# Intermediate Representations and Run-time Libraries for HPF Translation

D.B. Carpenter

*High Performance Computing Centre,*
*Southampton University,*
*Southampton, S09 5NH*

DRAFT

## 1 Introduction

This document is concerned with translation of *High Performance Fortran* (HPF) to a conventional sequential language. It is relatively informal, and by no means defines a complete translation scheme. However it does introduce a framework for such a scheme, in the form of an intermediate representation called *Adhoc*. It also defines the semantics of a high level run-time library for implementing HPF—namely that library needed to implement the primitives of Adhoc. By design, the necessary operations are much more transparent than for HPF itself. For example, there is no implicit data communication in Adhoc—all communication is performed in explicit data re-mapping operations. An explicit interface to the library, called *Mini-Adlib*, is given in C++[1]. Finally, many examples are given of translation from Adhoc to C++ with calls to Mini-Adlib.

The Adhoc notation is "human-readable". The idea is that particular phase of an our HPF translator will have some internal, *machine-readable* intermediate representation. This representation has a direct correspondence with an Adhoc program. In practise the machine-readable form would be translated directly to the target language. But for purposes of defining and illustrating the translation scheme, it is very valuable to have a textual notation.

Adhoc is a development of a notation described in [1]. Semantically it differs from the earlier language in various ways. For example it supports multiple active process arrays, whereas the earlier notation supported a single such array per scoping unit; it supports the kind of intra-dimensional alignments and embeddings required by HPF; and it introduces non-Cartesian distributed arrays

[1]Not because our translator will necessarily target C++—but because the interface can be specified most easily as a class library.

to support translation of the HPF *forall*. In the earlier notation the declaration of a process dimension (then called an *index*) had a specific association with spawning of physical processes. Adhoc dispenses with that association; the "processes" in its process array are virtual. A new process array is now interpreted as a different "map" (or "view") of a fixed SPMD physical process group.

Syntactically the new notation is modelled on C or C++, rather than occam. We should remark that, because the language is designed as a specification device rather than a compilable machine language, very little effort has been invested in eliminating trivial ambiguities, or avoiding clashes with other parts of C++.

Section 2 describes the Adhoc language. Section 3 is concerned with translation of HPF to Adhoc, and gives numerous examples. Section 4 discusses transliteration of Adhoc to a sequential language with calls to a run-time library which manages access to distributed data.

# 2 The Elements of Adhoc

Most of this section deals with "Vanilla Adhoc". This abstracts away various details of distribution format which are important in HPF. A more pragmatic distributed version is described in section 2.13. That version incorporates the same semantic ideas as the "vanilla" language, and adds information on distribution formats.

## 2.1 Process Arrays

The HPF declaration

```
!HPF$ PROCESSORS P (10, 10)
```

introduces a 10 by 10 arrangment of virtual processors, over which program data can be distributed. In Adhoc the role of the HPF processor arrangement is taken by the *process array*.

An Adhoc process array has a set of process dimensions. In Vanilla Adhoc—unlike HPF—process dimensions are unsized. A more important difference from HPF is that process arrays are initially declared *without* dimensions; dimensions are attached by subsequent declarations. In the fragment

```
procs p ;
dimension p_1 of p ;
dimension p_2 of p ;
```

a process array `p` is declared, then two dimensions of this array are declared, and named `p_1` and `p_2`. This fragment can be regarded as the translation of the single HPF statement above.

In the example above p_1 and p_2 exist as long as p does. In general though a process dimension may have a more restricted scope than the process array it belongs to; a dimension can appear and subsequently disappear during the course of program execution. This flexibility is needed because in Adhoc *every* dimension of a data array[2] is associated with some process dimension. So if we want to declare a rank 3 array within some scope, and distribute it over p, we first have to attach a new dimension to p.

Given the need such a mechanism, the purist approach is to use it the initial creation of multi-dimensional process arrays, as well as for creation of temporary dimensions (in the less pure *distributed* version of Adhoc, process arrays *are* declared with an initial set of dimensions; dimensions attached subsequently are associated with collapsed, on-processor mappings).

## 2.2 Ranges

A central idea of Adhoc is the *range*. There is no completely analogous object in HPF. A range is a one-dimensional index space. A range of extent $n$ has the values $0 \ldots n-1$ associated with it. The range also associates a process mapping with the elements of this index space; to this extent it resembles the HPF template, which also defines a mapping from an index space into distributed processor memories[3].

Suppose p_1 is a (process) dimension. Then

```
20 # p_1
```

is a *primitive range* of extent 20. This range is "distributed over" p_1. It can be named in a declaration:

```
range x(20 # p_1) ;
```

then referred to as x.

There is another kind of a range—the *derived range*. First we introduce the Adhoc version of a "triplet"—this looks slightly different to a Fortran triplet. The Adhoc triplet is characterised by an extent rather than an upper bound. For example

```
3 : base 2 step 5
```

represent the index values $2, 7, 12$. The base and step clauses are optional and default to 0 and 1 respectively, so

```
3 : base 2
```

represents $2, 3, 4$,

[2] Subsequently data array dimensions will nearly always be referred to as *ranges*, rather than "dimensions", to distinguish them from process dimensions.

[3] The exact correspondence is between a *primitive* range and an HPF template *dimension*.

```
3 : step 5
```

represents $0, 5, 10$, and

```
3 :
```

represents $0, 1, 2$. Using these triplets we can derive various ranges from `x`:

```
3 : base 2 step 2 : x
3 : base 2 : x
3 : step 5 : x
3 :: x
```

These ranges all have extent 3, so they have the same index range $0, 1, 2$. But they have different alignments to the primitive range. For the first of these ranges, for example, the 0 element is aligned with the 2 element of `x`, the 1 element with the 7, and the 2 element with the 12.

It is also allowed to derive a range from a derived range. Any range is characterised by the primitive range from which it is ultimately derived, and its alignment is defined with respect to that range. Two derived ranges are equivalent if they have to same (extent and) alignment, however they are derived.

Of course a derived range can be named, as in

```
range y(3 : base 2 step 2 : x) ;
```

## 2.3 Embeddings

An *embedding* is similar to a derived range, but is "scalar". It is aligned with a single element of some range. The embedding

```
3 : y
```

is aligned to element 3 of the range `x`. An embedding can be named in a declaration:

```
embed e(3 : y) ;
```

If `y` has primitive range `x`, `e` is said to be an embedding *in* `x`.

An *embedding set* is an unordered list of embeddings. The following embedding sets are equivalent[4]

```
{e, f, g}
{f, g, e}
{f, g} \/ {e}
```

[4]The last form, which is interpretted as a set union, is used in conjunction with the embedding-set inquiry function to establish certain alignments between arrays. See section 3.3 for an example.

## 2.4 Arrays

A *range list* is an ordered list of ranges and has the syntax

```
[x, y, z]
```

(assuming `x`, `y` and `z` are ranges)[5].

An *array* of rank $r$ has a range list of length $r$ and, optionally, an embedding set. The primitive ranges of all embeddings in the embedding set of an array, and of all ranges in its range list, must be distributed over distinct dimensions of the same process array.

A rank-0 `array` with empty embedding set is formally equivalent to a simple variable of the array-element type[6].

Suppose `t_1`, `t_2` and `t_3` are dimensions of some process array. Then we could have the following declarations

```
range i_a(10 # t_1) ;
range j_a(10 # t_2) ;
array int a [i_a, j_a] ;

range k_b(10 # t_3) ;
embed e_b(0 : k_b) ;
array float b {e_b} [i_a, j_a] ;

array int c {e_b, 9 : j_a} [8 : base 1 : i_a] ;
```

The arrays `a` and `b` have the same shape, but `b` is embedded at location 0 of the range `k_b`, which is distributed over dimension `t_3`. In HPF terms, `a` would be replicated in this process dimension. `c` is a rank 1 array, distributed over dimension `t_1`, and embedded in dimensions `t_2` and `t_3` [7].

*Every* dimension of an Adhoc array is associated with some process dimension. The effect of an HPF collapsed array dimension is achieved by creating an extra "temporary" process dimension, and aligning the array dimension to that.

Adhoc arrays can be subscripted in a way similar, syntactically, to Fortran. Unlike Fortran, the result of subscription is never a simple variable[8]. The signature of

```
a [2, 3]
```

[5] It may seem mysterious that ranges appear in ordered lists whereas embeddings appear in unordered sets. In the language of [1] there were no embeddings, but the analogues of ranges always appeared in *unordered* sets, as embeddings do here. It turns out that ordered lists of ranges make for easier translation of Fortran; but ordered embedding sets would be a nuisance.

[6] Or, perhaps more precisely, such an array can always be coerced to a simple variable, and vice versa.

[7] The `array` keyword distinguishes an Adhoc array from an ordinary C array.

[8] In Vanilla Adhoc—this statement is qualified in Distributed Adhoc. See section 2.13

is not simply `int`, but

```
array int {2 : i_a, 3 : j_a}
```

A simple subscript turns an array range into an embedding. A triplet subscript turns a range into a derived range, in an obvious way. It is also legitimate to use an embedding or range as a subscript, provided the latter is a subrange of the subscripted range of the array, and the former is an embedding in such a range. So the expressions

```
b [8 : base 2 : i_a, 4 : j_a]
```

and

```
b [8 : base 2, 4]
```

are equivalent; both have signature

```
array float {4 : j_a, e_b} [8 : base 2 : i_a]
```

It is convenient to allow the isolated symbol `:` as a subscript. This works exactly as in Fortran—it stands for the whole of the corresponding array range.

To summarise: subscripting an array doesn't involve any remapping (communication) of data. It yields a new `array` object. It *does not* provide access to the data in an array. Within Adhoc, the data in an array can only be accessed by using the `at` or `where` constructs described in the following sections.

As a very convenient shorthand, a range can be used to stand for a one-dimensional array containing its index values. Regarded as an array, the index `x` has signature

```
const array int [x]
```

and contains the values `0, ..., x.size() - 1`.

## 2.5 Embedding of control: *at*

In the preceding sections we have reinvented the HPF model of distributed data, using a slightly different set of primitive concepts, and a slightly different set of conventions. In this section major divergences from HPF start to emerge. Adhoc permits and requires explicit mapping of *control*, as well as mapping of data.

In Adhoc, elements of an array are accessed by using `at` and `where` constructs. This section deals with the simpler `at` construct, used for sequential access to array elements.

An `at` construction is built from an embedding set and a statement (possibly compound). The statement is executed on any processor which would hold data with the specified embedding. For example, if `x` is a range

```
array float a {3 : x} ;
array float b [x] ;
...
at({3 : x})
  a = 23 + b ;
```

assigns the sum of the 23 and the fourth element of `b` to `a`. We say that the construct *embeds control* in the process dimension associated with `x`. From a type-checking point of view, the way to think of this is that the `at` strips the embedding from `a`, and the range from `b` (focussing attention on the element of `b` with embedding `3 : x`). Formally, an `array` without embedding or range is equivalent to a simple variable. So, within the `at`, the variables `a` and `b` behave like a simple `float` objects. In particular ordinary C operators like `=` and `+` can be applied to them.

This is the oddest feature of Adhoc. Within `at` (and `where`) variables temporarily acquire types *different* from the ones with which they were declared. In the example above the construct turned the arrays into simple variables. In general it converts them to arrays with smaller embedding sets or range lists. So in

```
array float c [x, y, z] ;
array float d [y, x, z] ;
array float e {3 : x} [y, z] ;
...
at({3 : x})
  ...
```

`c`, `d` and `e` all have the effective signature

```
array float [y, z]
```

inside the `at` construct.

The set of variables that can be referenced inside the body of a particular `at` is restricted. First, a definition: if an `array` is distributed over some process array, and the process array has a dimension which doesn't appear in any range or embedding of the data array, the data array is *replicated* over that process dimension. By convention, a simple variable is replicated in every dimension of every process array. Now, consider a restricted `at` construct with a singleton embedding set (containing only one embedding):

```
at({e})
  S
```

An externally declared variable `v` may be references in `S` if *one* of the following conditions holds

1. *outside* the `at`, `v` is an array distributed over the process array which `e` embeds in, and `e` appears in the embedding set of `v`, *or* `v` has a range with an element aligned to `e`.
2. *outside* the `at`, `v` is replicated over the process dimension associated with `e` and, in `S`, `v` *is only accessed by value*.

Case 1 is illustrated by the references to `a` and `b` in the example above: the embedding set of `a` contains the only embedding specified by the `at`; the range of `b` has an element aligned to that embedding. Case 2 is fairly self-explanatory. The example above would remain legal if the 23 was replaced by the name of a variable replicated in the dimension of `x`—for example a simple variable[9].

These rules extend to general `at` statements by identifying

```
at({e_1, ..., e_n})
  S
```

with

```
at({e_1})
  ...
    at({e_n})
      S
```

So, for example

```
array float a {3 : x} ;
array float b {3 : x} [y] ;
...
at({3 : x, 1 : y})
  b = a ;
```

is equivalent to

```
array float a {3 : x} ;
array float b {3 : x} [y] ;
...
at({3 : x})
  at({1 : y})
    b = a ;
```

We have described the peculiarities of array reference inside `at`. Reference to *ranges* inside these constructs are subject to analogous rules. These rules

[9] From an implementation point of view, this makes sense if replicated variables are literally stored in a replicated fashion. On the other hand it wouldn't make sense to *assign* a simple variable inside an `at`, because the replicated copy of the variable needs to be updated everywhere, not just where the `at` statement is executed.

can be inferred by regarding an array as a one-dimensional array, as explained at the end of section 2.4. Suppose an `at` specifies an embedding in some range with primitive range is `x`. Outside the construct `y` is a named range, derived from `x`. Inside the construct, `y` can be referenced, *but* in that context it means a simple integer value—the associated index value of the range.

This regime automatically rules out nested `at`s which embed control in the same process dimension

```
at({3 : x})
  at({1 : x})
    ...
```

Inside the outer construct `x` "becomes" the simple integer value 3. It cannot be used as the target of an embedding. This is just as well, because it is contradictory for a statement to be executed only at location 3 *and* only at location 1!

Finally it is necessary to place restrictions on what functions can be called inside an `at`. If the `at` embeds control in a particular process dimension, it is illegal to call a function which directly or indirectly references that dimension (for example, through a global name). Similarly, no data distributed over a process array *other* than the one specified by the `at` can be accessed in the procedure.

## 2.6 Distribution of control: *where*

If `at` is like a conditional, `where` is like a loop (the constructor names are supposed to be reminiscent of `if` and `while`). Instead of an embedding set it takes a range[10]. The statement in the body of a `where` construct is executed for every element of the range, notionally in parallel (in practise the loop is split into sequential subranges processing data local to each processor).

The fragment

```
where(x)
  S
```

is *logically* equivalent to the fragment

```
for (int i = 0 ; i < x.size() ; i++)
  at({i : x})
    S
```

[10] Given that `at` specifies an embedding *set*, one might expect `where` to take a range *list*. To translate HPF `at` must take a set, because the size of the embedding set of arrays (specifically dummy arguments) isn't always known statically. To process such arrays we need a construct which takes embedding sets of arbitrary size. It *is* always known at compile time how many ranges any array has (this number is simply the rank of the array) so there's no practical reason why a `where` must take an arbitrary-length list. Translation of Adhoc is simplified if `where` it specifies a single range.

provided the statement `S` satisfies a set of restrictions, required for a correct program. The restrictions on the body of a `where` are very similar to the ones given in the previous section (in fact a little simpler, because the `where` specifies a single range) and can be deduced from them through the equivalence above. They won't be enumerated here.

## 2.7 *remap*

None of the operations discussed so far have required communication of data. Adhoc has a small handful of primitives for communication. The simplest is `remap`. Syntactically `remap` looks like a procedure call with two arguments. These are two arrays which must have the same shape, but will normally have different mappings. They could be distributed over different process arrays, or they could have the same process array with differently aligned ranges, or different embeddings. `remap` simply copies the values in its second argument into its first argument.

Here is an example where the arguments of `remap` share a process array

```
range x(12 # t_1) ;
array float u [x] ;
range x_internal(10 : base 1 : x) ;
array float n [x_internal] ;
array float p [x_internal] ;

...

for( ; ; ) {
  remap(n, u [10 : base 0]) ;
  remap(p, u [10 : base 2]) ;
  where(x_internal)
    u = 0.5 * (n + p) ;
}
```

The values in the two sections of `u` are copied into the arrays `n` and `p`.

`remap` has a poorer cousin—`broadcast`. The first argument of `broadcast` is a pointer to a simple C variable. The second argument is an `array`. If the `array` has rank 0, it's single data item is copied to the simple variable addressed by the first argument. It is also reasonable to permit higher rank `arrays`, in which case all data in the `array` is copied to the simple vector pointed at by the first argument, sequenced in some specified way. [BUT WHAT IF THE ARRAY IS NON-CARTESIAN?.]

## 2.8 *gather* and *scatter*

The most general communication patterns provided by Adhoc are achieved with the primitives `gather` and `scatter`. Again these look syntactically like procedure calls. The first argument is the "gathered data"; the second argument is the "scattered data"—an array of rank $r$. This argument is followed by $r$ further arguments—the subscripts. The subscripts are arrays with elements of type `int`.

The gathered data and the subscripts must *conform*, in the following sense: if any subscript has a particular range, the gathered data must also have that range. Likewise, if any subscript has a particular embedding, the scattered data must also have that embedding.

Note well: the *number* of subscript expressions is determined by the rank of the *scattered data*; their *mapping* is constrained by agreement with the *gathered data*.

The *gather* operation copies data from the second argument to the first. The *scatter* operation copies data from the first argument to the second.

In the following example the data from square $n \times n$ array is copied to an $n^2$ array.

```
range x_a(n # t_1) ;
range y_a(n # t_2) ;
array float a [x_a, y_a] ;

range x_b(n * n # s_1) ;
array float b [x_b] ;

...

array int subscript [x_a, y_a] ;
where(x_a)
  where(y_a)
    subscript = n * x_a + y_a ;
scatter(a, b, subscript) ;
```

The scattered data is `b`—a rank-1 array—so there is just one subscript expression. The code here is equivalent to the HPF

```
REAL A (0 : N - 1, 0 : N - 1) DISTRIBUTE T (BLOCK, BLOCK)

REAL B (0 : N * N - 1) DISTRIBUTE S (BLOCK)

...

FORALL (X = 0 : N - 1, Y = 0 : N - 1)
```

```
  B (N * X + Y) = A (X, Y)
END
```

Replacing the `scatter` operation by a `gather` would be equivalent to reversing the direction of the assignment in the HPF.

Another example, with two subscripts this time. Suppose `a` is declared as above, then...

```
array float c [x_a] ;

...

gather(c, a, x_a, x_a) ;
```

is equivalent to the HPF

```
REAL C (0 : N - 1) ALIGN A ( : , *)

...

FORALL (X = 0 : N - 1)
  C (X) = A (X, X)
END
```

We exploited the shorthand which allows a range to stand for a vector—the range `x_a` is used for the two equal subscript expressions.

Finally, if `a` is declared as above, and $m \leq n/2$...

```
range x_d(m :: x_a) ;
range y_d(m :: y_a) ;
array float d [x_d, y_d] ;

...

array int subscript2 [x_d, y_d] ;
where(x_d)
  where(y_d)
    subscript2 = x_d + y_d ;
gather(d, a, x_d, subscript2) ;
```

is equivalent to the HPF

```
      REAL D (0 : M - 1, 0 : M - 1)
!HPF$ ALIGN D WITH A (0 : M - 1, 0 : M - 1)

      ...
```

```
FORALL (X = 0 : M - 1, Y = 0 : M - 1)
  D (X, Y) = A (X, X + Y)
END
```

This illustrates how subscripts can have different rank to the gathered data, so long as they conform, in the sense explained above.

Why not introduce some neat syntax for `gather` and `scatter`, to make them look more like conventional subscripting operations—and less intimidating? Mainly to emphasise that these are big, expensive operations, implemented in a run-time library. We don't particularly *want* to make them easy to use! In this connection it's worth pointing out that in the last example one could achieve the same effect—almost certainly more efficiently—with the simpler `remap` operation. `a` and `c` are declared as above, then...

```
where(x_d)
  remap(d, a [m : base x_d : y_a]) ;
```

Inside the `where`, `d` behaves like a vector with range `y_d` and `a` behaves like a vector with range `y_a` (the `where` range `x_d` is a subrange of `x_a`, so `a` can properly appear here). `x_d` behaves like a simple integer value, so it can appear as the base of the section subscript on `a`. Of course, we can do a similar thing in HPF:

```
FORALL (X = 0 : M - 1)
  D (X, : ) = A (X, X : X + M - 1)
END
```

`gather` and `scatter` have an optional final argument—an array of type `int`, conforming with the gathered data. This is a mask, controlling assignment. The assignment is disabled where the mask is zero.

Notice that if `a` and `b` are rank-`r` arrays with the same shape, the following operations all have the same effect

```
remap(a, b) ;

gather(a, b, a.rng(1), ..., a.rng(r)) ;

scatter(b, a, b.rng(1), ..., b.rng(r)) ;
```

## 2.9 Procedures

The procedure interface should provide static information about the arguments, which assists in efficient compilation of both the procedure and the calling program.

The external interface of an HPF procedure can contain an extraordinary amount of detailed information. The shape and alignment parameters of some arguments can depend on run-time values of other arguments, or of global variables, through arbitrary "specification expressions"—more restrictive than ordinary Fortran expressions, but not by a great deal. This is different from most languages—for example C++—where the external interface only depends type information, always available at compile time.

Adhoc attempts to capture as much static information as practical from the original Fortran program, so it has a comparably complex procedure interface. It also allows attributes of arguments to depend on other arguments through arithmetic expressions and various inquiry functions. Like Fortran, Adhoc will require a restricted class of expression which can be used in this context. We don't intend to define this class here (probably it would be simpler than the Fortran specification expression, just allowing scalar integer arithmetic and some inquiry functions). The main novel feature of Adhoc relative to HPF is the systematic use of a "wild-card" mechanism to put incomplete information in the interface.

The identifier _ is reserved for a wild-card, which can stand for

- an integer expression,
- a process dimension,
- a triplet,
- a range, or
- an embedding set.

Array signatures appearing in the formal parameter list can involve _ in various places. For example the interface

```
void foo(array int a _ [_, _]) {
  ...
}
```

specifies that `foo` has a rank-2 array argument, but leaves its embedding and ranges unspecified. We could have specified just the size of these ranges, as in

```
void goo(array int a _ [10 :: _, 10 :: _]) {
  ...
}
```

or we could have specified empty embedding, and primitive ranges distributed over specific globally defined process dimensions...

```
void hoo(array int a [_ # t_1, _ # t_2]) {
  ...
}
```

...or ranges derived in an unspecified way from primitive ranges over the same process dimensions...

```
void koo(array int a [_ : _ # t_1, _ : _ # t_2]) {
  ...
}
```

...and so on[11]. Many further examples are given in section 3.3.

There is no analogue of HPF "prescriptive" mappings. Actual parameters must match the patterns specified in the formal parameter. Section 3.3 shows the various effects of the HPF dummy argument mappings are reproduced.

## 2.10 Allocatable and non-Cartesian arrays

Sometimes it is necessary to defer allocation of an array. For example, one may want to achieve the effect of a function returning an array whose shape is defined within the function call. Adhoc doesn't directly support `array`-valued functions[12]. But one can pass an allocatable array to a procedure, and have the procedure allocate it.

The declaration of an allocatable array uses the keyword `alloc` rather than `array`, and the embeddings and ranges can use wildcards in the same way as the signatures of formal parameters. Analysis of an Adhoc program is simplified if the *process dimensions* at least, are specified by the patterns used in the `alloc` declaration[13]. For example

```
alloc float a [_ # t_1, _ # t_2] ;
```

To allocate `a` we use the special syntax `<- new`...

```
a <- new array float [10 # t_1, 10 # t_2] ;
```

Subsequently `a` can be used exactly like an ordinary array, until it is deallocated with the `delete` operator:

```
delete a ;
```

A "non-Cartesian array" is a non-rectangular array. It is essentially an array of arrays. The declaration syntax is illustrated by

```
range x_b(10 # t_1) ;
range y_b(10 # t_2) ;

array float b [x_b, y_b] [_ # t_3] ;
```

[11] In these examples the syntax rules of Adhoc provide an unambiguous classification of each underscore into one of the five categories enumerated above. This is always possible.

[12] Adhoc is meant to be a relatively low-level target language. Array-valued functions are high-level features; the compiler has to allocate array temporaries for them.

[13] Otherwise it may be difficult to statically determine properties as fundamental as the effective rank of the array within `at` or `where` contructs.

which is a two-dimensional array of one-dimensional arrays (embedding lists are optionally permitted before each range list). More precisely, it is an `array` of `alloc` objects, so incomplete specifications of ranges can appear in "inner" embedding sets and range lists. The elements of a non-Cartesian array must to be allocated with the `new` operator. For example

```
where(x_b)
  where(y_b)
    b <- new array float [x_b + y_b # t_3] ;
```

which sets the extent of the third dimension of `b` to be some function of the indices in the first two dimensions. The elements must be deleted in a matching construct:

```
where(x_b)
  where(y_b)
    delete b ;
```

An extra restriction is needed concerning access to non-Cartesian arrays: a non-Cartesian array cannot be accessed inside an `where` or `at` construct specifying a process dimension associated with an *inner* range or embedding of the array. Does that render the data in non-Cartesian arrays forever inaccessible? Happily not, because within the context of the loops

```
where(x_b)
  where(y_b)
    ...
```

(for example) the variable `b` behaves like a one-dimensional Cartesian array, and is accessible by a loop specifying `t_3`. The new restriction only forbids access to `b` in a `where` or `at` specifying `t_3` *if* the `t_1` and `t_2` context has not been specified by enclosing `at` or `where` constructs, to "convert" the array to a Cartesian one.

The `remap` call can be applied to non-Cartesian arrays. Both arguments of `remap` must be fully allocated before the call. They must have the same detailed shape—the functions defining higher level extents in terms of lower level indices must be identical for the two arrays.

In `gather` and `scatter` the gathered data and the subscripts may be conforming non-Cartesian arrays. The *scattered data* should always be a simple Cartesian array [THAT MEANS THAT 'REMAP' FOR A NON-CARTESIAN ARRAY CANNOT BE EXPRESSED IN TERMS OF 'GATHER' OR 'SCATTER']. In this context *conforming* means that embedding sets and range lists are identical, except at the innermost level, where the weaker constraints for conforming Cartesian arrays apply[14].

[14] This wording is not unambiguous. It is arguable that the signature `array float [x, y] [z]` conforms with `array int [x]` because the innermost range list of the latter conforms with the *corresponding* range list of the former, according to the earlier rules

## 2.11 IO

The procedures `input` and `output` take a single array argument.

## 2.12 Miscellany

The communication primitives introduced so far are sufficient to translate any HPF program which doesn't use array intrinsics (and many that do). Efficient translation of some the array intrinsics need further operations; for example, reduction operations. Most of these can be specified by Adhoc procedure interfaces, and implemented as library functions.

Inquiry functions are provided for:

- extracting an embedding set from an array,
- extracting a range, identified positionally, from an array,
- extracting the extent of a range,
- extracting the process dimension from a range, and
- extracting a process dimension, identified positionally, from a process array.

## 2.13 Distributed Adhoc

*Distributed Adhoc* allows declarations like

```
procs p(4, 4) ;
dimension p_1(p.dim(1)) ;
dimension p_2(p.dim(2)) ;
```

This combination of declarations is supposed to be "semantically equivalent" to

```
procs p
dimension p_1 of p ;
dimension p_2 of p ;
```

in Vanilla Adhoc. With the new syntax `p_1` and `p_2` are "distributed dimensions".

The syntax for primitive ranges is changed along the following lines...

```
range x(10 #blk    p_1) ;
range y(10 #cyc    p_2) ;
range i(10 #blk(2) p_1) ;
range j(10 #cyc(2) p_2) ;

range z(10 #* p_3) ;
```

The first four declarations reflect the four distribution formats in HPF. The last declaration introduces a "collapsed range". We insist that collapsed ranges cannot be created over distributed dimensions—in the example, `p_3` may not be a distributed dimension. By definition, a range derived from a collapsed range is also collapsed (but not *primitive* collapsed—there is still a vestigial alignment relation between collapsed ranges, so that the control-mapping constructs can be meaningfully applied over collapsed ranges).

Semantically the new operators `#blk`, `#cyc`, etc, and `#*` are identical to the `#` of Vanilla Adhoc *with the following exception*: in Distributed Adhoc an embedding in a *collapsed* range is regarded as a *null embedding*. Two arrays whose mappings differ only by null embeddings are considered to have equivalent mappings; a null embedding in the embedding set of an `at` constructor can be omitted without changing the meaning of a program; finally, an array section which in Vanilla Adhoc would yield an `array` with only null embeddings (and an empty range list) *can be used like a simple variable* in Distributed Adhoc. For example

```
array float a [10 #* p_3] ;
...
a [3] = 3 + a [2] ;
```

is legal in Distributed Adhoc—an array whose ranges are collapsed can be used in much the same way an ordinary C or Fortran array[15].

# 3 Translation of "Mapped" HPF to Adhoc

This document is too short to spell out a detailed translation scheme from HPF-like languages to Adhoc. Instead, we attempt to justify the claim that Adhoc is a viable target for HPF translation by taking a series of fragments of HPF code and giving equivalent Adhoc programs.

## 3.1 Anatomy of a translator

First, a rough sketch of how a real translator might be arranged. Figure 1 illustrates a possible organisation. AST stands for Abstract Syntax Tree. The figure looks like a textbook model of a simple compiler, except for the insertion of the *mapper* phase.

The *front end* is responsible for syntactic and static semantic analysis of the input HPF. It typically outputs symbol table information, containing descriptor

[15] The introduction of null embeddings means that the semantics of Distributed Adhoc is strictly larger than Vanilla Adhoc. It would be nice to say that any Distributed Adhoc program can be trivially mapped to a Vanilla Adhoc program. This nice feature was sacrificed by introducing null embeddings. One could fix that by adding the idea of a collapsed range to Vanilla Adhoc—we preferred not to complicate its semantics—or by forgoing null embeddings in Distributed Adhoc—but they are rather convenient.

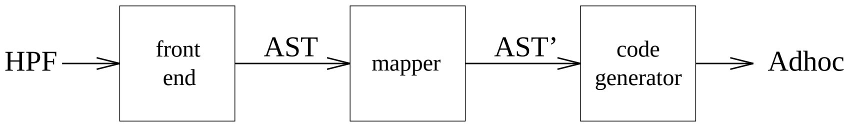


Figure 1: Schematic of an HPF to Adhoc translator

records for the entities in the source program. This table is integrated with some abstract syntax tree—a suitably reorganised parse tree, probably decorated with extra information, such as the type of expression nodes.

In our model this analysis is followed by "expression mapping". The *mapper* takes the analysed program and decides how all expression (and subexpression) nodes are to be mapped to the parallel computer[16]. The programmer typically attributes the named variables in an HPF program with some mapping. But a compiler also has to generate many expression temporaries. The problem of where to store these temporaries now arises—ie how they are mapped to the computer. This is directly related to the question of where individual computations are performed.

Mapping of expression temporaries is one of the most problematic aspects of HPF translation. In a real translator the the mapper would decorate each expression node of the abstract syntax tree AST with a mapping descriptor (in general two mapping descriptors—defining the point where the expression is calculated, and the point where its value is to be "consumed"). The updated program representation is labelled AST$'$ in figure 1.

For the examples in this section we assume that a mapping has already been chosen. This mapping will typically be indicated roughly in words. The subsequent translation to Adhoc should make clear *exactly* what mapping was intended.

In a real translator, the logical equivalent of this transliteration from mapped HPF to Adhoc is performed by the *code generator* phase.

A warning: many of the examples were deliberately contrived so that they are difficult to translate. For this reason the Adhoc programs will often be quite long, and look terribly inefficient. *Simple* HPF programs generally translate to simple Adhoc, but we have to live with the awkward fact that HPF can rather easily express many operations which are extremely difficult to translate efficiently.

## 3.2 Mapping of program variables

**Example 1**

[16] We use the term "expression" more broadly than in the Fortran definition—closer to the C idea of an expression. It includes procedure calls, assignments, forall constructs, read and write statements, and so on. "Expression mapping" covers all these cases.

```
      PROCESSORS P (4, 4)

      INTEGER a (16, 16)
!HPF DISTRIBUTE a (BLOCK, BLOCK) ONTO p

      INTEGER b (8, 4)
!HPF ALIGN b (i, j) WITH a (5 + j, 1 + 3 * i)

      INTEGER c
!HPF ALIGN c WITH b (2, *)
```

A translation where ranges are named...

```
procs p(4, 4) ;
dimension p_1(p.dim(1)) ;
dimension p_2(p.dim(2)) ;

range i_a(16 #blk p_1) ;
range j_a(16 #blk p_2) ;
array int a [i_a, j_a] ;

range i_b(8 : base 5 : j_a) ;
range j_b(4 : base 3 step 3 : i_a) ;
array int b [i_b, j_b] ;

array int c {2 : i_b} ;
```

A completely equivalent translation which doesn't explicitly name ranges...

```
procs p(4, 4) ;

array int a [16 #blk p.dim(1), 16 #blk p.dim(2)] ;

array int b [8 : base 5 : a.rng(2),
             4 : base 3 step 3 : a.rng(1)] ;

array int c {2 : b.rng(1)} ;
```

All Adhoc index ranges start at 0, whereas in Fortran the default is 1. Otherwise the 3 Adhoc arrays are in direct correspondence with the HPF arrays.

### 3.3 Mapping of dummy arguments

The HPF procedure interface is fairly clunky. There are three kinds of mapping directives for a dummy argument

- Prescriptive
- Descriptive
- Transcriptive

The exact distinction between the "prescriptive" and "descriptive" cases is not entirely clear cut. The rough idea is that a called procedure should be prepared to remap a prescriptively mapped dummy argument on entry, to achieve the desired mapping; the actual argument may have a different mapping. On the other hand a procedure is not required to remap a descriptively mapped dummy—either the programmer or the compiler arranges that the actual arguments have the stated mapping.

It is expected, though, that if the interface of the procedure is explicit in the calling program, the *caller* will perform the remapping of a *prescriptively* mapped argument (on the grounds that it has static information about the original mapping of the actual argument expression, so can perform the remapping more efficiently).

It is a *requirement* on the compiler that *if* the interface of the procedure is explicit in the calling program, and a *descriptive* directive appears for a dummy, the code generated for the caller remaps the actual argument before the call, so that the called procedure functions properly.

Hence, when the interface is explicit (the normal situation, one would hope, in a well-written Fortran 90 program) there is no discernable difference between the effect of prescriptive and descriptive mappings[17].

Notice, in any case, that the dummy argument remapping behaviour inside a given HPF procedure generally depends on whether or not the caller has an explicit interface for the procedure. To meet the requirements of the HPF specification efficiently it may be necessary to generate two different versions of the code for each procedure: one called when the interface is explicit, and one called when it isn't. We will work on this assumption below.

The meaning of *transcriptive* directives is relatively straightforward: they leave the mapping (or aspects of the mapping) unspecified in the called procedure. The procedure must be compiled to accept data with *any* legal HPF mapping, chosen by the calling program.

[17] In fact, the generated code for the procedure may reverse the naive expectation. Most obviously a prescriptively mapped dummy may not require code to perform remapping in the translated procedure, because the caller has already done the remapping. More subtly a descriptively mapped dummy might actually need such code. This could happen if, say, a descriptive alignment directive was parameterised by a very complex specification expression. The compiler may decide that it is too expensive to take the evaluation of this expression out of its natural context—in the code for the declaration block of the called procedure—and place it in the code of the caller (recall that a Fortran specification expression can depend on attributes of locally declared variables, can involve complex intrinsics, and so on). The only way to meet the requirements of the HPF specification in this case is if the implied remapping of the actual argument expression is actually done in the code for the called procedure (because the caller cannot calculate the target mapping).

The following examples are taken from chapter 5 of the *High Performance Fortran Handbook*. We adopt the convention of prepending and underscore to names of the translated functions versions called when the interface is *not* explicit in the calling program, where they differ from the version called when the interface is explicit. Also an underscore is prepended to un-remapped dummy arguments passed in to the functions.

**Example 1**

```
      SUBROUTINE MINGO (M)
      REAL M (100, 100)
!HPF$ DISTRIBUTE M (BLOCK, BLOCK)
      ...
      END
```

This is a very simple example of a prescriptive mapping. First the Adhoc version called when the interface is explicit in the calling program:

```
  void mingo(array float m [100 #blk def2d_1, 100 #blk def2d_1])
    ...
  }
```

The caller has already remapped `m` to the expected mapping, if necessary[18]. So no special code is needed in the body of `mingo`. If the interface is *not* explicit in the caller we have[19]

```
  void _mingo(array float _m _ [100 :: _, 100 :: _]) {
    array float m [100 # def2d_1, 100 # def2d_2] ;
    remap(m, _m) ;
    ...
    remap(_m, m) ;
  }
```

In these translations we suppose `def2d` is a default two dimensional process array with dimensions `def2d_1` and `def2d_2`. These identifiers are globally declared.

**Example 2**

```
      SUBROUTINE RICKARD (R)
      REAL R (100, 100)
!HPF$ DISTRIBUTE * R (BLOCK, BLOCK)
```

[18] In reference to the comments in footnote 17, we can certainly rely on the caller to do that here, since the mapping directive is so simple.

[19] If the Fortran argument had the `IN` or `OUT` attribute one of the two `remap` operations could be omitted.

```
      ...
      END
```

The mapping is descriptive. In this case only one translation is needed

```
  void rickard(array float r [100 #blk def2d_1,
                              100 #blk def2d_2])
    ...
  }
```

This is called whether or not the interface is explicit. If it isn't, and the actual argument doesn't have the expected mapping, the program will probably fail (it is illegal Adhoc). That's fine, because the HPF program was erroneous.

**Example 3**

```
      SUBROUTINE JAFFEE (J)
      REAL J (100, 100)
!HPF$ DISTRIBUTE J * ONTO *
!HPF$ INHERIT J
      ...
      END
```

This is a transcriptive mapping. Again, only one translation is needed

```
  void jaffee(array float j _ [100 :: _, 100 :: _]) {
    ...
  }
```

**Example 4**

```
      SUBROUTINE MELVIN (AXOLOTL, POIUYT)
      REAL AXOLOTL ( : , : ), POIUYT ( : , : )
!HPF$ INHERIT, DISTRIBUTE * (BLOCK, BLOCK) :: AXOLOTL
!HPF$ ALIGN POIUYTT ( : , : ) WITH * AXOLOTL ( : , : )
      ...
      END
```

The translation is

```
  void melvin(
      array float axolotl [_ : _ #blk def2d_1,
                           _ : _ #blk def2d_2],
      array float poiuytt [axolotl.rng(1), axolotl.rng(2)]) {
    ...
  }
```

The Adhoc makes explicit a (particularly ugly) feature of HPF. The `INHERIT` attribute means that the `DISTRIBUTE` directive applies to the *template* of `AXOLOTL`, so the ranges of `axolotl` are *derived* ranges. If the HPF mapping of `axolotl` had been just

```
!HPF$ DISTRIBUTE * (BLOCK, BLOCK) :: AXOLOTL
```

without `INHERIT`, the more restricted (and therefore more contentful) Adhoc translation

```
void melvin(array float axolotl [_ #blk def2d_1, _ #blk def2d_2],
              ...) {
    ...
  }
```

would be valid. Notice that a descriptive `DISTRIBUTE` directive can only be used in conjunction with an `INHERIT` directive *if* the dimensions of the argument are in direct one to one correspondence with the dimensions of its template. If in fact `AXOLOTL` was aligned to one-dimensional template, say, or to a two-dimensional template with dimensions transposed—both perfectly acceptable HPF mappings—then HPF provides no means of presenting the extra distribution information to the compiler. In contrast Adhoc would permit something like

```
  void melvin(
      array float axolotl [_ : _ #blk def1d_1, _ : _ #* _],
      ...) {
    ...
  }
```

or

```
  void melvin(
      array float axolotl [_ : _ # def2d_2, _ : _ # def2d_1],
      ...) {
    ...
  }
```

to represent such cases. In the first case we suppose `def1d` is a globally declared default one-dimensional process array, with distributed dimension `def1d_1`.

If the alignment of `POIUYTT` is replaced by

```
!HPF$ ALIGN POIUYTT (I, J) WITH * AXOLOTL (I, J)
```

(in which case `POIUYTT` may be a different shape to `AXOLOTL`—aligned to its lower corner) the translation is changed to

```
  void melvin(
      ...
      array float poiuytt [_ :: axolotl.rng(1),
                           _ :: axolotl.rng(2)]) {
    ...
  }
```

**Example 5 (p150)**

```
      REAL DOUGH (100)
!HPF$ DISTRIBUTE DOUGH (BLOCK (10))
      CALL PROBATE (DOUGH (7 : 23 : 2))
      ...
      SUBROUTINE PROBATE (BREAD)
      REAL BREAD (9)
!HPF$ INHERIT BREAD
```

Translates to

```
  array float dough [100 #blk(10) def1d_1] ;
  probate(dough [9 : base 6 step 2]) ;
  ...
  void probate(array float bread _ [9 :: _]) {
    ...
```

The fact that Adhoc range indices start at zero, and triplets are defined by extent rather than upper bound, introduces some differences from the HPF.

**Example 6 (p152)**

```
      LOGICAL FRUG (128), TWIST (128)
!HPF$ PROCESSORS DANCE_FLOOR (16)
!HPF$ DISTRIBUTE (BLOCK) ONTO DANCE_FLOOR :: FRUG, TWIST
      CALL TERPSICHORE (FRUG (1 : 40 : 3), TWIST (1 : 40 : 3))

      SUBROUTINE TERPSICHORE (FOXTROT, TANGO)
      LOGICAL FOXTROT ( : ), TANGO ( : )
!HPF$ INHERIT TANGO
```

No mapping directive is given for `FOXTROT`. We choose to collapse it[20]. If the interface is not explicit, the translation is

```
  procs dance_floor(16) ;
  range i_frug(128 #blk dance_floor.dim(1)) ;
```

[20] Not a very intelligent choice one might think, but one a compiler might make by default.

```
array int frug [i_frug] ;
array int twist [i_frug] ;
_terpsichore(frug [13 : step 3], twist [13 : step 3]) ;

void _terpsichore(array int _foxtrot _ [_],
                  array int tango _ [_]) {
  dimension def0d_1 of def0d ;
  array int foxtrot [_foxtrot.rng(1).size() #* def0d_1] ;
  remap(foxtrot, _foxtrot) ;
  ...
  remap(_foxtrot, foxtrot) ;
}
```

If the interface *is* explicit, a likely translation would be

```
procs dance_floor(16) ;
range i_frug(128 #blk dance_floor.dim(1)) ;
array int frug [i_frug] ;
array int twist [i_frug] ;
{
  dimension def0d_1 of def0d ;
  array int temp1 [13 #* def0d_1] ;
  range i_temp2(13 : step 3 : i_frug) ;
  remap(temp1, frug [i_temp2]) ;
  terpsichore(temp1, twist [i_temp2]) ;
  remap(frug [i_temp2], temp1) ;
}

void terpsichore(array int foxtrot [_ #* _],
                 array int tango _ [_]) {
  ...
}
```

An alternative in the present case is to translate `foxtrot` as a normal C array (a "simple variable", so far as Adhoc is concerned):

```
procs dance_floor(16) ;
range i_frug(128 # dance_floor.dim(1)) ;
array int frug [i_frug] ;
array int twist [i_frug] ;
{
  int temp1 [13] ;
  range i_temp2(13 : step 3 : i_frug) ;
  broadcast(temp1, frug [i_temp2]) ;
  terpsichore(temp1, twist [i_temp2]) ;
```

```
    where(i_temp2)
      frug = temp1 [i_temp2] ;
  }

  void terpsichore(int foxtrot [], array int tango _ [_]) {
    ...
  }
```

**Example 6′ (p154)**

```
      SUBROUTINE TERPSICHORE (FOXTROT, TANGO)
      LOGICAL FOXTROT ( : ), TANGO ( : )
!HPF$ INHERIT TANGO
!HPF$ PROCESSORS DANCE_FLOOR (16)
!HPF$ TEMPLATE, DISTRIBUTE (BLOCK) ONTO DANCE_FLOOR :: GURF (128)
!HPF$ ALIGN FOXTROT (J) WITH * GURF (3 * J + 2)
```

Most of the same information concerning the distribution of `FOXTROT` can be expressed in Adhoc through

```
  void terpsichore(
      array int foxtrot [_ : base 2 step 3 : 128 #blk _],
      array int tango _ [_]) {
    ...
  }
```

The only static information which the Adhoc translation fails to capture is the shape, `(16)`, of the processor arrangement (or process array).

**Example 7 (p155)**

```
      REAL TROGGS (1024, 473)
!HPF$ DISTRIBUTE TROGGS (BLOCK, *)
      DO J = 1, 473
        CALL WILD_THING (TROGGS ( : , J))
      END DO
      ...

      SUBROUTINE WILD_THING (GROOVY)
      REAL GROOVY ( : )
!HPF$ DISTRIBUTE GROOVY * (BLOCK) ONTO *
      ...
```

The translation is

```
  dimension def1d_2 of def1d ;
  array float troggs [1024 #blk def1d_1, 473 #* def1d_2] ;
  for (j = 0; j < 473 ; j++)
    wild_thing(troggs [ : , j]) ;
  ...

  void wild_thing(array float groovy [_ #blk def1d_2])
  ...
```

This translation relies on the fact that subscripting the second dimension of `troggs` yields a null embedding.

**Example 8 (p156)**

```
      SUBROUTINE NICHOLAS (TSAR, CZAR)
      REAL, DIMENSION (1918) :: TSAR, CZAR
!HPF$ INHERIT :: TSAR
!HPF$ ALIGN WITH TSAR :: CZAR
      ...
```

Translation...

```
  void nicholas(array float tsar _ [1918 :: _],
                array float czar tsar.emb() [tsar.rng(1)]) {
    ...
```

**Example 9 (p156)**

```
      SUBROUTINE GRUNGE (PLUNGE, SPONGE)
      REAL, DIMENSION (1000) :: PLUNGE SPONGE
!HPF$ ALIGN PLUNGE WITH * SPONGE
      ...

      REAL FIJI (5000), SQUEEGEE (2000)
!HPF$ ALIGN SQUEEGEE (K) WITH FIJI (2 * K)
      CALL GRUNGE (FIJI (2002 : 4000 : 2), SQUEEGEE (1001 : ))
      ...
```

To make things half-interesting, assume that `SPONGE` has the `INHERIT` attribute (otherwise a legitimate translation would be to turn `SPONGE` into a "collapsed" array, say).

```
  void grunge(array float plunge _ [1000 :: _],
              array float sponge plunge.emb() [plunge.rng(1)]) {
    ...
```

```
  array float
     squeegee fiji.emb() [2000 : base 1 step 2 : fiji.rng(1)] ;
  grunge(fiji [1000 : base 2001 step 2],
                             squeegee [1000 : base 1000]) ;
  ...
```

**Example 10**

(This is the only example in this section not taken from HPFH)

```
      SUBROUTINE FOO (A, B)
      REAL A ( : , : )
!HPF$ INHERIT A
      INTEGER B (SIZE (A, 1))
!HPF$ ALIGN B WITH A ( : , 1)
      ...
      END
```

The interesting feature is embedded alignment to an argument with inheritted mapping. The translation is

```
  void foo(array float a _ [_, _],
           array int b (a.emb() \/ (0 : a.rng(2))) [a.rng(1)]) {
    ...
  }
```

## 3.4 Executable Statements

Most of the examples so far concentrated on translation of declarations. The "mapping problem" arose occasionally when the HPF examples did not specify mappings for particular program variables; but, regarding that as an oversight on the part of the programmer, we justifiably took the trivial default, "collapsed" mapping for such arrays.

Executable statements typically involve compiler-generated temporaries, for which the programmer *cannot* define a mapping. In this case the compilation system must be responsible for defining mappings. As admitted earlier, we will not try to define any systematic way of choosing temporary mappings in this document. We will just make some intuitively sensible choice, and display the Adhoc code which would result.

**Example 1**

```
      SUBROUTINE COPY (U, V)
      INTEGER U ( : )
!HPF$ INHERIT U
      INTEGER V (SIZE (U))
```

```
!HPF$ ALIGN V WITH U
      ...
      END

      FUNCTION FOO (N)
      INTEGER N
      INTEGER FOO (N)
!HPF$ DISTRIBUTE F (CYCLIC)
      ...
      END

      INTEGER A (10)
!HPF$ DISTRIBUTE A (BLOCK)
      ...
      CALL COPY (A, FOO (10))
```

In this case a single remapping operation is required...

```
  void copy(array int u _ [_], array int v u.emb() [u.rng(1)]) {
    ...
  }

  void foo(int n, array int foo_res [n #cyc def1d_1]) {
    ...
  }

  array int a [10 #blk def1d_1] ;
  ...
  {
    array int temp1 [10 #cyc def1d_1] ;
    foo(10, temp1) ;
    array int temp2 [10 #blk def1d_1] ;
    remap(temp2, temp1) ;
    copy(a, temp2) ;
  }
```

The given translation assumes that the interfaces of the procedures was explicit in the calling program (note that Fortran *requires* the interface of an array-valued function like `foo` to be explicit).

Adhoc doesn't have `array`-valued functions, so `foo` is rendered as a two-parameter procedure, returning its result in the second parameter. The calling program defines a temporary to hold the result, and a second temporary to hold the remapped value of this result. The interface of `foo` defines the size of the result, so both temporaries can be automatic arrays.

In this example the mapping of the temporaries was rather obvious—essentially uniquely defined by the procedure interfaces.

**Example 2**

```
      INTEGER FUNCTION SUM (U)
      INTEGER U ( : )
!HPF$ INHERIT U
      ...
      END

      INTEGER A (10, 10)
!HPF$ DISTRIBUTE A (BLOCK, BLOCK)
      INTEGER V (5)
!HPF$ DISTRIBUTE V (BLOCK)
      INTEGER X
      ...
      X = SUM (A (3, 2 : F (X) : 2) + V (2 : G (X)))
```

Here `F` and `G` are some user-defined functions (which must return related values, if the two array sections are to conform).

In this case there is no obviously preferred mapping for array temporaries. We will assume that the argument of `SUM` is aligned with the first array section (although alignment with the second array section might be more intelligent).

```
int sum(array int u _ [_]) {
  ...
}

array int a [10 #blk def2d_1, 10 #blk def2d_2] ;
array int v [5 #blk def1d_1] ;
int x ;
...
{
  embed e_t1(4 : a.rng(1)) ;
  range i_t1(f(x) - 1 : base 1 step 2 : a.rng(2)) ;
  array int temp1 {e_t1} [i_t1] ;
  remap(temp1, v [g(x) - 1 : base 1]) ;
  at({e_t1})
    where(i_t1)
      temp1 += a ;
  x = sum(temp1) ;
}
```

We sensibly recycled `temp1`—it was used both to hold the remapped data from `v` and the sums of elements of `a` and `b`[21].

**Example 3**

```
      INTEGER A (100, 20)
!HPF$ DISTRIBUTE A (BLOCK, BLOCK) ONTO P
      INTEGER B (10, 10)
!HPF$ DISTRIBUTE B (BLOCK, BLOCK) ONTO Q
      ...
      FORALL (X = 1 : 10, Y = 1 : 10, Z = 1 : 20)
        A (10 * (X - 1) + Y, Z) = B (X, Y) + F (Y, Z)
      END FORALL
```

Here `F` is some `PURE` function.

First we give a naive translation. Assume that throughout the statement the forall index `X` is aligned with the first range of `B`, `Y` is aligned to the second dimension, and `Z` is a collapsed range. A straightforward translation is

```
  array int a [100 #blk p_1, 10 #blk p_2] ;
  array int b [10 #blk q_1, 10 #blk q_2] ;
  ...
  {
    range x_t1(b.rng(1)) ;
    range y_t1(b.rng(2)) ;
    dimension q_3 of q ;
    range z_t1 (20 #* q_3) ;
    array int temp1 [x_t1, y_t1, z_t1] ;

    // temp1 [x, y, z] = b [x, y] + f(y + 1, z + 1)
    where(x_t1)
      where(y_t1)
        where(z_t1)
          temp1 = b + f(y_t1 + 1, z_t1 + 1) ;

    array int temp2 [x_t1, y_t1] ;

    // temp2 [x, y] = 10 * x + y
    where(x_t1)
      where(y_t1)
        temp2 = 10 * x_t1 + y_t1
```

[21] Whether a compiler would be smart enough to do that is another question. It might allocate two temporary arrays.

```
    // a [10 * x + y, z] = temp1 [x, y, z]
    scatter(temp1, a, temp2, z_t1) ;
  }
```

The translation is divided into calculation of the RHS of the assignment, storing the result in `temp1`, followed by scattering the values into the array `a` on the LHS.

We will discuss a few optimisations which could be made to the calculation of `temp1`. The worst feature of the naive translation is that it redundantly calculates the expression `f(y + 1, z + 1)` for every value of `x`. That can easily be cured by rearranging the loops calculating of `temp1` as follows

```
  where(y_t1)
    where(z_t1) {
      int temp2 = f(y_t1 + 1, z_t1 + 1) ;
      where(x_t1)
        temp1 = b + temp2 ;
    }
```

We should also worry about the sequentialisation implicit in performing the `f` calculation over the collapsed range `z_t1`. There is a "spare" process dimension `q_1` available[22]. Unless evaluation of `f` is known to be very cheap, it would probably be better to distribute its calculation over this dimension, and remap the results afterwards. That leads to the more complex code

```
  {
    array int temp2 [y_t1, z_t1] ;

    // temp2 [y, z] = f(y + 1, z + 1)
    where(y_t1) {
      range z_t3(20 #blk q_1) ;
      array int temp3 [z_t3] ;

      // temp3 [z] = f (y + 1, z + 1)
      where(z_t3)
        temp3 = f(y_t1 + 1, z_t3 + 1) ;

      remap(temp2, temp3) ;
    }

    // temp1 [x, y, z] = b [x, y] + temp2 [y, z]
    where(y_t1)
```

[22] As the translation stands, although the function call is no longer calculated redundantly for *every* value of `x`, it is still calculated redundantly on every process in the associated `q_1` dimension, because the expression is implicitly replicated in this dimension.

```
      where(z_t1)
        where(x_t1)
          temp1 = b + temp2 ;
  }
```

It took a certain amount of intelligence to decide that this transformation would be a good idea. In our overall scheme of things, this intelligence is encapsulated in the mapping phase.

**Example 4**

```
      PURE FUNCTION FOO (N)
      INTEGER N
      INTEGER FOO (F (N))
      ...
      END

      INTEGER A (10, 10)
!HPF$ DISTRIBUTE A (BLOCK, BLOCK) ONTO P
      INTEGER B (10)
!HPF$ DISTRIBUTE B (BLOCK) ONTO Q
      ...
      FORALL (X = 1 : 10)
        A (X, 1 : F (X)) = B (FOO (X))
      END
```

F is supposed to be some complicated function[23]. A possible translation is

```
  void foo(int n, dimension d, alloc int foo_res [_ #* d]) {
    foo_res <- new array int [f(n) #* d] ;
    ...
  }

  array int a [10 #blk p_1, 10 #blk p_2] ;
  array int b [10 #blk q_1] ;
  ...
  {
    range x_t1(b.rng(1)) ;
    dimension q_2 of q ;
    array int temp1 [x_t1] [_ #* q_2] ;

    // temp1 [x] [i] = b [foo(x) [i]]
```

[23] The program isn't strictly legal Fortran 90, because the declaration of FOO includes a call to a user-defined function. Other very complex "specification expressions" are allowed, though; to make a correct program replace the F calls by some such expression.

```
  {
    array int temp2 [x_t1] [_ #* q_2] ;

    // temp2 [x] [i] = foo(x) [i]
    where(x_t1)
      foo(x_t1 + 1, q_2, temp2) ;

    // temp1 [x] [i] = b [temp2 [x] [i]]
    gather(temp1, b, temp2) ;

    where(x_t1)
      delete temp2 ;
  }

  array int temp2 [x_t1] [_ #* q_2] ;

  // temp2 [x] [i] = i
  where(x_t1)
    temp2 <- new range(f(x_t1 + 1) #* q_2) ;

  // a [x, temp2 [x] [i]] = temp1 [x] [i]
  scatter(temp1, a, x_t1, temp2) ;

  where(x_t1)
    delete temp2 ;

  where(x_t1)
    delete temp1 ;
}
```

The temporaries are all distributed over the process array of `b` in this translation (but, again, other choices of mapping would be equally plausible). Again, the translation falls broadly into the evaluation of the RHS in `temp1`, followed by scattering of results. `temp1` and `temp2` (em both `temp2`s) are non-Cartesian arrays. Notice that it was necessary to explicitly pass the process dimension `q_2` into the function `foo`; mapping inquiry functions can't be applied to unallocated arrays! The subscript array `temp2` are allocated as a range—recall that a range can be used to stand for an array with values $0, \ldots$, which is what is needed here. The elements of `temp1` and `temp2` have to be explicitly deallocated using the `delete` operator.

**Example 5**

```
INTEGER A (10, 10)
```

```
!HPF$ DISTRIBUTE A (BLOCK, BLOCK) ONTO P
      INTEGER B (10, 10)
!HPF$ DISTRIBUTE B (BLOCK, BLOCK) ONTO Q
      ...
      FORALL (X = 1 : 10)
        FORALL (Y = 1 : X)
          A (X, Y) = B (X, Y) + F (X) + G (Y)
        END FORALL
      END FORALL
```

First a translation which makes use of non-Cartesian arrays. The temporaries are mapped to the to process arra of `B` (namely `Q`). The `Y` range is collapsed, otherwise the temporaries would not have a legal HPF mapping.

```
  array int a [10 #blk p_1, 10 #blk p_2] ;
  array int b [10 #blk q_1, 10 #blk q_2] ;
  ...
  {
    range x_t1(b.rng(1)) ;
    dimension q_3 of q ;
    array int temp1 [x_t1] [_ #* q_3] ;
    array int temp2 [x_t1] [_ #* q_3] ;

    // temp1 [x] [y] = b [x, y] + f(x) + g(y)
    // temp2 [x] [y] = y
    {
      where(x_t1) {
        range y_t1(x_t1 + 1 #* q_3) ;

        temp1 <- new array int [y_t1] ;

        array int temp2 [yt_1] ;

        // temp2 [y] = b [x, y]
        remap(temp2, b [y_t1]) ;

        // temp3 = f(x + 1)
        int temp3 = f(x_t1 + 1) ;

        where(y_t1)
          temp1 = b + temp3 + g(y_t1 + 1) ;

        temp2 <- new range(y_t1) ;
      }
    }
```

```
    // a [x] [y] = temp1 [x] [y]
    scatter(temp1, a, x_t1, temp2) ;

    where(x_t1) {
      delete temp2 ;
      delete temp1 ;
    }
  }
```

Several inefficient aspects of this translation (sequentialisation implied by collapsing the `y` range; redundant calls to `g`; the use of non-Cartesian arrays; the use of the `scatter` primitive) can be avoided by adopting a "masked Cartesian" mapping.

```
  array int a [10 #blk p_1, 10 #blk p_2] ;
  array int b [10 #blk q_1, 10 #blk q_2] ;
  ...
  {
    range x_t1(b.rng(1)) ;
    range y_t1(b.rng(2)) ;
    array int temp1 [x_t1, y_t1] ;

    // temp1 [x] [y] = b [x, y] + f(x) + g(y)
    {
      array int temp2 [y_t1] ;

      // temp2 [y] = g(y + 1)
      where(y_t1)
        temp2 = g(y_t1 + 1) ;

      where(x_t1) {

        // temp3 = f(x + 1)
        int temp3 = f(x_t1 + 1) ;

        // temp1 [x, y] = b [x, y] + temp3 + temp2 [y] ;
        where(y_t1)
          if(y_t1 <= x_t1)
            temp1 = b + temp3 + temp2 ;
      }
    }

    range x_t2(a.rng(1)) ;
    range y_t2(a.rng(2)) ;
```

```
    array int temp2 [x_t2, y_t2] ;

    // temp2 [x, y] = temp1 [x, y]
    remap(temp2, temp1) ;

    // a [x, y] = temp2 [x, y]
    where(x_t2)
      where(y_t2)
        if(y_t2 <= x_t2)
          a = temp2 ;
  }
```

**Example 6**

```
      SUBROUTINE FOO (A)
      INTEGER A (10, 10)
!HPF$ INHERIT A
      INTEGER B (10)
!HPF$ ALIGN B WITH A ( * , : )
        ...
        FORALL (X = 1 : 10, Y = 1 : 10)
          A (X, Y) = F (X) + B (Y)
        ENDFORALL
      END
```

This example illustrates the translation of code in a procedure, when the arguments have inherited mapping. A translation is

```
  void foo (array int a _ [10 :: _, 10 :: _]) {
    array int b a.emb() [a.rng(2)] ;
    ...
    at(a.emb()) {
      range x(a.rng(1)) ;
      where(x) {
        int temp1 = f(x + 1) ;
        where(a.rng(1))
          a = b + temp1 ;
    }
  }
```

(Incidentally, notice how the `rng(1)` inquiry function was used inside the loop, to save introducing a new name, `y` say, outside the loop. It is worth emphasising two peculiarities that arise because the `x` range of `a` has vanished within the outer loop. First, it would be impossible to avoid naming the outer range, because we need the index value of `x` inside the loop—we can't apply apply an inquiry function to `a` to get this value because the range isn't in the array any more! Second, the `y` range is the *first*, not the second range of `a` at this point—hence `rng(1)` rather than `rng(2)`. To avoid likely confusions, it is probably better practise to explicitly name all affected ranges before entering any `where` loops.)

# 4 Translation of Adhoc to C++

By design *Adhoc* has a very straightforward transcription to C++. This transcription depends on a library containing:

- Classes for process arrays and process dimensions
- Classes for ranges
- Template classes for arrays
- Functions to implement `remap`, `broadcast`, `gather` and `scatter`, `input` and `output` (and some reduction operations).

*Mini-Adlib*[24] is a minimal run-time library for Adhoc. It is minimal in the sense of providing the smallest reasonable set of operations for translation of the language. The operations provided are deliberately "high-level"—*efficient* translation would depend on an extended RTL, providing a large set of simpler, lower-level operations which the translator can place "in line" in the generated code.

## 4.1 Process arrays and dimensions

The public part of the Mini-Adlib library for handling process arrays and process dimensions is displayed below.

```
class Dimension ;

class Procs {
  // ...
public :
  Procs() ;
  Procs(int, ...) ;  // Procs(int r, int n_1, ..., int n_r)
  Dimension* dim(int) ;
} ;

class Dimension {
  // ...
```

[24] The name "ADLIB" predates "Adhoc". It is the name of the (never-fully-implemented) communication library supporting the Southampton ADAPT system [].

```
  public :
    Dimension(Procs&) ;
    ~Dimension() ;
  } ;
```

A process array is represented by an object of class `Procs`. There are two constructors. The first constructor creates an "empty" process array, in the style of Vanilla Adhoc `procs` declaration. The second, more useful, constructor sets up a process array of rank `r`, and the shape specified by the remaining arguments[25].

The Adhoc process dimension is represented by the `Dimension` class. The `Procs(r, ...)` constructor internally creates `r` objects from this class. The `d`th dimension can be accessed using the `dim(d)` inquiry function. The constructor `Dimension(p)` attaches a new dimension to the pre-existing process array `p`. In practise a process dimension created dynamically in this way will be "collapsed" or "on-processor"—the only truly distributed dimensions are those created by the `Procs(r, ...)` constructor.

## 4.2 Ranges and embeddings

`Subscript` is a base class representing any object which can be used as an array subscript—namely an embedding or a range (in Adhoc simple variables and triplets can also be subscripts; Mini-Adlib assumes that these are promoted by applying them to the associated range of the array being subscripted).

```
  class Subscript {
    // ...
  } ;
```

`Subscript` has no public members.

An Adhoc range is represented by a `Range` object

```
  class Format ;

  class Range : public Subscript {
    // ...
  public :
    Range(int, Dimension&) ;
    Range(int, Dimension&, Format) ;
    Range(int, Dimension&, Format, int) ;
    Range(int, int, int, Range&) ;
```

[25] Note that this constructor will require access to information about the physical process group on which the SPMD program is executing. It has to decide how to map virtual processes of the array to this group. For the moment we assume that this information is available in some global variable.

```
    int size() ;

    operator void*() ;
    void first() ;
    void operator++() ;
    void last() ;
    void operator--() ;

    int operator()(Subscript&) ;
    int operator()() ;
  } ;
```

For all `Range` constructors the first argument is the extent of the range. The first three constructors build primitive ranges with assorted distribution formats. `Format` is some type which can take the value "block" or "cyclic". The fourth constructor builds a derived range—the second and third arguments are base and stride in the parent range.

`size` is an inquiry function, which returns the extent of a range.

The next five members are iterator functions. One of the röles of the `Range` class is as an iterator class for implementing `where` loops. The `first` function activates a range as an iterator, the `void*` operator is a test function, and the `++` operator is a "next" function. These operations are only ever used in a `for` loop. If `x` is a range then

```
for(x.first() ; x ; x++)
```

is a translation of

```
where(x)
```

Clearly the state of a `Range` structure changes while it is being used as an iterator. In particular, between a call to `first` and termination of the associated loop, an internal flag is set which alters the behaviour `Array` member functions when the range is passed as a `Subscript` argument. We return to this point later.

An alternative translation of `where(x)` is

```
for(x.last() ; x ; x--)
```

The only difference is that the single process sequential parts of the loop are performed in reverse order. This shouldn't alter the behaviour of a program, but by performing memory deallocation in inverted loops one can sometimes eliminate heap allocation of array elements in favour of stack allocation.

The two `()` operations are used to translate situations where an index is standing for a one-dimensional array; their effect will be explained later.

The interface of the `Embed` class, used to implement Adhoc embeddings, is simpler

```
class Embed : public Subscript {
  // ...
public :
  Embed(int, Range&) ;
  Embed(int, Range&, Embed*) ;

  Embed* next() ;

  operator void*() ;
} ;
```

The first constructor obviously creates an embedding at the specified location in the specified range. The second constructor does the same, but also initialises a link field to another embedding; these link fields are used to construct the (intrusive) linked lists which implement embedding sets in Mini-Adlib. The value of the link is accessed by `next()`.

The `void*` operator is a vestige of the `void*` iterator function in the `Range` class. It returns a non-zero value only on a processor which contains the embedding. It can be used to translate `at`. So

```
embed e(...) ;
...
at({e})
  ...
```

is translated by

```
Embed e(...) ;
...
if(e)
  ...
```

The general construct

```
at(es)
  S
```

can be translated by

```
Embed* tail = es ;
while(*tail)
  if(! (tail = tail.next())) {
    S' ;
    break ;
  }
```

assuming `es` is a pointer to an `Embed` structure at the head of a linked list representing the embedding set, and `S'` is the translation of `S`.

### 4.3 Arrays

The basic Adhoc array is represented by the `Array` class

```
class Frame {
  // ...
public :
} ;

template<int R, class T>
class Array : public Frame {
  // ...
public :
  Array(Embed*, ...) ;  // Array(es, x_1, ..., x_R)
  Array(Range&, ...) ;  // Array(x_1, ..., x_R)
  ~Array() ;

  Range* rng(int) ;
  Embed* emb() ;

  T& operator()(Subscript&, ...) ;  // this(i_1, ..., i_R)
  T& operator()() ;                 // ...used if R=0
} ;
```

The base class `Frame` is used to type certain function parameters in the library, where functions must accept arrays of any rank and type. Generally "users" will only need the derived template class `Array` (and sub-classes below) which statically specify rank and element type through the template arguments.

The first constructor takes an embedding set and a list of ranges. The second creates an array with an empty embedding set (the most common case).

The inquiry function `rng(d)` returns the `d`th range of the array; `emb()` returns its embedding set.

Finally the `()` operator implements subscripting of the array. It's arguments are of class `Subscript`—ie `Range` or `Embed`, and the result is a reference to a single array element. This operation *cannot* be used to construct array sections—they are constructed with the `Section` constructor, introduced below.

The effect of a subscript of class `Embed` should be fairly obvious

```
Array<1, int> a(x)
...
Embed e(2, x) ;
if(e)                 // ie, if 'e' is on this processor
  a(e) = 23 ;
```

is the translation of

```
array int a [x]
...
embed e(2 : x) ;
at(e)
  a = 23 ;
```

It should *not* be obvious why a `Range` can be used as a subscript in this operation, given that the operation cannot return a section! In fact a `Range` can only be used *while it is activated as an iterator*. During this time a flag is set in a `Range` object which causes subscripting operations to treat the range *as if* it were an embedding at the point associated with the current iteration of the loop. So

```
Array<1, int> a(x)
...
for(x.first() ; x ; x++)
  a(x) = 23 ;
```

is the translation of

```
array int a [x]
...
where(x)
  a = 23 ;
```

As emphasised, the subscripting operation in the `Array` class cannot be used to construct array sections; in fact this operation is normally used to implement Adhoc `at` and `where` constructs rather than Adhoc subscripting operations. The latter are translated using the `Section` subclass.

```
template<int R, class T>
class Section : public Array<R, T> {
  // ...
public :
  Section(Embed*, Frame, ...) ;  // Section(es, a, i_1, ..., i_s)
  Section(Frame, ...) ;          // Section(a, i_1, ..., i_s)
  ~Section() ;
} ;
```

The `Frame` argument of the constructors is the array to be subscripted; it has rank `s`, say. Again the subscripts are of type `Subscript`—either `Embed` or `Range`. Again, if the a `Range` subscript is activated as an iterator it will behave like an embedding; but if a `Range` object which is not currently activated as an iterator is used as a subscript, it will act like a triplet, and add a range to the `Section` constructed. So there are `R` such range subscripts.

The first constructor adds a non-trivial embedding set to the `Section` array created, the second constructor leaves the embedding set empty. It is left up to the "user" to get this embedding set right. In principle the constructor itself could work out an embedding set from the embedding of the original array and the subscripting pattern, but in practise the caller can generally do it more efficiently (recall also the rather subtle rules about array embedding sets in Adhoc, which depend on the `at` context).

Non-Cartesian arrays are translated by the `Aggregate` class.

```
template<int R>
class Aggregate : public Array<R, void*> {
  // ...
public :
} ;
```

So far as its public interface is concerned an `Aggregate` is just an array of untyped pointers; the private part of the implementation will include extra members, specific to non-Cartesian arrays. These are used internally by other functions in the library.

Finally, to support the treatment of ranges as one dimensional arrays of index values we introduce the `Index` subclass.

```
class Index : public Array<1, int> {
  // ...
public :
  Index(Range&) ;
} ;
```

Now

```
Index(x)
```

can be used as a constant array containing the index values in `x`.

We can now explain the two members

```
int operator()(Subscript&) ;
int operator()() ;
```

in the range class. These are subscripting operations which optimise common usage The single-argument `()` operation

```
x(i)
```

is defined to return the same value as

```
Index(x)(i)
```

The no-argument `()` operation

```
x()
```

is defined in turn to return the same value as

```
x(x)
```

### 4.4 Communication primitives

The semantics of the functions

```
void remap(Frame&, Frame&) ;
void gather(Frame&, Frame&, ...) ;
void scatter(Frame&, Frame&, ...) ;
void input(Frame&) ;
void output(Frame&) ;
```

are defined by the semantics of `Adhoc`—they should not require further explanation.

### 4.5 Examples of translation

The appendix gives translations of many of the Adhoc examples in earlier sections to C++ and Adlib.

## References


[1] D.B. Carpenter. *Geometric Parallelism.* Esprit Project PUMA workpackage 5.1, interim report.