Translation Schemes for the *HPJava* Parallel Programming Language

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

School of Computational Science and Information Technology,
400 Dirac Science Library,
Florida State University,
Tallahassee, Florida 32306-4120
{*dbc,fox,hkl,slim*}*@csit.fsu.edu*

July 28, 2001

Abstract

The article describes the current status of the authors' HPJava programming environment. HPJava is a parallel dialect of Java that imports Fortran-like arrays—in particular the distributed arrays of High Performance Fortran—as new data structures. The article discusses the translation scheme adopted in a recently developed translator for the HPJava language. It also gives an overview of the language.

1 Introduction

HPJava [**?**] is a language for parallel programming, especially suitable for programming massively parallel, distributed memory computers.

Several of the ideas in HPJava are lifted from the High Performance Fortran (HPF) programming language. However the programming model of HPJava is "lower level" than the programming model of HPF. HPJava sits somewhere between the explicit SPMD (Single Program Multiple Data) programming style—often implemented using communication libraries like MPI—and the higher level, data-parallel model of HPF. An HPF compiler generally guarantees an equivalence between the parallel code it generates and a sequential Fortran program obtained by deleting all distribution directives. An HPJava program, on the other hand, is defined from the start to be a distributed MIMD program, with multiple threads of control operating in different address spaces. In this sense the HPJava programming model is "closer to the metal" than the HPF programming model. HPJava *does* provide special syntax for HPF-like distributed arrays, but the programming model may best be viewed as an incremental improvement on the programming style used in many hand-coded

applications: explicitly parallel programs exploiting collective communication libraries or collective arithmetic libraries.

We call this general programming model—essentially direct SPMD programming supported by additional syntax for HPF-like distributed arrays—the *HPspmd model*. In general SPMD programming has been very successful. Many high-level parallel programming environments and libraries assume the SPMD style as their basic model. Examples include ScaLAPACK [**?**], DAGH [**?**], Kelp [**?**] and the Global Array Toolkit [**?**]. While there remains a prejudice that HPF is best suited for problems with rather regular data structures and regular data access patterns, SPMD frameworks like DAGH and Kelp have been designed to deal directly with irregularly distributed data, and other libraries like CHAOS/PARTI [**?**] and Global Arrays support unstructured access to distributed arrays. Presently, however, the library-based SPMD approach to data-parallel programming lacks the uniformity and elegance that was promised by HPF. The various environments referred to above all have some idea of a distributed array, but they all describe those arrays differently. Because the arrays are managed entirely in libraries, the compiler offers little support and no safety net of compile-time or compiler-generated run-time checking. The HPspmd model is one attempt to address such shortcomings.

HPJava is a particular instantiation of this HPspmd idea. As the name suggests, the *base language* in this case is the Java™ programming language. To some extent the choice of base language is incidental, and clearly we could have added equivalent extensions to another language, such as Fortran itself. But Java does seem to be a better language in various respects, and it seems plausible that in the future more software will be available for modern object-oriented languages like Java than for Fortran.

HPJava is a strict extension of Java. It incorporates all of Java as a subset. Any existing Java class library can be invoked from an HPJava program without recompilation. As explained above, HPJava adds to Java a concept of multi-dimensional, distributed arrays, closely modelled on the arrays of HPF[1]. Regular sections of distributed arrays are fully supported. The multidimensional arrays can have any rank, and the elements of distributed arrays can have any standard Java type, including Java class types and ordinary Java array types.

A translated and compiled HPJava program is a standard Java class file, which will be executed by a distributed collection of Java Virtual Machines. All externally visible attributes of an HPJava class—e.g. existence of distributed-array-valued fields or method arguments—can be automatically reconstructed from Java signatures stored in the class file. This makes it possible to build libraries operating on distributed arrays, while maintaining the usual portability and compatibility features of Java. The libraries themselves can be implemented in HPJava, or in standard Java, or through Java Native Interface (JNI) wrappers to code implemented in other languages. The HPJava language specification carefully documents the mapping between distributed arrays and the standard-

[1] "Sequential" multi-dimensional arrays—essentially equivalent to Fortran 95 arrays—are available as a subset of the HPJava distributed arrays.

Java components they translate to.

While HPJava does not incorporate HPF-like "sequential" semantics for manipulating its distributed arrays, it does add a small number of high-level features designed to support direct programming with distributed arrays, including a distributed looping construct called `overall`. To directly support lower-level SPMD programming, it also provides a complete set of inquiry functions that allow the local array segments in distributed arrays to be manipulated directly, where necessary.

In the current system, syntax extensions are handled by a preprocessor that emits an ordinary SPMD program in the base language. The HPspmd syntax provides a relatively thin veneer on low-level SPMD programming, and the transformations applied by the translator are correspondingly direct—little non-trivial analysis should be needed to obtain good parallel performance. What the language does provide is a uniform model of a distributed array. This model can be targetted by reusable libraries for parallel communication and arithmetic. The specific model adopted very closely follows the distributed array model defined in the High Performance Fortran standard.

This article describes ongoing work on refinement of the HPJava language definition, and the development of a translator for this language.

2 HPJava—an HPspmd language

HPJava extends its base language, Java, by adding some predefined classes and some additional syntax for dealing with distributed arrays. We aim to provide a flexible hybrid of the data parallel and low-level SPMD paradigms. To this end HPF-like distributed arrays appear as language primitives. The distribution strategies allowed for these arrays closely follow the strategies supported in HPF—any dimension of an array can independently be given blockwise, cyclic, or other distribution format[2], array dimensions can have strided alignments to dimensions other arrays, arrays as a whole can be replicated over axes of processor arrangements, and so on.

A design decision is made that all access to *non-local* array elements should go through explicit calls to library functions. These library calls must be placed in the source HPJava program by the programmer. This requirement may be surprising to people expecting to program in high-level parallel languages like HPF, but it should not seem particularly unnatural to programmers presently accustomed to writing parallel programs using MPI or other SPMD libraries. The exact nature of the communication library used is not part of the HPJava language design, per se. An appropriate communication library might perform collective operations on whole distributed arrays (as illustrated in the following examples), or it might provide some kind of `get` and `put` functions for access

[2]The current HPJava translator does not implement block-cyclic distribution format, and in general the HPJava language design can't very easily accomodate the `INDIRECT` mappings present in the extended version of HPF. To our knowledge these are the only major omission from the HPF standards.

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

  float [[-,-]] a = new float [[x, y]], b = new float [[x, y]],
                c = new float [[x, y]] ;

  ... initialize values in 'a', 'b'

  overall(i = x for :)
    overall(j = y for :)
      c [i, j] = a [i, j] + b [i, j] ;
}
```

Figure 1: A parallel matrix addition.

to remote blocks of a distributed array, similar to the functions provided in the Global Array Toolkit [**?**], for example.

A subscripting syntax can be used to directly access *local* elements of distributed arrays. A well-defined set of rules—automatically checked by the translator—ensures that references to these elements can only be made on processors that hold copies of the elements concerned. Alternatively one can access local elements of a distributed array indirectly, by first extracting the locally held block of elements, then subscripting this block as a local sequential array.

To facilitate the general scheme, the language adds three *distributed control* constructs to the base language. These play a role something like the `ON HOME` directives of HPF 2.0 and earlier data parallel languages [**?**]. One of the special control constructs—a distributed parallel loop—facilitates traversal of locally held elements of distributed arrays.

Mapping of distributed arrays in HPJava is described in terms of a two special classes: `Group` and `Range`. *Process group* objects generalize the processor arrangements of HPF, and *distributed range* objects are used in place HPF templates. A distributed range is comparable with a single dimension of an HPF template. The changes relative to HPF (with its processor arrangements and multi-dimensional templates) are best be regarded as a modest change of *parametrization* only: the set of mappings that can be represented is unchanged.

Figure 1 is a simple example of an HPJava program. It illustrates creation of distributed arrays, and access to their elements. The class `Procs2` is a standard library class derived from the special base class `Group`, and representing a two-dimensional grid of processes. The distributed range class `BlockRange` is a library class derived from the special class `Range`; it denotes a range of subscripts distributed with BLOCK distribution format. Process dimensions associated with a grid are returned by the `dim()` inquiry. The `on(p)` construct is a new control construct specifying that the enclosed actions are performed only by processes in group `p`.

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

  float [[-,-]] u = new float [[x, y]] ;

  ... some code to initialise 'u'

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

    Adlib.writeHalo(u) ;

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

Figure 2: Red-black iteration.

The variables `a`, `b` and `c` are all distributed array objects. The type signature of an r-dimensional distributed array involves double brackets surrounding r comma-separated slots. The constructors specify that these all have ranges `x` and `y`—they are all `M` by `N` arrays, block-distributed over `p`.

A second new control construct, `overall`, implements a distributed parallel loop. 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, with a couple of exceptions noted below, the subscripts in element references must be distributed indexes. The locations associated with these indexes must be in the range associated with the array dimension. This restriction is a principal means of ensuring that referenced array elements are held locally.

This general policy is relaxed slightly to simplify coding of stencil updates. A subscript can be a *shifted index*. Usually this is only legal if the subscripted array is declared with suitable *ghost regions* [**?**]. Figure 2 illustrates the use of the standard library class `ExtBlockRange` to create arrays with ghost extensions (in this case, extensions of width 1 on either side of the locally held "physical" segment). A function, `writeHalo`, from the communication library Adlib updates the ghost region. If `i` is a distribute index, the expression `i'` (read "i-primed") yields the integer global loop index.

Distributed arrays can be defined with some sequential dimensions. The sequential attribute of an array dimension is flagged by an asterisk in the type signature. As illustrated in Figure 3, element reference subscripts in sequential

```
Procs1 p = new Procs1(P) ;
on(p) {
  Range x = new BlockRange(N, p.dim(0)) ;

  float [[-,*]] a = new float [[x, N]], c = new float [[x, N]] ;
  float [[*,-]] b = new float [[N, x]], tmp = new float [[N, x]] ;

  ... initialize ‘a’, ‘b’

  for(int s = 0 ; s < N ; s++) {

    overall(i = x for :) {

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

      c [i, (i‘ + s) % N] = sum ;
    }

    // cyclically shift ‘b’ (by amount 1 in x dim)...

    Adlib.cshift(tmp, b, 1, 1) ;
    Adlib.copy(b, tmp) ;
  }
}
```

Figure 3: A pipelined matrix multiplication program.

dimensions can be ordinary integer expressions.

The last major component of the basic HPJava syntax 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 new array contains 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 sections of a target array. As in Fortran, subscripts in section expressions can be index triplets. The language also has built-in ideas of *subranges* and *restricted groups*. These can be used in array constructors on the same footing as the ranges and grids introduced earlier, and they enable 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 and irregular patterns of communication would be dealt with in libraries, which can implement many richer operations than the `writeHalo` and `cshift` functions of the examples.

The examples given so far look very much like HPF data-parallel examples, written in a different syntax. We will give one final example to emphasize the point that the HPspmd model is *not* the HPF model. If we execute the following HPJava program

```
Procs2 p = new Procs2(2, 3) ;
on(p) {
  Dimension d = p.dim(0), e = p.dim(1) ;

  System.out.println("My coordinates are (" + d.crd() +
                                        ", " + e.crd() + ")") ;
}
```

we could see output like:

```
My coordinates are (0, 2)
My coordinates are (1, 2)
My coordinates are (0, 0)
My coordinates are (1, 0)
My coordinates are (1, 1)
My coordinates are (0, 1)
```

There are 6 messages. Because the 6 processes are running concurrently in 6 JVMs, the order in which the messages appear is unpredictable. An HPJava program is a MIMD program, and any appearance of collective behavior in previous examples was the result of a particular programming style and a good library of collective communication primitives. In general an HPJava program can freely exploit the weakly coupled nature of the process cluster, often allowing more efficient algorithms to be coded.

3 Miscellaneous language issues

Early versions of HPJava (see for example, [**?**]) adopted the position that a distributed array should be a kind of Java object. After working with this approach for some time, our position changed. In our current language definition a distributed array type is not an ordinary Java reference type. It is a new kind of reference type that does not extend `Object`. In practise a single distributed array is translated to several Java objects in the emitted code.

An early motivation for this change was to avoid introducing infinitely many different Java classes for the different distributed array types. However the change has other advantages. Now that a distributed array type no longer extends `Object` we are liberated from having to support various object-like behaviors, that would make efficient translation of operations on distributed arrays harder than it needs to be.

The HPJava translator only applies its transformations code in *HPspmd classes*. These are classes that implement a marker interface called `HPspmd`. Classes that do not implement this interface are not transformed and cannot use the special syntax extensions of HPJava.

Many of the special operations in HPJava rely on the knowledge of the currently active process group—the *APG*. This is a context value that will change during the course of the program as distributed control constructs limit control to different subsets of processors. In the current HPJava translator the value of the APG is passed as a hidden argument to methods and constructors of HPspmd classes (so it is handled something like the `this` reference in typical object-oriented languages).

The HPJava language has a set of rules that the translator enforces to help ensure a rational parallel program. These govern where in a program certain constructs can legally appear, and the allowed subscripts in distributed arrays.

The value of the current active process group is used to determine whether particular distributed control contructs and certain collective array operations are legal at particular points in a program. So long as these basic rules are respected, distributed control constructs can be nested freely, and generally speaking collective operations will operate properly within the restricted APGs in effect inside the constructs.

So far as subscripting is concerned, a characteristic rule is that distributed array element reference in:

```
overall(i = x for l : u : s) {
  ... a[e0,...,er-1,i,er+1,...]  ...
}
```

is allowed if and only if

1. The expression a is invariant in the `overall` construct.
2. All locations in x`[`l`:`u`:`s`]` are of elements a`.rng(`r`)`.

The syntax x`[`l`:`u`:`s`]` represents a subrange, and the inquiry a`.rng(`r`)` returns the rth range of a. This rule is a statement about the `overall` construct as a whole, not about the array accesses in isolation. The rule applies to any access that appears textually inside the constructs, even if some conditional test in the body of the construct might prevent those accesses from actually being executed. This is important because it allows any associated run-time checking code to be lifted outside the local loops implied by an `overall`.

4 Basic translation scheme

In some ways the philosophy behind our HPspmd translator is orthogonal to the approach in writing a true compiler. There is a deliberate effort to keep the translation scheme simple and apparent to the programmer. Aggressive optimizations of the local code are left to the compiler (or JVM) used as a backend. The full translation scheme is documented in the HPJava report at [**?**]. This is a work in progress, and the document evolves as the translator matures.

SOURCE:

T `[[`$attr_0$`, ..., `$attr_{R-1}$`]]` a `;`

TRANSLATION:

T `[]` a'_{dat} `;`

`ArrayBase` a'_{bas} `;`

$DIMENSION\_TYPE(attr_0)$ a'_0 `;`
`...`
$DIMENSION\_TYPE(attr_{R-1})$ a'_{R-1} `;`

Figure 4: Translation of a distributed-array-valued variable declaration.

4.1 Translation of distributed arrays

Figure 4 gives a schema for translating a distributed array declaration in the source HPJava program. Here T is some Java type, a'_{dat}, a'_{bas} and $a'_0 \ldots a'_{R-1}$ are new identifiers, typically derived from a by adding some suffixes, the strings $attr_r$ are each either a hyphen, `-`, or an asterisk, `*`, and the "macro" *DIMENSION_TYPE* is defined as

$$DIMENSION\_TYPE(attr_r) \equiv \texttt{ArrayDim}$$

if the term $attr_r$ is a hyphen, or

$$DIMENSION\_TYPE(attr_r) \equiv \texttt{SeqArrayDim}$$

if the term $attr_r$ is an asterisk.

If, for example, a class in the source program has a field:

```
float [[-,-,*]] bar ;
```

the translated class may be assumed to have the five fields:

```
float [] bar__$DDS ;

ArrayBase bar__$bas ;

ArrayDim bar__$0 ;
ArrayDim bar__$1 ;
SeqArrayDim bar__$2 ;
```

In general a rank-r distributed array in the source program is converted to $2 + r$ variables in the translated program. The first variable is an ordinary, one-dimensional, Java array holding local elements. A simple "struct"-like object of

SOURCE:

`overall (`i `=` x `for` e_{lo} `:` e_{hi} `:` e_{stp}`)` S

TRANSLATION:

```
Block b = x.localBlock(T[e_lo], T[e_hi], T[e_stp]) ;

Group p = ((Group) apg.clone()).restrict(x.dim()) ;

for (int l = 0 ; l < b.count ; l++) {
    int sub = b.sub_bas + b.sub_stp * l ;
    int glb = b.glb_bas + b.glb_stp * l ;

    T[S|p]
}
```

where:

i is an index name in the source program,
x is a simple expression in the source program,
e_{lo}, e_{hi}, and e_{stp} are expressions in the source,
S is a statement in the source program, and
b, p, l, *sub* and *glb* are names of new variables.

Figure 5: Translation of `overall` construct.

type `ArrayBase` contains a base offset in this array and an HPJava `Group` object (the distribution group of the array). r further simple objects of type `ArrayDim` each contain an integer stride in the local array and an HPJava `Range` object describing the dimensions of the distributed array. The class `SeqArrayDim` is a subclass of `ArrayDim`, specialized to parameterize sequential dimensions conveniently.

One thing to note is that a class file generated by compiling the translated code will contain the generated field names. These follow a fixed prescription, so that when a pre-compiled class file (from some library package, say) is read by the HPJava translator, it can reconstruct the original distributed array signature of the field from the $2 + r$ fields in the class file. It can then correctly check usage of the external class. By design, the translator can always reconstruct the HPspmd class signatures from the standard Java class file of the translated code.

SOURCE:

$$e \quad \equiv \quad a \ \texttt{[}e_0\texttt{, } \ldots\texttt{, } e_{R-1}\texttt{]}$$

TRANSLATION:

$$\mathbf{T}\,[e] \quad \equiv \quad \mathbf{T}_{\mathrm{dat}}\,[a] \quad \texttt{[}\mathit{OFFSET}(a, e_0, \ldots, e_{R-1})\texttt{]}$$

where:

The expression a is the subscripted array,
each term e_r is either an integer, a distributed index name,
or a shifted index expression, and
the macro *OFFSET* is defined in the text.

Figure 6: Translation of distributed array element access.

4.2 Translation of the `overall` construct

The schema in Figure 5 describes basic translation of the `overall` construct. The `localBlock()` method on the `Range` class returns parameters of the locally held block of index values associated with a range. These parameters are returned in another simple "struct"-like object of class `Block`. Terms like $\mathbf{T}\,[e]$ represent the translated form of expression e.

The *local subscript* for the index i is the value of *sub*. This value is used in subscripting distributed arrays. The *global index* for the index i is the value of *glb*. This value is used in evaluating the global index expression i`‘`.

Because we use the run-time inquiry function `localBlock()` to compute parameters of the local loop, this translation is identical for every distribution format supported by the language (block-distribution, simple-cyclic distribution, aligned subranges, and several others). Of course there is an overhead associated with abstracting this computation into a method call; but the method call is made at most once at the start of each loop, and we expect that in many cases optimizing translators will recognize repeat calls to these methods, or recognize the distribution format and inline the computations, reducing the overhead further.

$\mathbf{T}\,[S\,|p]$ means the translation of S in the context of p as active process group.

4.3 Translating element access in distributed arrays

We only need to consider the case where the array reference is a distributed array: the general scheme is illustrated in Figure 6. The macro *OFFSET* is

defined as

$$
\begin{aligned}
&\mathit{OFFSET}(a, e_0, \ldots, e_{R-1}) \equiv \\
&\qquad \mathbf{T}_{\mathrm{bas}}[a]\texttt{.base} \;\; \texttt{+} \;\; \mathit{OFFSET\_DIM}(\mathbf{T}_0[a], e_0) \\
&\qquad\qquad\qquad\qquad \ldots \\
&\qquad\qquad\qquad\quad \texttt{+} \;\; \mathit{OFFSET\_DIM}(\mathbf{T}_{R-1}[a], e_{R-1})
\end{aligned}
$$

There are three cases for the macro *OFFSET_DIM* depending on whether the subscript argument is a distributed index, a shifted index, or an integer subscripts (in a sequential dimension). We will only illustrate the case where e_r is a distributed index i. Then

$$
\mathit{OFFSET\_DIM}(a'_r, e_r) \equiv \quad a'_r\texttt{.stride * } \mathit{sub}
$$

where *sub* is the local subscript variable for this index (see the last section).

Ultimately—as we should expect for regular access patterns—the local subscript computations reduce to expressions linear in the indices of local loops. Such subscripting patterns are readily amenable to optimization by the compiler back-end, or, more likely, they can be further simplified by the HPspmd translator itself.

We have only sketched three of the more important schema, leaving out details. The full translation scheme for HPJava, recorded in [**?**], involves perhaps a couple of dozen such schema of varying complexity. In practice the translation phase described here is preceded by a "pre-translation" phase that simplifies some complex expressions by introducing temporaries, and adds run-time checking code for some of the rules described in section 3.

5 Status and prospects

The first fully functional version of the HPJava translator is now operational. Over the last few weeks the system as been tested and debugged against a small test suite of available HPJava programs. Currently most of the examples are short, although the suite does include an 800-line Multigrid code, transcribed from an existing Fortran 90 program. One pressing concern over the next few months is to develop a much more substantial body of test code and applications.

As we have emphasized, HPJava includes all of standard Java as a subset. "Translation" of the conventional Java part of the language is very easy. It is a design feature of HPJava that the translation system handles code that *looks like* base language code in *exactly* the same way as it would be handled by a compiler for the base language. In our source-to-source translation strategy, this means that standard Java statements and expressions are copied through essentially unchanged. On the other hand the inclusion of Java means that we do need a front-end that covers the whole of Java. The translation scheme for HPJava depends in an important way on type information. It follows that we need type analysis for the whole language, including the Java part. Writing

a full type-checker for Java is not trivial (especially since the introduction of nested types). In practice development of the front-end, and particularly the type-checker, has been the most time-consuming step in developing the whole system. The HPJava translator is written in Java. The parser was developed using the JavaCC and JTB tools.

It is too early to give detailed benchmarks. However we will give some general arguments that lead us to believe that in the near future we can hope to obtain effective performance using our system. For the sake of definiteness, consider the Multigrid example referred to above. This is a good example for HPJava, because it is an example of an algorithm that is quite complex to code "by hand" as a parallel program, and relatively easy to code using HPJava together with the communication library Adlib. The detailed logic of the Multigrid algorithm has an interesting recursive structure, but the core of the computational work boils down to red-black relaxation (Figure 2). If we can code this simple algorithm well, we expect the whole of the solver should work reasonably well.

The general feasibility of programming this kind of algorithm on a massively parallel, distributed memory computer is presumably not at issue. As discussed in the previous section, the translation scheme for HPJava ultimately reduces the `overall` constructs of the source program to simple `for` loops—much the same as the kind of `for` loops one would write in a hand-coded MPI program[3]. Earlier experiences, using the high-level Adlib communication library as run-time support for research implementations of HPF, lead us to believe that this library should not introduce unacceptable overheads. (Adlib is a C++ library built on top of MPI. The Java binding is through JNI.) So this leaves us with the question of whether Java will be the weak link, in terms of performance. The answer seems to be "perhaps not". Recent benchmarking efforts [**?**] indicate that—at least on major commodity platforms—Java JIT performance is approaching parity with C and even Fortran compilers. We believe that these results will carry over to our applications.

6 Conclusion

HPJava is conceived as a parallel programming language extended from, and fully compatible with, Java—perhaps the most modern programming language in widescale use at the time of writing. It imports language constructs from Fortran 90 and High Performance Fortran that are believed to be important to support effective scientific programming of massively parallel computers.

HPJava is an instance of what we call the *HPspmd model*: 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 F`--` [**?**], but HPJava and F`--` have many obvious differences.

[3]More precisely, they will be much the same once the raw code, generated by the basic translation scheme, has been cleaned up by some straightforward optimizations. This is the step that is missing at the time of writing.

Parallel programming aside, HPJava is also one of several recent efforts to put "scientific", Fortran-like arrays into Java. At meetings of the Java Grande Forum, for example, this has been identified as an important requirement for wider deployment of scientific software in Java. Work at IBM over the last few years [**?**, **?**, **?**, **?**] has been particularly influential in this area, leading to a JSR (Java Specification Request) for standardized scientific array classes. The approach taken in HPJava—using a preprocessor to break arrays into components rather than introducing multidimensional array classes as such—is somewhat different. For us the preprocessor approach was essentially mandated by the proliferation of distinct distributed array types in our model. Generating the large number of array classes that would be needed seems to be impractical. In recent talks the IBM group have also discussed comparable approaches for sequential multiarrays without introducing specific array classes [**?**].

7 Acknowledgements

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