe not only the conventional type of component data items, but also their relative offsets in the memory. This allows datatypes to describe data scattered through memory. Then, if the communication hardware supports it, non-contiguous data can then be gathered directly from memory to the network device (or vice versa) without copying to an intermediate buffer.
public class Buffer {
    public static final int SECTION_OVERHEAD = 8;
    public void ensureCapacity(int newCapacity) { ... }
    public void restoreCapacity() { ... }
    public void free() { ... }
}

Figure 17. The public interface of Buffer class.

of message buffers. Because we want to make sure that usually data need only 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.

5.1.1. 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 17. 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 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 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 5.2.

5.1.2. Communication API

The currently specified API for mpjdev is small compared to MPI. It only includes point-to-point communications. Currently the only messaging modes for mpjdev are the standard blocking mode (like MPI_SEND, MPI_RECV) and the standard non-blocking mode (like MPI_ISEND, MPI_IRECV), together with a couple of ‘wait’ primitives.
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 18. The public interface of mpjdev Comm class.

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 18 shows the public interface of Comm class. We can get the number of processes that are spanned by this communicator by calling size(). The current id of the process relative to this communicator is returned by id().

The two methods send() and recv() are blocking communication modes. These two methods block until the communication finishes. The functionalities of send() and recv() methods are same as the 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 has been safely stored away.

The other two communication methods, isend() and irecv(), are non-blocking versions of send() and recv. These are equivalent to MPI_ISEND and MPI_I_RECV in MPI. Unlike a blocking send, a non-blocking send returns immediately after its call and does not wait for completion. A non-blocking receive also work similarly.

The method dup() creates a new communicator 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 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.
5.2. Message format

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. 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.

The primary buffer, which is used to store message elements of primitive type, and the secondary buffer, which is intended to hold the data from object elements in the message, are two main parts of a message. The primary buffer has a fixed capacity. The size of the secondary buffer, if it is used, is likely to be determined ‘dynamically’—for example, as objects are written to the buffer.

The primary buffer contains two main parts. A message starts with a short primary header followed by primary buffer contents (sections). The primary header defines the encoding scheme and total number of data bytes in the primary buffer. 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 have Object type. The secondary header of the secondary buffer will only store information about the number of bytes of data in the secondary buffer. The layout of the secondary buffer will be defined by the Java Object Serialization specification. Figure 19 illustrates the overall layout of the logical message.

By calling the \texttt{free()} method, we can destroy this communicator (like \texttt{MPI\_COMM\_FREE} in MPI). This method is usually called when this communicator is no longer in use. It frees any resources that are used by this communicator.

We should call static \texttt{init()} method once before calling any other methods in the communicator. This static method initializes \texttt{mpjdev} and makes it ready for use. The static method \texttt{finish()} (which is equivalent to \texttt{MPI\_FINALIZE}) is the last method that should be called in \texttt{mpjdev}.
Figure 20. Red-black relaxation of the two-dimensional Laplace equation with size 512².

While it supports transmission of serialized objects, this layout is optimized for communication of the kind of arrays of primitive elements that are common in scientific computation. This is in contrast to, say, Java Remote Method Invocation (RMI), which incurs the overhead of standard object serialization in handling all datatypes—even arrays of primitive elements.

The design of the mpjdev layer owes much to our earlier experiences with mpiJava [4]. In mpiJava we originally tried to simply wrap the native MPI API—derived datatypes and all—in Java, using JNI. This can be made to work, and on suitable platforms mpiJava can achieve the zero-copying goal. However, portability remains a problem, and on many Java platforms mpiJava ends up copying buffers like mpjdev anyway.

6. BENCHMARKS

The results of our benchmarks use an IBM SP3 running with four Power3 375 MHz CPUs and 2 GB of memory on each node. This machine uses the 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. We use the -O compiler option for javac, with no non-default options on the JVM itself (javac command). For comparison, we have also completed experiments for sequential Java, Fortran, and the HPF version of the HPJava programs. The sequential results appear on the following figures as horizontal lines, to give some feeling for the sequential ‘baseline’ performance. For the HPF version of the program, it uses IBM XL HPF version 1.4 with xhpf95 compiler command and -O3 and -qhot flag. XL Fortran for AIX with -O5 flag is used for the Fortran version.

All the benchmark results presented are best results over several runs. Runs are timed after some ‘warm-up’ iterations, to give the JIT a chance to compile the code. Timings use the standard millisecond
A single SP3 node has four processors; we are using one node for jobs with one to four processes, two nodes for jobs with five to eight processes, and so on. We are working in dedicated mode, which means other jobs are not running on the same node. Normally there is one JVM per SPMD process, which is the unit of parallelism. In other words we are not intentionally using Java thread parallelism within a node\(^1\).

Figure 20 shows the performance results of four different versions (HPJava, sequential Java, HPF, and Fortran) of red-black relaxation of the two-dimensional Laplace equation with size $512^2$. We also show the speedup of HPJava and HPF in Figure 21. In our runs HPJava can out-perform sequential Java by a factor of 17. On 36 processors HPJava can get about 78% of the performance of HPF. This is not a 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 [3]. The scaling behavior of HPJava is slightly better than HPF. This probably 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. However, clearly future versions of Adlib could be optimized for the platform. We see a similar behavior on the large size three-dimensional diffusion equation benchmark (Figures 22 and 23). In general, we expect that three-dimensional problems will be more amenable to parallelism, because of the large problem size.

\(^1\)Although, as pointed out by one of the referees, it could happen that, if we happen to have fewer than four JVMs in a node, some background threads like garbage collection might run on idle processors—a kind of accidental parallelism. We do not see any specific evidence for this.
Figure 22. The three-dimensional diffusion equation with size $128^3$.

Figure 23. Speedup of the three-dimensional diffusion equation with size $128^3$. 
On the small size three-dimensional diffusion equation benchmark (Figures 24 and 25), we can see that the speed of sequential Fortran is about four to five times faster than Java. This may be representative of the IBM platform, but note that on other, probably more widely-installed, platforms (to be specific, Linux platforms) a factor of two or less between Java and Fortran timings is common (see [17]).
Finally, we consider benchmark results for our original problem, the multigrid solver, in Figures 26 and 27. For the complete multigrid algorithm we currently get slightly disappointing speedup and absolute performance. These results are new at the time of writing, and 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 both.
Table I. Speedup of benchmarks as compared with one processor.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Processors</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multigrid solver</td>
<td>HPJava—512</td>
<td>1.90</td>
<td>2.29</td>
<td>2.39</td>
<td>2.96</td>
<td>3.03</td>
</tr>
<tr>
<td></td>
<td>HPF—512</td>
<td>1.42</td>
<td>1.52</td>
<td>1.70</td>
<td>1.98</td>
<td>2.21</td>
</tr>
<tr>
<td>Two-dimensional Laplace equation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processors</td>
<td></td>
<td>4</td>
<td>9</td>
<td>16</td>
<td>25</td>
<td>36</td>
</tr>
<tr>
<td>HPJava—256</td>
<td></td>
<td>2.67</td>
<td>3.73</td>
<td>4.68</td>
<td>6.23</td>
<td>6.23</td>
</tr>
<tr>
<td>HPF—256</td>
<td></td>
<td>1.35</td>
<td>1.58</td>
<td>1.64</td>
<td>1.69</td>
<td>1.57</td>
</tr>
<tr>
<td>HPJava—512</td>
<td></td>
<td>4.06</td>
<td>7.75</td>
<td>9.47</td>
<td>12.18</td>
<td>17.04</td>
</tr>
<tr>
<td>HPF—512</td>
<td></td>
<td>3.28</td>
<td>5.09</td>
<td>5.87</td>
<td>6.51</td>
<td>7.01</td>
</tr>
<tr>
<td>HPJava—1024</td>
<td></td>
<td>3.64</td>
<td>13.05</td>
<td>18.11</td>
<td>23.99</td>
<td>27.27</td>
</tr>
<tr>
<td>Three-dimensional diffusion equation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processors</td>
<td></td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>HPJava—32</td>
<td></td>
<td>2.72</td>
<td>3.45</td>
<td>4.75</td>
<td>5.43</td>
<td></td>
</tr>
<tr>
<td>HPF—32</td>
<td></td>
<td>1.02</td>
<td>1.05</td>
<td>1.57</td>
<td>1.81</td>
<td></td>
</tr>
<tr>
<td>HPJava—64</td>
<td></td>
<td>3.00</td>
<td>4.85</td>
<td>7.47</td>
<td>8.92</td>
<td></td>
</tr>
<tr>
<td>HPF—64</td>
<td></td>
<td>1.84</td>
<td>1.94</td>
<td>2.50</td>
<td>3.12</td>
<td></td>
</tr>
<tr>
<td>HPJava—128</td>
<td></td>
<td>3.31</td>
<td>5.76</td>
<td>9.98</td>
<td>13.88</td>
<td></td>
</tr>
<tr>
<td>HPF—128</td>
<td></td>
<td>1.02</td>
<td>1.05</td>
<td>1.57</td>
<td>1.81</td>
<td></td>
</tr>
</tbody>
</table>
| The speedup of HPJava is summarized in Table I. Problems of different size 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 26.77 times speedup on the Laplace equation using 36 processors with a problem size of 1024^2. Many realistic applications with more computation for each grid point (for example, CFD) will be more suitable for the parallel implementation than the Laplace equation and similar simple benchmarks described here. Many such algorithms will be equally amenable to implementation in HPJava—see, for example, the CFD demo at http://www.hpjava.org.

7. DISCUSSION

We have described how a collective communication library for HPJava can be used for various parallel applications, and how the library itself can be implemented naturally in terms of the HPJava language primitives, plus an underlying, low-level set of messaging primitives called mpjdev.

Some benchmark results were given. We get reasonable performance for the simple problems like Laplace equation with the initial HPJava implementation. Results for the multigrid solver indicate
further optimization for the HPJava translation scheme and Adlib library is required. This work is in progress.

A subset of the Adlib library has been available as a JNI wrapper to the C++ version for a couple of years. We have now finished the task of converting the whole library to Java. The underlying mpjdev library is available in a form that runs on top of a native MPI implementation, using JNI. There is also a version of mpjdev available that runs in multithreaded SPMD mode in a single JVM, using Java thread primitives for communication.

The complete HPJava system, including all the software described here, is available as a free download from http://www.hpjava.org.

ACKNOWLEDGEMENT

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

REFERENCES
