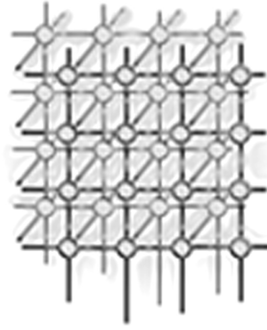
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2000; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*



Collective Communication for the HPJava Programming Language

Sang Boem Lim[1,2,*,†], Bryan Carpenter[1†], Geoffrey Fox[1†], and Han-ku Lee[1,2†]

[1] *Pervasive Technology Labs at Indiana University Bloomington, IN 47404*
[2] *Florida State University Tallahassee, FL 32306*

SUMMARY

This paper addresses functionality and implementation of an *HPJava* version of the *Adlib* collective communication library for data parallel programming. It begins 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 © 2003 John Wiley & Sons, Ltd.**

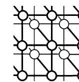
KEY WORDS: HPspmd Programming Model, HPJava, Adlib, Java.

1. Introduction

HPJava [5, 9, 10] 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 [6], which can be used for low-level SPMD programming in HPJava. This

*Correspondence to: Sang Lim, Pervasive Technology Labs at Indiana University, Bloomington, IN 47404
†E-mail: {sblim, dbcarpen, gcf, hanklee}@indiana.edu
Contract/grant sponsor: National Science Foundation Division of Advanced Computational Infrastructure and Research, contract number; contract/grant number: 9872125

Copyright © 2000 John Wiley & Sons, Ltd.
Received 15 January 2003
Revised 1 September 2003



paper will concentrate more on the high-level collective library that is currently the preferred mechanism for communication in HPJava programs.

This library, called Adlib [12, 17], 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 of 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 High Performance Fortran 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 be developed for the SPMD framework, including Kelp [8], CHAOS/PARTI [7] and Global Arrays [15]. 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 from the user. The user just specifies the distribution format of arrays when they are declared, then tells the library to do particular operations on particular distributed arrays. It is left to the library to work out whether or not a communication is implied. In effect the library is operating at the same level as a language like HPF. One such library is A++/P++ [16].

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 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 explicity specifying communications through suitable libraries; but the compiler can do various sanity

Copyright © 2000 John Wiley & Sons, Ltd.

checks on 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 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

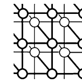
HPJava is an environment for parallel programming, especially suitable for programming massively parallel, distributed memory computers. HPJava is a strict extension of Java—it incorporates all of Java as a subset. For dealing with distributed arrays, it adds some predefined classes and some extra syntax. Through the *HPspmd* programming model of HPJava, we aim to provide a hybrid of the data parallel and the low-level SPMD (Single Program Multiple Data) approaches. So 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 be 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 [15], 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 `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` 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

[†]The HPJava extended syntax is currently implemented by providing a translator from the extended language to standard Java.



```
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. Matrix addition using HPJava.

`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 size of the range and the target process dimension. Thus, x has `M` elements and distributed over first dimension of p and y has `N` elements and distributed over second dimension of p.

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 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 adjunct to Java in its own right, as many people have noted [14, 13]).

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`.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

```
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);
}
```

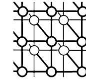
Figure 2. Solution of Laplace equation by Jacobi relaxation.

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 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

Copyright © 2000 John Wiley & Sons, Ltd.



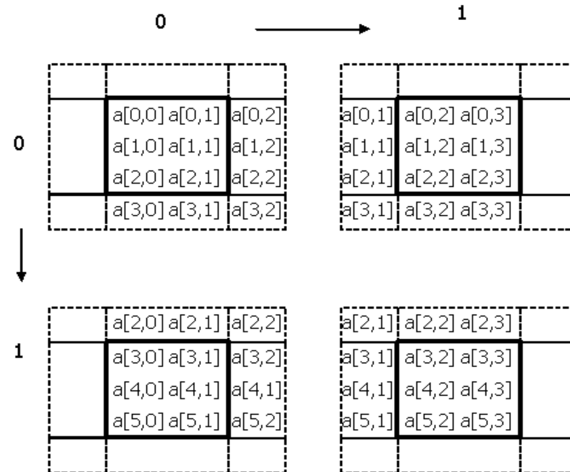


Figure 3. Example of a distributed array with ghost regions.

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, 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 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.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

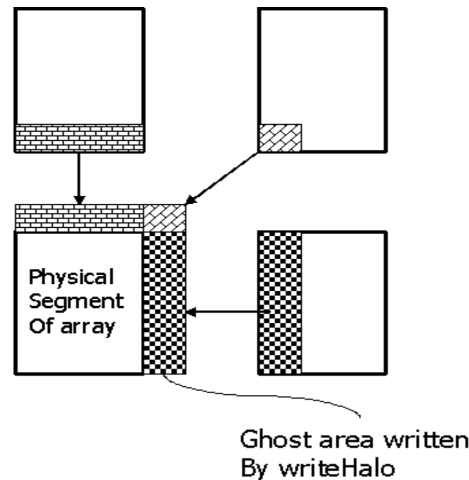


Figure 4. Illustration of the effect of executing the writeHalo function.

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 a existing Fortran program (called PDE2), taken from the Genesis parallel benchmark suite [1]. 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* [3] 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.

Copyright © 2000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls