Semantic Checking in *HPJava*

Bryan Carpenter, Geoffrey Fox and Guansong Zhang

Northeast Parallel Architectures Centre,
Syracuse University,
111 College Place,
Syracuse, New York 13244-410
{dbc,gcf,zgs}@npac.syr.edu

February 13, 2000

Abstract

The article discusses various rules about use of distributed arrays in *HPJava* programs. These rules are peculiar to the HPspmd programming model. They can be enforced by a combination of static semantic checks, compile-time analysis and compiler-generated run-time checks. We argue that the cost of any necessary run-time checks should be acceptable, because, by design, the associated computations can be lifted out of inner loops.

1 Introduction

HPJava [3] is a Java language binding of a programming model we call the *HPspmd* model. This is a particular version of the general SPMD programming model that adds special support for distributed arrays of the kind defined by the HPF standard. The arrays are bound to the base language through a series of syntax extensions. The assumption is that the syntax extensions will be handled by a relatively simple preprocessor which emits an SPMD program in the base language.

Of course this implementation strategy has been followed with varying degrees of success in many translation systems for HPF and similar languages [7, 9]. The difference in the HPspmd approach is that it is assumed the source code is already written in an explicitly MIMD style. The HPspmd syntax provides only a thin veneer on low-level SPMD programming, and the transformations applied by the translator are relatively simple and direct—no non-trivial analysis should be needed to obtain good parallel performance. Meanwhile the language model provides a uniform model of a distributed array, which can be targetted by libraries for parallel communication and arithmetic.

SPMD programming has been very successful. There are countless applications written in the most basic SPMD style, using direct message-passing through MPI [10] or similar low-level packages. Many higher-level parallel programming environments and libraries assume the SPMD style as their basic model. Examples include ScaLAPACK [1], DAGH [13], Kelp [5] and the Global Array Toolkit [11]. While there remains a prejudice that HPF is best suited for problems with very 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 [4] and Global Arrays support unstructured access to distributed arrays.

These successes aside, the library-based SPMD approach to data-parallel programming lacks the uniformity and elegance of HPF. All the environments referred to above have some idea of a distributed array, but they all describe those arrays differently. Compared with HPF, creating distributed arrays and accessing their local and remote elements is clumsy and error-prone. 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. These observations motivate our introduction of the HPspmd model.

This article concentrates in particular on the issue of semantic checking in HPspmd languages. The basic features of the HPspmd language model are introduced in the Java context in section 2. Section 3 adds a discussion of *array sections*. This discussion serves to introduce the ideas of subranges and general process groups in HPspmd. Building on these ideas, section 4 describes some general rules about access to distributed array elements, and section 5 describes how a translator can ensure these conditions are met.

2 HPJava—an HPspmd language

HPJava [3] is an instance of our HPsmpd language model. 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. But a design decision is made that all access to *non-local* array elements should go through library functions—either calls to a collective communication library, or simply `get` and `put` functions for access to remote blocks of a distributed array[1].

A subscripting syntax is applied to distributed arrays to reference elements. But an array element reference must not imply access to a value held on a different processor. To simplify the task of the programmer, who must be sure accessed elements are held locally, the languages adds *distributed control* constructs. These play a role something like the `ON HOME` directives of HPF 2.0

[1]The distributed arrays are orthogonal to the sequential arrays of the base language—we deliberately keep them completely separate.

```
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.

and earlier data parallel languages [8]. One special control construct—a distributed parallel loop—facilitates traversal of locally held elements from a group of aligned arrays.

Array mapping is described in terms of a slightly different set of basic concepts from HPF. *Process group* objects generalize the processor arrangements of HPF. *Distributed range* objects are used instead HPF templates. A distributed range is comparable with a single dimension of an HPF template. These substitutions are a change of parametrization only. Groups and ranges fit better with our distributed control constructs.

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`. It represents a two-dimensional grid of processes. Similarly 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 over a specific process dimension. 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`.

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 constructs here iterate over all locations (selected by the degenerate interval " `:` ") of ranges `x` and `y`. The symbols `i` and `j` scoped by these constructs are *bound locations*. In HPF, a distributed array element is referenced using integer subscripts, like an ordinary array. In HPJava, with a couple of exceptions noted below, the subscripts in element references must be bound locations, and these must be locations in the range associated with the array

```
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.

dimension. This rather drastic restriction is a principal means of ensuring that referenced array elements are held locally.

The general policy is relaxed slightly to simplify coding of stencil updates. A subscript can be a *shifted location*. Usually this is only legal if the subscripted array is declared with suitable *ghost regions* [6]. 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. This example also illustrates application of a postfix backquote operator to a bound location. 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 dimensions can be ordinary integer expressions.

The short examples here have covered the basic syntax of HPJava. The language itself is relatively simple. Complexities associated with varied and irregular patterns of communication are dealt with in libraries, which can implement many richer operations than the `writeHalo` and `cshift` functions of the examples. The remaining important extensions to the language itself can be motivated most easily by considering the need to support Fortran 90 style *sections* of distributed arrays.

```
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) ;
    HPspmd.copy(b, tmp) ;
  }
}
```

Figure 3: A pipelined matrix multiplication program.

3 Array sections

HPJava has a syntax for representing subarrays. An *array section expression* has a similar syntax to a distributed array element reference, but uses double brackets. Whereas an element reference is a variable, an array section is an expression representing a new distributed array object. The 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. The HPJava implementation of an array section is closely related to the Fortran 90 idea of an array pointer—in Fortran an array pointer can reference an arbitrary regular section of a target array.

In the previous section it was stated that the subscripts in a distributed array element reference are either *locations* or (restrictedly) integer expressions. Options for subscripts in array section expressions are freer. For example, a section subscript is allowed be a triplet. In the simplest kinds of array section the rank of the result is equal to the number of triplet subscripts. If the section also has some scalar subscripts, this will be lower than the rank of the parent array. We would like to be able define the mapping of an arbitrary array section.

The mapping of a distributed array is defined by its *distribution group* and its list of ranges. In earlier examples the distribution group of arrays defaulted to the process group specified by the surrounding on construct. In general the distribution group can be specified explicitly by appending an "on" clause to the constructor itself:

new *type* [[x_0, x_1, ...]] **on** p

Here each of $x_0, x_1, \ldots$ is a range object or an integer (in which case the dimension is sequential), and p is a group contained within the set of processes that create the array. The ranges must be distributed over distinct dimensions of p. If there is any dimension of p which is not a distribution target of some range from $x_0, x_1, \ldots$, the array is *replicated* over that process dimension. The inquiries `grp` and `rng(int r)` return the group and ranges for any array[2].

Now consider array section `b` defined by

```
float [[,]] a = new float [[x, y]] on p ;

float [[,]] b = a [[0 : N - 1 : 2, 0 : N / 2 - 1]] ;
```

(see Figure 4). What are the ranges of `b`? In fact they are new kind of range called a *subrange*. For completeness there is a special syntax for constructing subranges directly:

```
Range u = x [0 : N - 1 : 2] ;
Range v = y [0 : N / 2 - 1] ;
```

The extents of both `u` and `v` are `N / 2`.

The *distribution group* of `b` can be identified with the distribution group of the parent array `a`. But sections constructed using a scalar subscript, eg

[2]For a sequential dimension the result of `rng(r)` is a member of the subclass `CollapsedRange`.

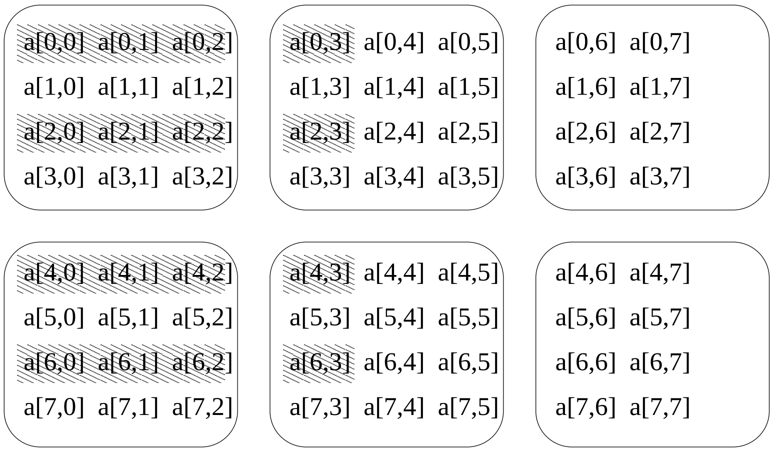

Figure 4: A two-dimensional section of a two-dimensional array (shaded area).

```
float [[]] c = a [[0, :]]
```

(see Figure 5) present another problem. Clearly `c.rng(0)` is `y`. But identifying the distribution group of `c` with `p` is not right. It would imply that `c` is replicated over the first dimension of `p`. Where does the information that `c` is localized to the top row of processes go?

We are driven to define a new kind of group: a *restricted group* is the subset of processes in some parent group to which a particular location is be mapped. The distribution group of `c` is defined to be the subset of processes in `p` to which the location `x[0]` is mapped. It can be written explicitly as[3]

```
p / x [0]
```

An equivalent definition of a restricted group is as some slice of a process grid, chosen by restricting some of the coordinates to single values.

The idea of a restricted group may look slightly ad hoc, but the implementation is quite elegant. A restricted group is uniquely specified by its set of effective process dimensions and the identity of the *lead* process in the group—the process with coordinate zero relative to the effective dimensions. The dimension set can be specified as a subset of the dimensions of the parent grid using a simple bitmask. The identity of the lead process can be specified through a single integer ranking the processes of the parent grid. So a general (restricted) HPJava group can be fully parametrized by a reference to the parent process grid together with just two `int` fields.

Now we can formally define of the mapping of a typical array section. As a matter of definition an integer subscript n in dimension r of array a is

[3] The expression `x[0]` is "pure syntax" in HPJava. There is no Java type associated with a location and no way to store a location value in a variable. The only named locations in HPJava are the bound locations scoped by overall (and *at*—see section 7). Besides group restrictions, expressions like `x[0]` can legally appear as array section subscripts or in the header of an at construct.

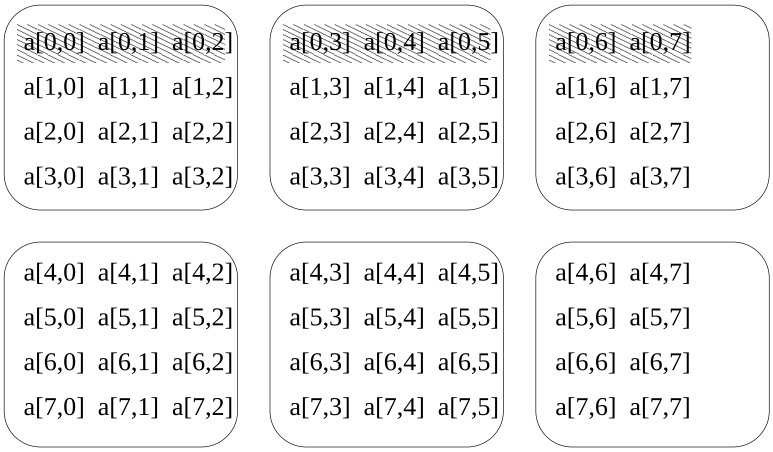

Figure 5: A one-dimensional section of a two-dimensional array (shaded area).

equivalent to a location-valued subscript a`.rng(`r`)[`n`]`. By definition, a triplet subscript l`:`u`:`s in the same dimension is equivalent to range-valued subscript a`.rng(`r`)[`l`:`u`:`s`]`. If all integer and triplet subscripts in a section are replaced by their equivalent location or range subscripts, and the location-valued subscripts are $i, j, \ldots$, then the distribution group of the section is

```
a.grp() / i / j / ...
```

and the sth range of the section is equal to the sth range-valued subscript.

Subranges and restricted groups can be used in array constructors on the same footing as the ranges and grids introduced earlier. This enables HPJava arrays to reproduce any alignment option allowed by the `ALIGN` directive of HPF.

4 Variable usage rules

A basic principle of the HPspmd model is that a program can only directly reference locally held variables. Therefore we want to define the conditions under which it is legal to access a particular variable (most especially, a particular array element) at a particular point in a program.

In a distributed array element reference, as explained in section 2, a subscript in a distributed dimension of an array must be a bound location, and this location must be a location in the appropriate range of the array[4]. Hence

Rule 1 *Subscripts in distributed array element references must be bound locations (unless the relevant dimension is sequential).*

[4] We can relax this restriction slightly by adding rules for identifying locations in certain closely related ranges. For example it is very natural to regard the locations in a subrange to be a subset of the locations in the parent range.

```
Procs2 p = new Procs2(P, P) ;

Range x = new BlockRange(N, p.dim(0)) ;
Range y = new BlockRange(N, p.dim(1)) ;

float [[,]] a = new float [[x, y]] on p ;

float [[]]  c = a [[0, :]] ;

on(p / x [N - 1])
  overall(j = y for :)
    c [j] = j` ;              // error!
```

Figure 6: Erroneous access to non-local elements of a section

and

Rule 2 *If a location appears as a subscript, it must be a location in the relevant range of the array.*

The rules just stated go a long way towards ensuring that only local elements are accessed, but they don't cover all cases. Consider the example of Figure 6. The subscripts on the element reference `c[j]` are legal—`j` is a location in `c.rng(0)` (which is equal to `y`). But, referring to Figure 5, the section `c` is localized to `p/x[0]`—the top row of processes in the figure—whereas the on construct specifies that the element assignments are performed in the group `p/x[7]`—the bottom row of processes.

Before we can give a general rule about accesses to elements of distributed arrays we need to refine the concept of the *active process group*. Suppose the set of processes currently executing the program text is represented by the group q, and the construct

`on(`p`)` S

appears. For the body, S, the active process group is changed to p. The group *p must represent a subset of the processes in q.* Similarly, suppose the current active process group is q, and the construct

`overall(`i `=` x `for` l `:` u `:` s`)` S

appears. For S the active process group is changed to q/i. Unless x is a collapsed range, the process dimension over which it is distributed *must be a dimension of q.*

Applied recursively these rules define the active process group at any point in a legal HPJava program.

Next we will define the *home group* of a program variable. The home group of an ordinary variable (not a distributed array element) is the group that was active at the point of declaration of the variable. For a distributed array element the rule is similar to the definition of distribution groups of section: suppose the subscripts in the reference

```
a [e0, e1, ...]
```

include locations $i, j, \ldots$, then the home group of the array element is[5]

```
a.grp() / i / j / ...
```

(if the array has a replicated distribution this group may contain several processes; otherwise it contains a single process).

An HPspmd rule for accessing a variable can be stated as follows:

Rule 3 *A variable can only be accessed when the active process group is contained in the home group of the variable.*

This is essentially a statement that all processes executing the current text must hold a copy of any variable accessed. Strictly speaking it is slightly stronger than the requirement that the local process holds a copy of any element it accesses. The version given here is convenient for formal discussion of programs.

5 Compiling usage checks

It is very important for the efficiency of HPJava programs that inner loops in the translated code—especially those derived from overall constructs in the source program—be kept as simple as possible. For example, the body of the emitted loop should avoid calls to methods associated with the runtime descriptor of the distributed array (unless, say, the programmer forced appearance of these calls by including them in the source). In particular runtime checking code associated with references to distributed array element should be lifted out of loops wherever possible.

Rule 1 can be enforced at compile time by a trivial extension to normal type checking. So far as Rule 2 is concerned, the intention is that run-time checks needed to enforce this rule should be lifted outside of the loop body. Consider this fragment

```
overall(i = x for l : u : s) {
  ...
    ... a [..., i, ...] ...
}
```

where i is the rth subscript in the list. Of course the expression a must be a distributed array. A naive insertion of runtime checks might lead to code something like

```
overall(i = x for l : u : s) {
  ...
    ASSERT(a.rng(r).containsLocation(x, i')) ;
    ... a [..., i, ...] ...
}
```

[5]As a matter of definition, for a shifted location p/(i ± *expression*) ≡ p/i. The rationale is that a shifted location is supposed to find an array element in the same process as the original location (albeit that the element could be in a ghost region).

with the runtime test appearing immediately before the element reference. What we want to achieve instead is something like

```
ASSERT(a.rng(r).containsSubrange(x, l, u, s)) ;
overall(i = x for l : u : s) {
  ...
    ... a [..., i, ...] ...
}
```

In fact we expect that in *typical* HPJava programs this transformation will be legal. In simple cases there may even be enough static information about the ranges of a to eliminate the run-time check altogether, because the lifted assertion can be proven at compile time.

There are cases where lifting the usage check may be problematic. Consider this example

```
Range y = x [1 : N - 2] ;
float [[]] a = float [[y]] ;

overall(i = x for 0 : N - 1) {
  ...
  if(i` > 0 && i` < N - 1)
    ... a [i] ...
}
```

The array is defined over some subrange of `x`. We iterate over the whole of `x`, but mask the disallowed accesses with a nested conditional construct. Such usage would block naive lifting of runtime range-checks out of the loop. The lifted range-check would cause an exception although the code is apparently legal. Another potential difficultly is that the expression a might not be a loop invariant with respect to the overall construct scoping the subscript. For example:

```
float [] [[]] s = new float [n] [[]] ;
... allocate individual distributed arrays in `s' ...

overall(i = x for :) {
  ...
  for(int j = 0 ; j < n ; j++)
    s [j] [i] = foo(j, i`) ;
}
```

The variable `s` is a Java array of HPJava distributed arrays. The array expression, a, is now `s[j]`. This expression is not loop invariant with respect to the overall construct. Without detailed (and not obviously practical) compile-time analysis the range check for the distributed array reference cannot be lifted out of the loops[6].

[6]Both examples would probably lead to inefficient translated code anyway, because various other computations associated with element reference cannot be lifted out of loops.

We assume these uses are idiosyncratic. In the first case the desired effect could be normally achieved more directly by changing the triplet in the overall header[7]:

```
overall(i = x for 1 : N - 2) {
  ...
  ... a [i] ...
}
```

In the second example the appearance of `i` as a common subscript for all distributed arrays in `s` strongly suggests that these have a common range, probably `x`. So replacing `s` with a two-dimensional distributed array,

```
float [[*,]] s = new float [[n, x]] ;
```

would probably be acceptable, and would remove the blockage to lifting the range-check.

Because efficient translation is a primary concern, and the examples above are difficult to translate efficiently, we outlaw them by adding a new rule

Rule 4 *Logically, the effects of any bound location subscripts appearing in a program are resolved at the head of the construct that scopes the location.*

Interpretations of this rule (which is more pragmatic than beautiful) include

a) a range check error may be thrown by the overall construct, even if later conditional code masks the element usage—so far as range checks are concerned the use is assumed to be unconditional—and

b) if a bound location subscript is applied to a distributed array expression (in an element reference or array section), the array expression must be invariant in the scope of the location.

The translator may apply suitable dataflow analysis to verify condition b)[8].

Next we consider Rule 3 of section 4. We assume that the translator makes the current state of the active process group visible in a variable of type `Group` called `apg`[9]. Most naively, the range checks for the example in Figure 6 could be inserted as follows:

```
float [[,]] a = new float [[x, y]] on p ;

float [[]]  c = a [[0, :]] ;

on(p / x [N - 1])
```

[7] We emphasize that the problem here is not with appearance of conditional code in general inside an overall construct—it is only with use of conditional code to mask element references otherwise forbidden by range-checking considerations.

[8] Dependence of the array expression on field variables (effectively global variables) may frustrate this analysis, if method calls also appear inside the loops.

[9] The variable `apg` is not a conventional Java global variable. It is more closely akin to the `this` expression of Java. It will be passed to HPJava-aware methods through a hidden argument.

```
overall(j = y for :) {
  ASSERT((c.grp() / j).contains(apg)) ;
  c [j] = j' ;
}
```

We note, however, that this can be optimized by lifting the assertion through the overall scoping `j`:

```
  on(p / x [N - 1]) {
    ASSERT(c.grp().contains(apg)) ;
    overall(j = y for :)
      c [j] = j' ;
}
```

and the lifted assertion is even simpler to compute, because the restriction by `j` is no longer needed.

In general in

```
overall(j = y for l : u : s) {
  ...
    ASSERT((a.grp() / ... / i / j / k / ... ).contains(apg)) ;
    ... a [..., j, ...] ...
    ...
}
```

provided the ellided code between the overall header and the assertion contains no statements that change `apg` (ie, no other overall or on headers) the assertion can be lifted and simplified as follows:

```
ASSERT((a.grp() / ... / i / k / ... ).contains(apg)) ;
overall(j = y for l : u : s) {
  ...
    ... a [..., j, ...] ...
    ...
}
```

We can legitimize the lifting, even if the element reference appears in conditional code inside the loop, by appealing to Rule 4. If overall constructs are suitably nested this transformation can be applied recursively.

Exact computation of the group `contains` operation may be expensive—in principle it is a set containment test. In the present case, where containment of the active process group is the issue, an alternative is available. If all members of the active process group make the assertion

```
ASSERT(p.contains(apg)) ;
```

this is essentially equivalent to them all making the assertion

```
ASSERT(p.amMember()) ;
```

The `amMember` method on a group returns true if the the local process is a member of the group and false otherwise. If the `contains` assertion is false, then at least one member of the active process group will throw an exception. If an exception is expected to abort the program globally, this is probably good enough. The `amMember` function can be computed very efficiently—in fact an extra boolean field should be added to the `Group` record containing this precomputed value, so it is simply a lookup.

Whether this approximation to the exact `contains` test is acceptable in practise is a slightly open issue. In "normal" styles of HPJava programming we expect that it is essentially equivalent to the exact set containment test. But in principle its adoption alters the semantics of the program. Replacing `contains` by `amMember` only weakens the requirement of Rule 3, which may not be a problem, unless Java-like exact exception handling is expected. In general our policy of lifting runtime checks is somewhat at odds with the Java ethos that exceptions should be thrown and control switched at exactly the point where the error (eg, the erroneous subscripting operation) occurred. In HPJava the most important thing is to get good performance, and probably some aspects of the strict Java exception model have to be sacrificed.

We did not discuss use of shifted locations as subscripts. They do not introduce any fundamental issues. A runtime inquiry on array ranges will be needed giving the size of the ghost extensions. Rule 4 can be interpretted to mean that use of non-invariant expressions for the shift-amounts is disallowed.

6 Other Usage Rules

In this section we mention two usage guidelines which have some relationship to the usage rules of the previous sections, but which are *not* required to make a legal HPJava program, and are not checked by the translator.

The first relates to a *coherence* property of variables. Informally, we will call a variable coherent if, at corresponding stages of execution of an SPMD program, processes belonging to the variable's home group always hold identical values in their local copies of the variable. If the home group of a variable is a single process, the variable is trivially coherent.

Giving a precise formal definition of coherence is quite difficult, because it is difficult to define formally what is meant by "corresponding stages of execution". What we can give is a precise rule which will keep all variables in a program coherent. This rule, which looks similar to usage Rule 3, is

Rule 5 *A variable should only be updated when the active process group is identical to the home group of the variable.*

Assuming that any external method calls also respect the coherence property[10] we assert that if this rule is applied throughout a program it will ensure all variable stay coherent. An HPspmd translater does not try to enforce this rule because the associated checks can be very expensive, and because judicious use of incoherence is often convenient.

[10] Examples of methods that do *not* respect coherence include functions like `MPI_Comm_rank`, which return results intrinsically dependent on process id.

It is often required that scalar arguments of collective operations should be coherent. There is another common requirement for arguments of collective operations—namely that all copies of all elements of distributed array arguments should be held in the group of processes that collectively invoke the operation. We will say that a distributed array a is *fully contained* (in the active process group) if a`.grp()` $\subseteq$ `apg`. Therefore a natural requirement is

Rule 6 *All distributed array arguments of collective operations should be fully contained.*

This rule is not required for all non-local operations. For example it is not required for `get` and `put` operations that implement one-sided communication. So it is not appropriate for the translator to try to enforce this rule. Instead checks such as

```
ASSERT(apg.contains(a.grp())) ;
```

should appear inside the implementation of collective library functions, as required.

7 Discussion

We have outlined a set of usage rules that HPspmd programs must follow to ensure that only locally available array elements are accessed directly. Special control constructs and other syntax in HPJava are designed to ensure that these rules can be enforced by a combination of static semantic checks, compile-time analysis and compiler-generated run-time checks. We showed how the language can be set up to ensure that the cost of any run-time checks required should not be prohibitive, even taking into account the complex index spaces of HPF-like distributed arrays.

The discussion has been in the context of a Java-based HPspmd language called HPJava. In [2] we have outlined possible syntax extensions to Fortran to provide similar semantics to HPJava. One feature of the HPJava language that was not discussed in this article is the *at construct*. This is very similar to the overall construct except that it specifies execution for a single location. For the purposes of this article it can be regarded as a special case of overall.

Two recent languages that have some similarities to our HPspmd languages are F-- and ZPL. F-- [12] is an extended Fortran dialect for SPMD programming. It does not incorporate the HPF-like idea of a global index space, so most of the issues discussed in the this article do not arise in F--. ZPL [14] is an array parallel programming language for scientific computations. ZPL uses pure array syntax for accessing its parallel arrays, and does not allow them to subscripted directly. So the range-checking issues are unlikely to be directly comparable to those in HPJava.

8 Acknowledgements

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

References

[1] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK User's Guide.* SIAM, 1997.

[2] Bryan Carpenter, Geoffrey Fox, Donald Leskiw, Xinying Li, Yuhong Wen, and Guansong Zhang. Language bindings for a data-parallel runtime. In *Third International Workshop on High-Level Parallel Programming Models and Supportive Environments.* IEEE Computer Society Press, 1998. http://www.npac.syr.edu/projects/pcrc.

[3] Bryan Carpenter, Guansong Zhang, Geoffrey Fox, Xiaoming Li, Xinying Li, and Yuhong Wen. Towards a Java environment for SPMD programming. In David Pritchard and Jeff Reeve, editors, *4th International Europar Conference*, volume 1470 of *Lecture Notes in Computer Science.* Springer, 1998. http://www.npac.syr.edu/projects/pcrc/HPJava.

[4] R. Das, M. Uysal, J.H. Salz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[5] Stephen J. Fink and Scott B. Baden. Run-time data distribution for block-structured applications on distributed memory computers. In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

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

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

[8] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, 1991.

[9] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steel, Jr., and M.E. Zosel. *The High Performance Fortran Handbook.* MIT Press, 1994.

[10] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tenessee, Knoxville, TN, June 1995. http://www.mcs.anl.gov/mpi.

[11] J. Nieplocha, R.J. Harrison, and R.J. Littlefield. The Global Array: Non-uniform-memory-access programming model for high-performance computers. *The Journal of Supercomputing*, 10:197–220, 1996.

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

[13] Manish Parashar and J.C. Browne. Systems engineering for high performance computing software: The HDDA/DAGH infrastructure for implementation of parallel structured adaptive mesh. In *Structured Adaptive Mesh Refinement Grid Methods*, IMA Volumes in Mathematics and its Applications. Springer-Verlag.

[14] Lawrence Snyder. A ZPL programming guide. Technical report, University of Washington, May 1997. http://www.cs.washington.edu/research/projects/zpl/.