# Benchmarking HPJava: Prospects for Performance

Han-Ku Lee, Bryan Carpenter, Geoffrey Fox, Sang Boem Lim
{*hkl, dbc, gcf, slim*}*@grids.ucs.indiana.edu*

*Pervasive Technology Labs*
Indiana University
*Bloomington, Indiana 47401-3730*

*Computational Science and Information Technology,*
Florida State University,
*Tallahassee, Florida 32306-4120*

## Abstract

*We briefly review the authors'* HPJava *programming environment, and compare and contrast with systems like HPF. Because the underlying programming language is Java, and because the HPJava programming model relies centrally on object-oriented run-time descriptors for distributed arrays, the achievable performance has been somewhat uncertain. Individual node performance is the most critical issue. We argue with simple benchmarks that we can in fact hope to achieve performance in a similar ballpark to more traditional HPC languages*[1].

## 1 Introduction

We have described the HPJava language for High Performance Computing in earlier publications such as [3, 2]. HPJava is an implementation of what we call the *HPspmd* programming language model. To facilitate programming of massively parallel, distributed memory computers, it extends the Java language with some additional syntax and some pre-defined classes for handling distributed arrays. In the earlier publications we argued that HPJava should ultimately provide acceptable performance to make it a practical tool for HPC. Here we start to discuss HPJava performance quantitatively. Our system has not quite reached the point where we can provide accurate benchmarks of fully functional HPJava applications running on parallel computers, but we address some of the performance questions which seem most urgent in the particular case of HPJava.

Our HPJava language incorporates some ideas from High Performance Fortran (HPF) [5], and also from lower level library-based SPMD approaches, including MPI. The principles of HPF are *not* adopted wholesale in HPJava. Instead we import a just subset of basic HPF language primitives into HPJava: in particular *distributed arrays*. Although distributed arrays appear as language primitives—the better to support development of libraries operating on distributed data—the actual programming model is not the single-threaded model of HPF. Instead it is direct SPMD programming. Library-based SPMD programming has been very successful in its domain of applicability, and is well-established. We can assume fairly confidently that the lessons and successes of SPMD programming will carry over HPJava. What it is less obvious is that HPJava can provide acceptable performance at each computing node.

There are two reasons why HPJava node performance is uncertain. The first one is that the base language is Java. We believe that Java is a good choice for implementing our HPspmd model. But, due to finite development resources, we can only reasonably hope to use the available commercial Java Virtual Machines (JVMs) to run HPJava node code. HPJava is, for the moment, a source-to-source translator. Thus, HPJava node performance depends heavily upon the third party JVMs. Are they good enough?

The second reason is related to nature of the HPspmd model itself. The data-distribution directives of HPF are most effective if the distribution format of arrays ("block" distribution format, "cyclic" distribution and so on) is known at compile time. This extra static information contributes to the generation of efficient node code[2].

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

[2]It is true that HPF has *transcriptive mappings* which allow code to be developed when the distribution format is not known at compile time, but arguably these are an add-on to the basic

HPJava starts from a slightly different point of view. It is primarily intended as a framework for development of—and use of—libraries that operate on distributed data. Even though distributed arrays may be clearly distinguished from sequential arrays by type signatures, their *distribution format* is typically *not* known in advance (at compile time). Instead distribution format is described by several objects associated with the array—collectively the *Distributed Array Descriptor*. This makes the implementation of libraries simple and natural. But, can we still expect the kind of node performance possible when one has detailed compile-time information about array distribution?

We begin to address these questions in this paper with benchmarks for some simple but relevant algorithms. These algorithms will be benchmarked, comparing the output of current HPJava compiler—which is quite general but not yet particularly efficient—and also with the same code modified by applying some straightforward optimizations that we intend to add to the translator in the near future. We will compare with C++, Fortran and ordinary Java versions of the same algorithms. For now the HPJava versions are executed on a single processor. But—because of the way the translation scheme works—we are reasonably confident that the similar results will carry over to node code on multiple processors.

## 2 Case Study 1: Matrix Multiplication

### 2.1 HPJava Language Featues

Figure 1 is a basic HPJava program for a matrix multiplication. It includes much of the HPJava special syntax, so we will take the opportunity to briefly review the featues of the HPJava language. The program starts by creating an instance `p` of the class `Procs2`. This is a subclass of the special base class `Group`, and describes 2-dimensional grids of processes. When the instance of `Procs2` is created, `P` × `P` processes are selected from the set of processes in which the SPMD program is executing, and labelled as a grid.

The `Group` class, representing an arbitrary HPJava process group, and closely analogous to an MPI group, has a special status in the HPJava language. For example the group object `p` can parametrize an `on(p)` construct. The `on` construct limits control to processes in its parameter group. The code in the `on` construct is *only* executed by processes that belong to `p`. The `on` construct fixes `p` as the *active process group* within its body.

language model rather than a central feature.

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

  double [[-,-]] c = new double [[x, y]] on p ;

  double [[-,*]] a = new double [[x, N]] on p ;
  double [[*,-]] b = new double [[N, y]] on p ;

  ... initialize 'a', 'b'

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

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

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

**Figure 1. Matrix Multiplication in HPJava.**

The `Range` class describes a distributed index range. There are subclasses describing index ranges with different properties. In this example, we use the `BlockRange` class, describing block-distributed indexes. The first argument of the constructor is the global size of the range; the second argument is a *process dimension*—the dimension over which the range is distributed. Thus, ranges `x` and `y` are distributed over the first dimension (i.e. `p.dim(0)`) and second dimension (i.e. `p.dim(1)`) of `p`, and both have `N` elements.

The most important feature HPJava adds to Java is the *distributed array*. 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. Unlike an ordinary array, the index space and associated elements are scattered across the processes that share the array. There are some similarities and differences between HPJava distributed arrays and the ordinary Java arrays. Aside from the way that elements of a distributed array are distributed, the distributed array of HPJava is a *true multi-dimensional array* like that of Fortran. Like in Fortran, one can form a *regular section* of an array. These features of Fortran arrays have adapted and evolved to support scientific and parallel algorithms.

With a process group and a suitable set of ranges, we can declare distributed arrays. The type signature of a distributed array is clearly told by double brackets. In the type signature of a distributed array, each slot

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

  Group p = c.grp() ;

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

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

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

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

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

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

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

**Figure 2. General matrix multiplication.**

holding a hypen, `-`, stands for a distributed dimension, and a astrisk, `*`, a sequential dimension. The array `c` is distributed in both its dimensions. Besides, Arrays `a` and `b` are also distributed arrays, but now each of them has one distributed dimension and one *sequential dimension*.

The `overall` construct is another control construct of HPJava. It represents a distributed parallel loop, sharing some characteristics of the *forall* construct of HPF. The symbol `i` scoped by the `overall` construct is called a *distributed index*. Its value is a *location*, rather an abstract element of a distributed range than an integer value. The indexes iterate over all locations. It is important to note that (with a few special exceptions) the subscript of a distributed array must be a distributed index, and the location should be an element of the range associated with the array dimension. This unusual restriction is an important feature of the model, ensuring that referenced array elements are locally held.

Figure 1 doesn't have any run-time communications because of the special choice of alignment relation between arrays. All arguments for the inner most scalar product are already in place for the computation. We can make a completely general matrix multiplication method by taking arguments with arbitrary distribution, and remapping the input arrays to have the correct alignment relation with the output array. Figure 2 shows the method. The method has two temporary arrays `ta`, `tb` with the desired distribution format. This is determined from `c` by using DAD inquiry functions `grp()` and `rng()` to fetch the distribution group and index ranges of a distributed array. `Adlib.remap()` does the actual communication to remap.

This implementation has some performance issues associated with its memory usage. These issues can be patched up—see [3] for more details. Meanwhile the simple version given here encapsulates some interesting principles of library construction with HPJava—in particular how arrays can be created and manipulated, even though the distribution formats are only determined at run-time.

## 2.2 Translation

In stark distinction to HPF, the HPJava translation scheme *does not* require insertion of compiler-generated communications, making it relatively straightforward. The most complicated part is ensuring that node code works independently of the distribution format of arrays. The current translation schemes is documented in detail in the HPJava manual [3] and translation scheme [1].

Figure 3 gives the translation of the "loop nest" from Figure 1—the two overall constructs with included for loop. Apart from cosmetic whitespace, and renaming of compiler-generated temporary names to the more readable form $t_i$, this is the actual code emitted by the current version of the translator.

## 2.3 Optimization Strategies

For common parallel algorithms, where HPJava is likely to be successful, distributed element access is generally located inside distributed `overall` loops. One main issue optimization strategies must address is the complexity of the associated terms in the subscript expressions for addressing local element (see Figure 3). Optimization strategies should remove overheads of the naive translation scheme (especially for `overall` construct), and speed up HPJava, i.e. produce a Java-based environment competitive in performance with existing Fortran programming environments.

To eliminate complicated distributed index subscript expressions in the inner loops, the final translator will certainly make extensive use strength-reduction optimizations—introducing induction variables that can be computed efficiently by incrementing at suitable

```
int t1 = x.str() ;

Block t2 = x.localBlock() ;
Group t3 = p.restrict(x.dim(),p) ;

for(int t4=0; t4<t2.count; t4++) {
  int t5 = t2.sub_bas + t2.sub_stp * t4 ;
  int t6 = t2.glb_bas + t2.glb_stp * t4 ;

  int t7 = y.str() ;

  Block t8 = y.localBlock() ;
  Group t9 = t3.restrict(y.dim(),t3) ;

  for(int t10=0; t10<t8.count; t10++) {
    int t11 = t8.sub_bas + t8.sub_stp * t10 ;
    int t12 = t8.glb_bas + t8.glb_stp * t10 ;

    double sum = 0 ;

    for(int k=0; k<N; k++){
      sum +=
        a__$DS[a__$bas.base + a__$0.stride * t5 +
               a__$1.off_bas + a__$1.off_stp * k] *
        b__$SD[b__$bas.base + b__$0.off_bas +
               b__$0.off_stp * k +
               b__$1.stride * t11] ;
    }

    c__$DD[c__$bas.base + c__$0.stride * t5 +
           c__$1.stride * t11] = sum ;
  }
}
```

**Figure 3. Naive translation of nested overall loops from Figure 1.**

points with the induction increments. Another simple but potentially profitable strategy is loop-unrolling. Furthermore we are likely to need code movement optimizations to reduce the need for method invocations on the special run-time support classes, like `Block` and `Group`. In the original `overall` translation scheme, we use the `localBlock()` method to compute parameters of the local loop. The translation is identical for every distribution format—block-distribution, simple-cyclic distribution, aligned subranges, and so on—supported by the language. Of course there is an overhead related to abstracting this local-block parameter computation into a method call, but at least the method call is made at most once at the start of each loop. Still we need to minimize this overhead where possible, by code movement strategies like partial redundancy elimination, which will need to be adapted for our source-to-source translator. There is considerable scope for common-subexpression elimination in typical `overall` loops. Naive translation for inner loops tends to repeatly declare some variables for holding `str()` methods, global bases and steps, and so on.

| HPJava | Naive | Strength Reduction | Loop unrolling |
|---|---|---|---|
| HPJava | 125.0 | 166.7 | 333.4 |
| Java | 251.3 | 403.2 | 388.3 |
| C++ | 533.3 | 436.7 | 552.5 |
| F77 | 536.2 | 531.9 | 327.9 |

**Table 1. Matrix multiplication performance. HPJava code for single processor. Other codes sequential. All speeds in MFLOPS.**

## 2.4 Benchmarks

For purposes of this paper—and to understand what directions are most profitable for future development—the simplest and most practical kinds of optimization strategies have been applied "manually" (but we believe honestly—i.e. it seems straightforward to apply these strategies automatically, with a modest amount of extra work on the translator) to the HPJava matrix multiplication programs. The matrix sizes are 100 by 100. The results use the IBM Developer Kit 1.3 (JIT) with `-O` flag on Pentium4 1.5GHz Red Hat 7.2 Linux machines. We also compared the sequential Java, C++, and Fortran version of the HPJava program, all with `-O` or `-O5` (i.e. maximun optimization) flag when compiling.

Table 1 shows that the strength reduction optimization helps in the Java case (for C++ and Fortran, the compilers can be expected to implement similar optimizations, and we don't see any improvement from the source-level "optimizations"). The main message is that, with a little help in the form of source-to-source transformations, Java can get 70-75% of the performance of C++ or Fortran. After similar optimizations, HPJava with more complex subscripting expressions has similar but marginally slower performance. Of course we expect that the HPJava results will scale on suitable parallel platforms, so a *modest* penalty in node performance is considered acceptable.

```
Procs2 p = new Procs2(2, 3) ;
on(p) {
  Range x = new ExtBlockRange(N, p.dim(0)) ;
  Range y = new ExtBlockRange(N, p.dim(1)) ;

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

  ... initialization for 'a'

  for(int iter=0; iter<count; iter++){

    Adlib.writeHalo(a, wlo, whi);

    overall(i=x for 1 : N - 2)
      overall(j=y for 1+(i'+iter)%2 : N-2 : 2) {
        a[i,j] = 0.25F * (a [i-1,j] + a [i+1,j] +
                          a [i,j-1] + a [i,j+1]);
      }
  }
}
```

**Figure 4. Red-black iteration.**

## 3 Case Study 2: Red-Black Relaxation

Red-black relaxation is an old favorite. It is still interesting since it is a kernel in some practical solvers (for example we have an HPJava version of a multigrid solver in which relaxation it is a dominantly time-consuming part). Also it conveniently exemplifies a whole family of similar, local, grid-based algorithms and simulations.

We can see an HPJava version of red-black relaxation of the two dimensional Laplace equation in Figure 4. Here we use a different class of distributed range. The class `ExtBlockRange` adds *ghost-regions* [4] to distributed arrays that use them. A library function called `Adlib.writeHalo` updates the cached values in the ghost regions with proper element values from neighboring processes.

There are a few additional pieces of syntax here. The range of iteration of the overall construct can be restricted by adding a general triplet after the `for` keyword. The i' is read "i-primed", and yields the integer *global index* value for the distributed loop (`i` itself does not have a numeric value—it is a symbolic subscript). Finally, if the array ranges have ghost regions, the general policy that an array subscript must be a simple distributed index is relaxed slightly—a subscript can be a *shifted index*, as here. The value of the numeric shift—symbolically added to or subtracted from the index—must not exceed the width of the ghost regions, and the index that is shifted must be a location in the distributed range of the array, as before.

Figure 5 gives the translation of the "loop nest" from

```
for(int iter=0; iter<count; iter++){

  Adlib.writeHalo(a__$DD,a__$bas,a__$0,a__$1,p) ;

  int t1= x.str();

  Block t2=x.localBlock(1,N-2);
  Group t3=p.restrict(x.dim(),p);

  for(int t4=0;t4<t2.count;t4++){
    int t5=t2.sub_bas+t2.sub_stp*t4;
    int t6=t2.glb_bas+t2.glb_stp*t4;

    int t7= y.str();

    Block t8=y.localBlock(1+(t6+iter)%2,N-2,2);
    Group t9=t3.restrict(y.dim(),t3);

    for(int t10=0;t10<t8.count;t10++){
      int t11=t8.sub_bas+t8.sub_stp*t10;
      int t12=t8.glb_bas+t8.glb_stp*t10;

      a__$DD[a__$bas.base+a__$0.stride*t5+
                          a__$1.stride*t11]=
         0.25F*(a__$DD[a__$bas.base+
                       a__$0.stride*(t5-t1*1)+
                       a__$1.stride*t11]+
                a__$DD[a__$bas.base+
                       a__$0.stride*(t5+t1*1)+
                       a__$1.stride*t11]+
                a__$DD[a__$bas.base+
                       a__$0.stride*t5+
                       a__$1.stride*(t11-t7*1)]+
                a__$DD[a__$bas.base+
                       a__$0.stride*t5+
                       a__$1.stride*(t11+t7*1)]);
    }
  }
}
```

**Figure 5. Naive translation of main loop from Figure 4.**

| HPJava | Naive | Strength Reduction | Loop unrolling |
|---|---|---|---|
| HPJava | 113.5 | 168.1 | 212.0 |
| Java | 238.5 | 239.6 | 242.0 |
| C++ | 248.0 | 249.3 | 248.0 |
| F77 | 246.8 | 246.8 | 171.6 |

**Table 2. Red-black relaxation performance. HPJava code for single processor. Other codes sequential. All speeds in MFLOPS.**

Figure 4. Here the subscripting expressions are particularly complex, the strength-reduction is likely to be very effective. There is a relatively expensive method call:

```
y.localBlock(1 + (t6 + iter) % 2, N-2, 2)
```

used in the translation of the overall construct:

```
overall(j = y for 1 + (i‘ + iter) % 2 : N - 2 : 2)
  ...
```

is repeated in every iteration of the loop associated with the outer overall. Because red-black iteration is a common pattern, the translator should probably try to recognize this idiom. If a nested overall includes expressions of the form

```
(i‘ + expr) % 2
```

where `i‘` is the global index of the outer overall, and `expr` is invariant with respect to the outer loop, this should be taken as an indication to unroll the outer loop by 2. The arguments of the call to `localBlock()` then become loop invariant, and the whole invocation is a candidate to be lifted out of the outer loop (of course this depends on some special knowledge of the method `localBlock()`). In Table 2 this is what we mean by "loop unrolling".

The table shows that in this more complex example the relative performance of the optimized HPJava node code is actually somewhat better than in the previous test case—85% of straightforward C++ or Fortran. Perhaps the reference codes could be improved, but note that whenever we quote C++ or Fortran performance we are using the highest optimization level supported by the compiler (-O5). Because we are only interested in performance of the generated node code, HPJava timings omit call to `Adlib.writeHalo()`.

## 4 Conclusion

HPJava is an instance of what we call the *HPspmd model*: arguably it is not exactly a high-level parallel programming language in the ordinary sense, but rather a tool to assist parallel programmers in writing SPMD code. In this respect the closest recent language we are familiar with is probably Co-Array Fortran [6], but HPJava and Co-Array Fortran have many obvious differences. In Co-Array Fortran, array subscripting is local by default, or involves a combination of local subscripts and explicit process ids. There is no analogue of global subscripts, or HPF-like distribution formats. In Co-Array Fortran the logical model of communication is built into the language—remote memory access with intrinsics for synchronization. In HPJava, there are no communication primitives in the language itself. We follow the MPI philosophy of providing communication through separate libraries.

The main purpose of this paper was to verify if our library-based HPspmd language extensions can be implemented efficiently in the context of Java. The underlying communication libraries and parallelization strategies have been proven in other domains of application (e.g. [7]). Thus, the emphasis is on demonstrating that a simple translation scheme for HPJava can produce efficient *node code*. Although we are still a little way from realizing this efficient scheme in our actual translator, the results here are encouraging.

## References


[1] B. Carpenter, G. Fox, H.-K. Lee, and S. B. Lim. Translation Schemes for the HPJava Parallel Programming Language. In *11th International Workshop on Languages and Compilers for Parallel Computing 2001*, 2001.

[2] B. Carpenter, G. Zhang, G. Fox, X. Li, X. Li, and Y. Wen. Towards a Java environment for SPMD programming. In D. Pritchard and J. Reeve, editors, *4th International Europar Conference*, volume 1470 of *Lecture Notes in Computer Science*. Springer, 1998. http://aspen.csit.fsu.edu/pss/HPJava.

[3] B. Carpenter, G. Zhang, H.-K. Lee, and S. Lim. *Parallel Programming in HPJava*. Draft, 2001. http://aspen.csit.fsu.edu/pss/HPJava.

[4] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, 2(3):171–193, 1990.

[5] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, special issue, 2, 1993.

[6] R.W.Numrich and J.L.Steidel. F- -: a simple parallel extension to Fortran 90. *SIAM News*, page 30, 1997.

[7] G. Zhang, B. Carpenter, G. Fox, X. Li, X. Li, and Y. Wen. PCRC-based HPF compilation. In Z. L. et al, editor, *10th International Workshop on Languages and Compilers for Parallel Computing*, volume 1366 of *Lecture Notes in Computer Science*. Springer, 1997.