« SUPE 11227 layout: Small Extended reference style: basic file: supe125.tex (GIT) aid: 125 doctopic: OriginalPaper class: spr-small-v1 v.2007/02/23 Prn:26/02/2007; 8:00 p. 1»

J Supercomput
DOI 10.1007/s11227-007-0125-5

Runtime support for scable programming in Java

Sang Boem Lim · Bryan Carpenter · Geoffrey Fox · Han-Ku Lee

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

The paper will discuss in detail a particular environment called *HPJava*. HPJava is the environment for parallel programming—especially data-parallel scientific programming—in Java. Our HPJava is based around a small set of language extensions designed to support parallel computation with distributed arrays, plus a set of communication libraries. A high-level communication API, *Adlib*, is developed as an application level communication library suitable for our HPJava. This communication library supports *collective operations* on distributed arrays. We include Java `Object` as one of the Adlib communication data types. So we fully support communication of intrinsic Java types, including primitive types, and Java object types.

S.B. Lim (✉)
Department of Advanced Technology Fusion, Konkuk University, Seoul, Republic of Korea
e-mail: slim@kisti.re.kr

B. Carpenter
OMII, University of Southampton, Southampton SO17 1BJ, UK
e-mail: dbc@ecs.soton.ac.uk

G. Fox
Pervasive Technology Labs at Indiana University, Bloomington, IN 47404-3730, USA
e-mail: gcf@indiana.edu

H.-K. Lee
School of Internet and Multimedia Engineering, Konkuk University, Seoul, Republic of Korea
e-mail: hlee@konkuk.ac.kr

Journal ID: 11227, Article ID: 125, Date: 2007-02-26, Proof No: 1

Keywords ???

1 Introduction

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

Amongst *computational* scientists Java may well become a very attractive language to create new programming environments that combine powerful object-oriented technology with potentially high performance computing. The popularity of Java has led to it being seriously considered as a good language to develop scientific and engineering applications, and in particular for parallel computing [1, 6, 7]. Sun's claims on behalf of Java, that is simple, efficient and platform-natural—a natural language for network programming—make it attractive to scientific programmers who wish to harness the collective computational power of parallel platforms as well as networks of workstations or PCs, with interconnections ranging from LANs to the Internet. This role for Java is being encouraged by bodies like Java Grande [8].

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

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

2 Related works

UC Berkeley is developing Titanium [13] to add a comprehensive set of parallel extensions to the Java language. Support for a shared address space and compile-time analysis of patterns of synchronization is supported.

The Timber [12] project is developed from Delft University of Technology. It extends Java with the Spar primitives for scientific programming, which include multidimensional arrays and tuples. It also adds task parallel constructs like a foreach construct.

Jade [5] from University of Illinois at Urbana-Champaign focuses on message-driven parallelism extracted from interactions between a special kind of distributed

« SUPE 11227 layout: Small Extended reference style: basic file: supe125.tex (GIT) aid: 125 doctopic: OriginalPaper class: spr-small-v1 v.2007/02/23 Prn:26/02/2007; 8:00 p. 3»

object called a Chare. It introduces a kind of parallel array called a ChareArray. Jade also supports code migration.

HPJava differs from these projects in emphasizing a lower-level (MPI-like) approach to parallelism and communication, and by importing HPF-like distribution formats for arrays. Another significant difference between HPJava and the other systems mentioned above is that HPJava translates to Java byte codes, relying on clusters of conventional JVMs for execution. The systems mentioned above typically translate to C or C++. While HPJava may pay some price in performance for this approach, it tends to be more fully compliant with the standard Java platform.

3 Features of HPJava

HPJava is a strict extension of its base language, Java, adding some predefined classes and some extra syntax for dealing with distributed arrays. HPJava is thus an environment for parallel programming, especially suitable for data parallel scientific programming. An HPJava program can freely invoke any existing Java classes without restrictions because it incorporates all of Java as a subset.

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 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

```
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] ;
}
```

Fig. 1 A parallel matrix addition

Journal ID: 11227, Article ID: 125, Date: 2007-02-26, Proof No: 1

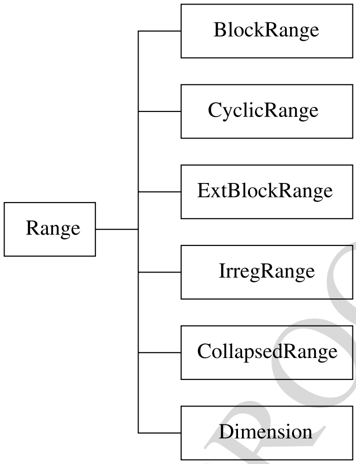


Fig. 2 The HPJava range hierarchy

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 [10, 11]).

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

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

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

HPJava defines a class hierarchy of different kinds of range object (Fig. 2). Each subclass represents a different kind of distribution format for an array dimension. The simplest distribution format is *collapsed* (sequential) format in which the whole of the array dimension is mapped to the local process. Other distribution formats (motivated by High Performance Fortran) include *regular block* decomposition, and *simple cyclic* decomposition. In these cases the index range (thus array dimension) is distributed over one of the dimensions of the process grid defined by the group object. All ranges must be distributed over different dimensions of this grid, and if a particular dimension of the grid is targeted by none of the ranges, the array is said to be *replicated* in that dimension.[1] Some of the range classes allow *ghost extensions* to support stencil-based computations.

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

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

4 Usage of high-level communication library

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

The general purpose matrix multiplication routine (Fig. 3) has two temporary arrays `ta`, `tb` with the desired distributed format. This program is also using information which is defined for any distributed array: `grp()` to fetch the distribution group and `rng()` to fetch the index ranges.

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

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

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

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

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

```
public void matmul(float [[-,-]] c, float [[-,-]] a, float [[-,-]] b) {
  Group2 p = c.grp();
  Range x = c.rng(0); Range y = c.rng(1);

  int N = a.rng(1).size();
  float [[-,*]] ta = new float [[x, N]] on p;
  float [[*,-]] tb = new float [[N, y]] on p;

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

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

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

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

Fig. 3 A general Matrix multiplication in HPJava

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 has to be tested at run-time.)

As another example, Fig. 4 is a HPJava program for the Laplace program that uses *ghost regions*. It illustrates the use the library class `ExtBlockRange` to create arrays with ghost extensions. In this case, the extensions are of width 1 on either side of the locally held "physical" segment. Figure 5 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*. This particular one is used to fill the *ghost cells* or *overlap regions* surrounding the "physical segment" of a distributed array. A call to a collective operation must be invoked simultaneously by all members of some active process group (which may or may not be the entire set of processes executing the program). The effect of `writeHalo` is to overwrite the ghost region with values from processes holding the corresponding elements in their physical segments. Figure 6 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.

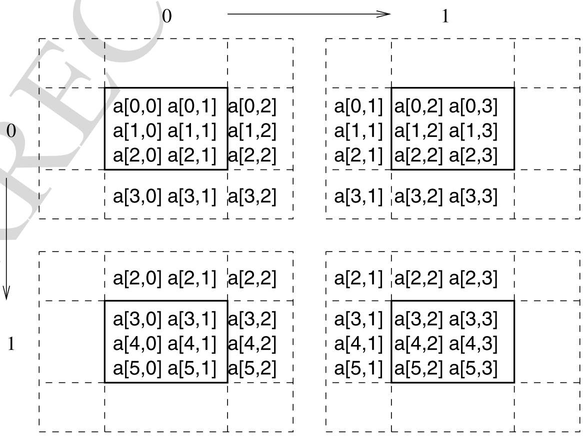
We will discuss implementation issues of high-level communication libraries in following section.

« SUPE 11227 layout: Small Extended reference style: basic file: supe125.tex (GIT) aid: 125 doctopic: OriginalPaper class: spr-small-v1 v.2007/02/23 Prn:26/02/2007; 8:00 p. 7»


```
    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);
    }
```

Fig. 4 Solution of Laplace equation by Jacobi relaxation

Fig. 5 Example of a distributed array with ghost regions

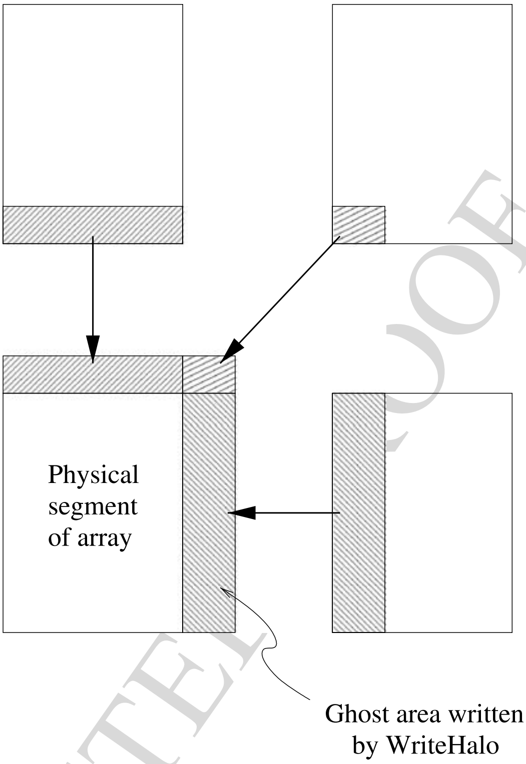


5 Implementation of collectives

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.

Journal ID: 11227, Article ID: 125, Date: 2007-02-26, Proof No: 1

Fig. 6 Illustration of the effect of executing the writeHalo function



General algorithms for this primitive have been described by other authors in the past. For example it is essentially equivalent to the operation called *Regular_Section_Copy_Sched* in [2]. In this section we want to illustrate how this kind of operation can be implemented in term of the particular `Range` and `Group` classes of HPJava, complemented by suitable set of messaging primitives.

All collective operations in the library are based on communication *schedule* objects. Each kind of operation has an associated class of schedules. Particular instances of these schedules, involving particular data arrays and other parameters, are created by the class constructors. Executing a schedule initiates the communications required to effect the operation. A single schedule may be executed many times, repeating the same communication pattern. In this way, especially for iterative programs, the cost of computations and negotiations involved in constructing a schedule can often be amortized over many executions. This pattern was pioneered in the CHAOS/PARTI libraries [4]. If a communication pattern is to be executed only once, simple wrapper functions are made available to construct a schedule, execute it, then destroy it. The overhead of creating the schedule is essentially unavoidable, because even in the single-use case individual data movements generally have to be sorted and aggregated, for efficiency. The data structures for this are just those associated with schedule construction.

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

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

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

The # notation was explained in previous section.

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

During construction of a `RemapSkeleton` schedule, all send messages, receive messages, and internal copy operations implied by execution of the schedule are enumerated and stored in light-weight data structures. These messages have to be sorted before sending, for possible message agglomeration, and to ensure a deadlock-free communication schedule. These algorithms, and maintenance of the associated data structures, are dealt with in a base class of `RemapSkeleton` called `BlockMessSchedule`. The API for the superclass is outlined in Fig. 7. To set-up such a low-level schedule, one makes a series of calls to `sendReq` and `recvReq` to define the required messages. Messages are characterized by an offset in some local array segment, and a set of strides and extents parameterizing a multi-dimensional patch of the (flat Java) array. Finally the `build()` operation does any necessary processing of the message lists. The schedule is executed in a "forward" or "backward" direction by invoking `gather()` or `scatter()`.

In general Top-level schedules such as `Remap`, which deal explicitly with distributed arrays, are implemented in terms of some lower-level schedules such as `BlockMessSchedule` that simply operate on blocks and words of data. These lower-level schedules do not directly depend on the `Range` and `Group` classes. The lower level schedules are tabulated in Table 1. Here "words" means contiguous memory blocks of constant (for a given schedule instance) size. "Blocks" means multidi-

```
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() { ... }

  ...
}
```

Fig. 7 API of the class `BlockMessSchedule`

Table 1 Low-level Adlib schedules

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

mensional (r-dimensional) local array sections, parameterized by a vector of r extents and a vector of memory strides. The point-to-point schedules are used to implement collective operations that are deterministic in the sense that both sender and receiver have advanced knowledge of all required communications. Hence `Remap` and other regular communications such as `Shift` are implemented on top of `BlockMessSchedule`. The "remote access" schedules are used to implement operations where one side must inform the other end that a communication is needed. These negotiations occur at schedule-construction time. Irregular communication operations such as collective `Gather` and `Scatter` are implemented on these schedules. The tree schedules are used for various sorts of broadcast, multicast, synchronization, and reduction.

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

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

The range hierarchy of HPJava was illustrated in Fig. 2, and some of the relevant virtual functions are displayed in the API of Fig. 8. Most methods optionally take arguments that allow one to specify a contiguous or strided subrange of interest. The `Triplet` and `Block` instances represent simple struct-like objects holding a few `int` fields. Those integer files are describing respectively a "triplet" interval, and the strided interval of "global" and "local" subscripts that the distribution format maps to a particular process. In the examples here `Triplet` is used only to describe a range of *process coordinates* that a range or subrange is distributed over.

Now the `RemapSkeleton` communication schedule is built by two methods called *sendLoop* and *recvLoop* that enumerate messages to be sent and received respectively. Fig. 9 sketches the implementation of `sendLoop`. This is a recursive function—it implements a multidimensional loop over the `rank` dimensions of the arrays. It is initially called with `r = 0`. An important thing to note is how this function uses the virtual methods on the range objects of the source and destination arrays to enumerate blocks—local and remote—of relevant subranges, and enumerates the messages that must be sent. Figure 10 illustrates the significance of some

« SUPE 11227 layout: Small Extended reference style: basic file: supe125.tex (GIT) aid: 125 doctopic: OriginalPaper class: spr-small-v1 v.2007/02/23 Prn:26/02/2007; 8:00 p. 11»


```
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) {...}
    . . .
}
```

Fig. 8 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.sub_bas;
    int step       = src.str(r) * loc.sub_stp;

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

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

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

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

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

Fig. 9 `sendLoop` method for `Remap`

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. The variables `src` and `dst` represent the distributed array arguments.

Journal ID: 11227, Article ID: 125, Date: 2007-02-26, Proof No: 1

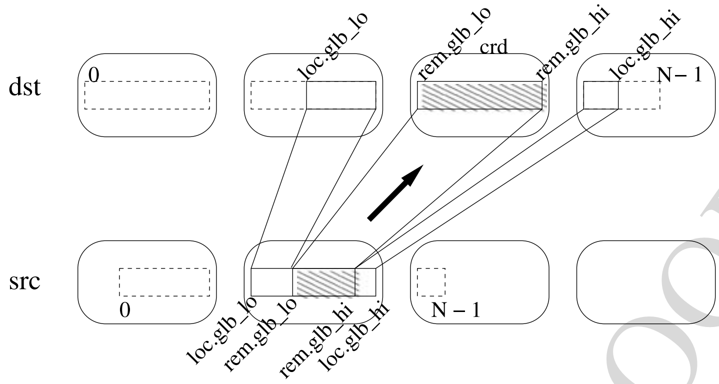


Fig. 10 Illustration of `sendLoop` operation for `remap`

The inquiries `rng()` and `grp()` extract the range and group objects of these arrays.

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

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

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

6 Collective communications

In the previous section we described the Adlib communication implementation issues with a characteristic collective operation example, `remap()`. In this section we will overview functionalities of all collective operations in Adlib. The Adlib has three main families of collective operation: regular communications, reduction oper-

Journal ID: 11227, Article ID: 125, Date: 2007-02-26, Proof No: 1

ations, and irregular communications. We discuss usage and high-level API overview of Adlib methods.

6.1 Regular collective communications

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

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

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

and

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

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

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

6.2 Reductions

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

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

```
t maxval (t # a)
```

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

The boolean reductions:

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

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

6.3 Irregular collective communications

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

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

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

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

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

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

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

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

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

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

7 Application of HPJava

The multigrid method [3] is a fast algorithm for solution of linear and nonlinear problems. It uses a hierarchy or stack of grids of different granularity (typically with a geometric progression of grid-spacings, increasing by a factor of two up from finest to coarsest grid). Applied to a basic relaxation method, for example, multigrid hugely

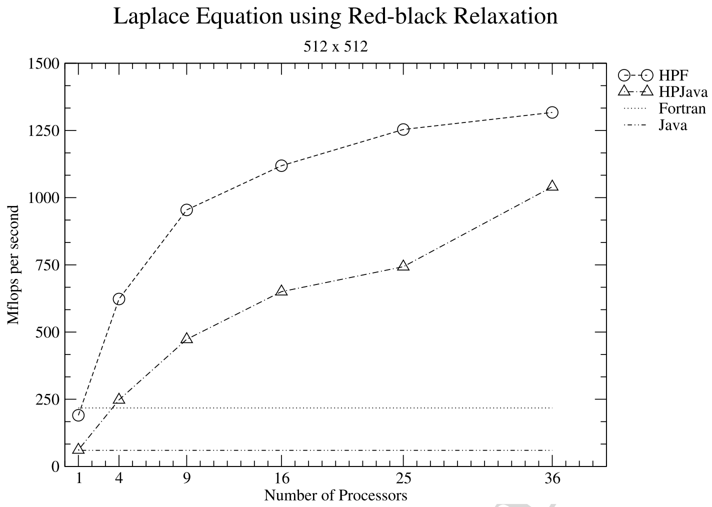


Fig. 11 Laplace equation with size of 512^2

accelerates elimination of the residual by restricting a smoothed version of the error term to a coarser grid, computing a correction term on the coarse grid, then interpolating this term back to the original fine grid. Because computation of the correction term on the fine grid can itself be handled as a relaxation problem, the strategy can be applied recursively all the way up the stack of grids.

The experiments were performed on the SP3 installation at Florida State University. The system environment for SP3 runs were as follows:

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

For best performance, all sequential and parallel Fortran and Java codes were compiled using -O5 or -O3 with -qhot or -O (i.e. maximum optimization) flag.

First we present some results for the computational kernel of the multigrid code, namely unaccelerated red-black relaxation algorithm. Figure 11 gives our results for this kernel on a 512 by 512 matrix. The results are encouraging. The HPJava version scales well, and eventually comes quite close to the HPF code (absolute megaflop performances are modest, but this feature was observed for all our codes, and seems to be a property of the hardware).

The flat lines at the bottom of the graph give the sequential Java and Fortran performances, for orientation. We did not use any auto parallelization feature here. Corresponding results for the complete multigrid code are given in Fig. 12. The results

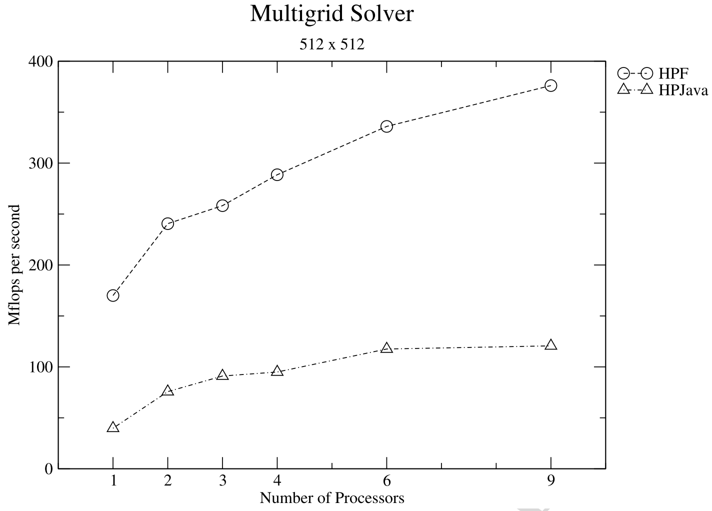


Fig. 12 Multigrid solver with size of 512^2

here are not as good as for simple red-black relaxation-both HPJava speed relative to HPF, and the parallel speedup of HPF and HPJava are less satisfactory.

The poor performance of HPJava relative to Fortran in this case can be attributed largely to the naive nature of the translation scheme used by the current HPJava system. The overheads are especially significant when there are many very tight overall constructs (with short bodies). Experiments done elsewhere [9] leads us to believe these overheads can be reduced by straightforward optimization strategies which, however, are not yet incorporated in our source-to-source translator.

The modest parallel speedup of both HPJava and HPF is due to communication overheads. The fact that HPJava and HPF have similar scaling behavior, while absolute performance of HPJava is lower, suggests the communication library of HPJava is slower than the communications of the native SP3 HPF (otherwise the performance gap would close for larger numbers of processors). This is not too surprising because Adlib is built on top of a portability layer called *mpjdev*, which is in turn layered on MPI. We assume the SP3 HPF is more carefully optimized for the hardware. Of course the lower layers of Adlib could be ported to exploit low-level features of the hardware.

8 HPJava with GUI

In this section we will illustrate how our HPJava can be used with a Java graphical user interface. The Java multithreaded implementation of mpjdev makes it possible for HPJava to cooperate with Java AWT. We ported the mpjdev layer to communicate between the threads of a *single* Java Virtual Machine. The threads cooperate in solving a problem by communicating through our communication library, Adlib, with pure Java version of the mpjdev. By adding pure Java version of the mpjdev to the Adlib communication library, it gives us the possibility to use the Java AWT and other

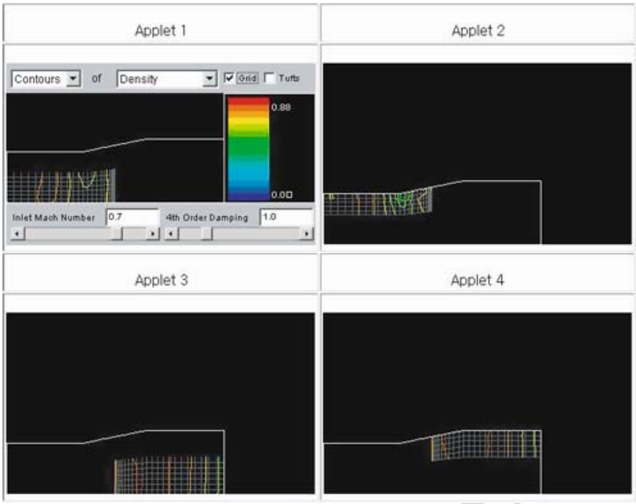


Fig. 13 A 2 dimensional inviscid flow simulation

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 (Fig. 13). The simulation yields contour plots of all flow variables, including velocity components, pressure, Mach number, density and entropy, and temperature. The plots show the location of any shock wave that would reside in the nozzle. Also, the code finds the steady state solution to the 2 dimensional Euler equations, seen below.

$$\frac{\partial U}{\partial t} + \frac{\partial E}{\partial x} + \frac{\partial F}{\partial y} = \alpha H \tag{1}$$

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

The source vector H is zero for this case.

The demo consists of 4 independent Java applets communicating through the Adlib communication library which is layered on top of mpjdev. Applet 1 is handling all events and broadcasting control variables to other applets. Each applet has the responsibility to draw its own portion of the data set into the screen, as we can see in the figure. That this demo also illustrates usage of Java object in our communication library. We are using `writeHalo()` method to communicate Java class object between threads.

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

You can view this demonstration and source code at http://www.hpjava.org/demo.html

9 Conclusions and future work

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

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

References

1. ACM (1998) Workshop on Java for high-performance network computing. Concurr Pract Exp 10(11-13):821–824
2. Agrawal A, Sussman A, Saltz J (1995) An integrated runtime and compiletime approach for parallelizing structured and block structured applications. IEEE Trans Parallel Distrib Syst 6
3. Briggs WL, Van Henson E, McCormick SF (2000) A multigrid tutorial. The Society for Industrial and Applied Mathematics (SIAM)
4. Das R, Uysal MH, Salz JH, Hwang YS (1994) Communication optimizations for irregular scientific computations on distributed memory architectures. J Parallel Distrib Comput 22(3):462–479
5. DeSouza J, Kale LV (2003) Jade: A parallel message-driven java. In: Proceedings of the 2003 workshop on Java in computational science, Melburne, Australia. Available from http://charm.cs.uiuc.edu/papers/ParJavaWJCS03.shtml
6. Java for computational science and engineering—simulation and modelling (1997) Concurr Pract Exp 9(6)
7. Java for computational science and engineering—simulation and modelling II (1997) Concurr Pract Exp 9(11):1001–1002
8. Java Grande Forum home page. http://www.javagrande.org
9. Lee HK (2003) Towards efficient compilation of the HPJava language for high performance computing. PhD thesis, Florida State University
10. Moreira JE, Midkiff SP, Gupta M, Lawrence R (1999) High performance computing with the array package for Java: a case study using data mining. In: Supercomputing 99, November 1999
11. Moreira JE, Midkiff SP, Gupta M (2001) A comparision of three approaches to language, compiler and library support for multidimensional arrays in Java. In: ACM 2001 Java grande/ISCOPE conference. ACM Press
12. Timber Compiler Home Page. http://pds.twi.tudelft.nl/timber
13. Yelick K, Semenzato L, Pike G, Miyamoto C, Liblit B, Krishnamurthy A, Hilfinger P, Graham S, Gay D, Colella P, Aiken A (1998) Titanium: a high-performance Java dialect. In: ACM workshop on Java for high-performance network computing. Concurr Pract Exp (to appear)