An Outline Proposal for the PCRC Library Interface

D.B. Carpenter

Northeast Parallel Architectures Centre,
Syracuse University,
111 College Place,
Syracuse, New York 13244-410

DRAFT

Contents

1 Introduction **2**

2 Local housekeeping **2**
2.1 Process coordinates. 2
2.2 Address translation. 3
2.3 Parallel iteration 4
2.4 Miscellaneous constructors 7

3 Collective operations **7**
3.1 Regular data remapping 8
3.2 Gather and scatter operations 10
3.3 Collective arithmetic operations 11

A Glossary of Function Interfaces **12**

B Remarks on the DAD **14**
B.1 Some shortcomings of the current definitiion 14
B.2 An improved definition 14
B.3 Clarification of problems with the DAD 15
B.3.1 Scalar subscripts vs extent-1 triplet subscripts 15
B.3.2 Lower bounds in the run-time representation 16
B.4 The data model 17

1 Introduction

This document sketches a possible interface to the PCRC run-time library. The text includes various concrete procedure interfaces in C. This "concrete syntax" will definitely change later. For now the important thing is to define the underlying semantics of the library[1].

The library contains two types of routine.

- "Housekeeping" routines: local procedures used for operations such as translation between global indices and local indices, enumeration of local subranges, and initialising DAD records.
- Collective operations: collective communications, and collective arithmetic operations.

Section 2 outlines a possible set of routines in the first category. Section 3 gives examples of routines in the second category. Eventually this category will include support for all the array intrinsic functions of Fortran 90 and HPF.

The text of this paper includes possible translations of various HPF fragments to C. A preliminary glossary of C function interfaces is collected in the appendix. Once the principles of the library are agreed, Fortran interfaces can also be defined.

The appendix also discusses some problems we have encounterd with the *Distributed Array Descriptor* (DAD) defined in [1]. Some improvements are suggested.

2 Local housekeeping

This section proposes "top-level" routines to perform various housekeeping operations. It is presumed that an optimising compiler will often place the implementations of lightweight operations (such as these) in-line in the generated code.

2.1 Process coordinates.

HPF has an idea of processor arrangements, which are multi-dimensional arrays of logical processes.

The most important run-time operations associated with processor arrangements is to determine the position (the *process coordinates*) of *this* processor within the process array.

[The original version of this specification called for the introduction of a run-time representation of the processor arrangement. This proposal has been

[1] So, for example, this text does not to adopt a systematic naming convention for the routines, such as pre-pending `PCRC` to the name of every procedure, and it sometimes slips into C++-style overloading of names.

omitted, following private discussions with Xiaoming Li, who points out that MPI already provides similar functionality.]

2.2 Address translation.

A *global subscript* is an HPF-program-level index into a distributed array. A *local subscript* is an index into the locally held segment of the distributed array.

Following the proposal in [1], the local segments of an aligned array are allocated *as if* the array has the shape of its template. If the array is aligned to just a section of its template, elements of the allocated array outside the alignment section remain unused.

A *template subscript* is a global index into a template.

Address translation has two components. There is a mapping from global subscript to template subscript—this part of the mapping depends on on the alignment parameters of the array. Then there is a mapping between template subscripts and pairs of the form

```
(coord, l_i)
```

where `l_i` is a local index and `coord` is a process coordinate—this part of the mapping depends only on the size and distribution format of the template.

The primitive routines for converting between global an local indices are

```
void globToLoc(int* coord, int* l_i, int g_i,
           int g_lb, int g_ub, int offset, int stride,
           int t_lb, int t_ub, int distCode, int numProcs) ;

void locToGlob(int* g_i, int coord, int l_i,
           int g_lb, int g_ub, int offset, int stride,
           int t_lb, int t_ub, int distCode, int numProcs) ;
```

The procedure `globToLoc` takes a global subscript `g_i` and returns a coordinate and local subscript pair `*coord` and `*l_i`. The procedure `locToGlob` performs the reverse translation. The remaining parameters are fields in the DAD describing the mapping of an individual array dimension.

More convenient interfaces are

```
void globToLoc(int* coord, int* l_i, int g_i, DIM* range) ;

void locToGlob(int* g_i, int coord, int l_i, DIM* range) ;
```

which work in terms of the `DIM` structure in the DAD. This `DIM` structure can be extracted directly from the DAD. For example, if `a_DAD` is the DAD of some array, a pointer to the `DIM` structure for the `d`th dimension is given by

```
a_DAD.axes + d ;
```

The HPF fragment

```
a(i) = 27
```

could be translated (to C) as

```
globToLoc(&coord, &l_i, i, a_DAD.axes) ;

if(coord == locCoord)
  a_LOC [l_i - 1] = 27 ;
```

assuming `locCoord` is the coordinate of the local processor in the relevant process dimension, and `a_LOC` is the C array holding the local segment of the HPF array.

2.3 Parallel iteration

The function

```
void locBlock(int* l_lb, int* l_ub, int* l_stride,
              int g_lb, int g_ub, int offset, int stride,
              int t_lb, int t_ub, int distCode, int numProcs,
              int coord) ;
```

computes the parameters `l_lb`, `l_ub` and `l_stride` of the segment of an array held on a process with coordinate `coord`. In the simplest case, where the alignment is trivial, and the template extent divides the process extent exactly, the values of the three returned parameters are just

```
*l_lb     = 1
*l_ub     = t_blockSize
*l_stride = 1
```

`t_blockSize` being the extent of a local segment. In general the local lower bound, upper bound and stride have non-trivial value. The returned value `*l_stride` is always positive, even if `stride` is negative. A couple of examples are given in figures 1 and 2.

A more convenient interface is

```
void locBlock(int* l_lb, int* l_ub, int* l_stride,
              DIM* range, int coord) ;
```

The procedure `locBlock` is used to initialise the `l_lb` and `l_ub` fields in the DAD, or it could be used directly to translate parallel loops. For example

```
FORALL (i = l : u : s)
  a(i) = ...
END FORALL
```

l_i					
7	-	11	-	-	
6	-	-	13	-	
5	8	-	-	15	
4	-	10	-	-	-
3	-	-	12	-	-
2	-	-	-	14	-
1	-	9	-	-	-
coord	0	1	2	3	4

l_lb	5	1	3	2	1
l_ub	5	7	6	5	-2
l_stride	3	3	3	3	3

Figure 1: Output of `locBlock` for `distCode` = 1, `numProcs` = 5, `t_lb` = 1, `t_ub` = 32, `g_lb` = 8, `g_ub` = 15, `offset` = 4, `stride` = 3

l_i						
7	2	-	-	-	1	-
6	-	-	3	-	-	-
5	5	-	-	-	4	-
4	-	-	6	-	-	-
3	8	-	-	-	7	-
2	-	-	9	-	-	-
1	-	-	-	-	10	-
coord	0	1	2	3	4	5

l_lb	3	1	2	1	1	1
l_ub	7	0	6	0	7	0
l_stride	2	1	2	1	2	1

Figure 2: Output of `locBlock` for `distCode` = 2, `numProcs` = 6, `t_lb` = 1, `t_ub` = 42, `g_lb` = 1, `g_ub` = 10, `offset` = 40, `stride` = -4

could be translated by

```
DIM* j = a_DAD.axes ;
locBlock(&l_lb, &l_ub, &l_stride,
    l, u, offset + (l - j->g_lb) * j->stride, s * j->stride,
    j->t_lb, j->t_ub, j->distCode, j->numProcs,
    locCoord) ;
for(l_i = l_lb ; l_i <= l_ub ; l_i += l_stride)
  a_LOC [l_i - 1] = ...
```

If the RHS of the forall assignment involves the value `i` in some functional context (eg, `SIN(q * i)`) the value of the global index can be recovered from `l_i` by using the `locToGlob` address translation function introduced in section 2.2 (see section 3.2 for an example).

The more compact second form of the `locBlock` interface can be used in this translation if we introduce an additional function

```
void rangeSection(DIM* range, DIM* parent,
                  int l, int u, int s, int coord) ;
```

This initialises a new `DIM` structure aligned with the original one through the triplet `l : u : s`. Then the opaque fragment

```
DIM* j = a_DAD.axes ;
locBlock(&l_lb, &l_ub, &l_stride,
    l, u, offset + (l - j->g_lb) * j->stride, s * j->stride,
    j->t_lb, j->t_ub, j->distCode, j->numProcs,
    locCoord) ;
```

can be replaced by the more concise

```
DIM i ;
rangeSection(&i, a_DAD.axes, l, u, s, locCoord) ;
locBlock(&l_lb, &l_ub, &l_stride, &i, locCoord) ;
```

The `rangeSection` function can also be used to initialise the `DIM` fields in a DAD describing a Fortran 90 regular section of some parent array.

[*Aside*: the `coord` parameter to `rangeSection` is needed so that the fields `l_lb`, `l_ub` *within* that structure can be initialised. If the `DIM` structure was extended to include a `locCoord` field, this parameter could be omitted, because its value could be taken from the parent structure.]

[*Aside*: if, additionally the `DIM` structure was extended to include the `l_stride` field, the whole translation of the original HPF fragment could be simplified to

```
DIM i ;
rangeSection(&i, a_DAD.axes, l, u, s) ;
for(l_i = i.l_lb ; l_i <= i.l_ub ; l_i += i.l_stride)
  a_LOC [l_i - 1] = ...
```

because we would expect the `rangeSection` operation to initialise the fields `l_lb`, `l_ub` and `l_stride` of `i`, by calling `locBlock` internally.]

2.4 Miscellaneous constructors

In this context a *constructor* means a function which initialises the fields of a data structure. The relevant data structures here are the `DAD` structure itself and the `DIM` structure.

One example of a `DIM` constructor, is the function `rangeSection`. Another useful example is

```
void range(DIM* range,
           int g_lb, int g_ub, int offset, int stride,
           int t_lb, int t_ub, int distCode,
           int numProcs, int locCoord) ;
```

which initialises all fields of a `DIM` structure (it will probably call `locBlock` internally to initialise the `l_lb`, `l_lu` fields).

DAD constructors will also be useful. For example a *section* constructor could have the interface

```
void arraySection(DAD* array, DAD* parent,
                  int l1, int u1, int s1,
                  int l2, int u2, int s2,
                  ...)
```

If `*parent` describes an array `a`, the result `*array` describes the section

```
a (l1 : u1 : s1, l2 : u2 : s2, ...)
```

This interface needs some further elaboration; a mechanism is needed to allow simple subscripts (as well as subscript triplets) to be represented.

The `DAD` constructor `arraySection` will probably call the `DIM` constructor `rangeSection` internally.

3 Collective operations

Collective operations can be divided into three broad categories

- Operations which take data held in regular sections of one distributed array, and copy the values into regular sections of another distributed or sequential array. Familar examples are shifts, transposes and broadcasts.
- Operations which gather or scatter data using more general patterns of subscripting. In Fortran 90, a familar example is an array assignment using a vector subscript; in HPF or Fortran 95, another example is a

forall assignment where the subscript expression are non-linear functions of the forall indices. More esoteric examples include the *combining scatter* functions in the HPF library.

- Collective arithmetic operations on regular data. These include reduction operations operations and matrix arithmetic.

3.1 Regular data remapping

The majority of operations in this class can be reduced to HPF array assignments in which a regular section of one array is assigned to a regular section of another. Broadcasts fall into this category if we regard an array element as a rank-0 section of an array.

As the prototype operation in this category we propose

```
void remap(DAD* dst, DAD* src) ;
```

which simply copies the values in the array described by `*src` to the array described by `dst`.

A construct such as

```
FORALL (i = l : u : s)
  a(i) = b(i) + ...
END FORALL
```

in which `a` and `b` may have unrelated distribution formats, might be translated by

```
arraySection(&bSect_DAD, &b_DAD, l, u, s) ;
arraySection(&tmpSect_DAD, &tmp_DAD, l, u, s) ;

remap(&tmpSect_DAD, &bSect_DAD) ;

DIM* i = tmpSect_DAD.axes ;

for(l_i = i->l_lb ; l_i < i->l_ub ; l_i += i->l_stride)
  a_LOC [l_i - 1] = tmp_LOC [l_i - 1] + ...
```

This is very similar to the example in section 2, but now the selected section of array `b` needs to be remapped to a section of a temporary aligned with `a` before the computation can proceed. We assume that `tmp` has been set up by the compiler with the same mapping parameters as `a`. The section constructors introduced in section 2.4 are called to build DADs for the sections of `b` and `tmp` which will be referenced in the loop. Then the `remap` operation is invoked to perform the communication.

Because the PCRC distributed data model is proposed not to support replicated alignments, the `remap` operation does not directly implement broadcast operations. An array element can be broadcast by

```
void broadcast(void* dst, DAD* src, int i1, int i2, ...) ;
```

Here `i1`, `i2`, ...are the global subscripts for the element. It may or may not be necessary to provide a lower-level operation which broadcasts an arbitrary datum from a specified processor.

Due to its generality, some compromises with efficiency will be required in implementing `remap`. Therefore less general variants which impose restrictions on the alignments of their arguments may be provided. One which has been mooted is

```
void remapShift(DAD* dst, DAD* src, int dim) ;
```

This variant assumes `dst` and `src` have identical alignments except in the dimension selected by `dim`, and in this dimension their alignments only differ in their offset. If the compiler establishes such a relationship between alignments, this streamlined routine can be called.

The `remap` operation is already very general, but there are some common situations which it does not cover. For example it cannot implement the F90 `TRANPOSE` intrinsic, and (again because the PCRC distributed data model is proposed to not support replicated alignments) it cannot implement the `SPREAD` intrinsic. So we propose a slightly more general operation

```
void promote(DAD* dst, DAD* src, int* injection) ;
```

The rank of the array described by `*dst` is greater than or equal to the rank of the array described by `*src`. The length of the `injection` vector the rank of the *source* array; its elements are distinct values in the range $0, \ldots, r-1$, where r is the rank of the *destination* array. Values in the source array are copied to the destination, mapping dimensions of the source to dimensions of the destination according to the injection vector. The values are replicated in any dimension of the destination not selected by an element of the injection vector. For example, if `a` and `b` are both rank 2 arrays, the operation

```
inj [0] = 1 ; inj [1] = 0 ;
promote(&b_DAD, &a_DAD, inj) ;
```

implements a transpose. If `a` is rank 1 and `b` is rank 2, the operation

```
inj [0] = 0 ;
promote(&b_DAD, &a_DAD, inj) ;
```

copies the values in `a` into `b`, replicating them in the second dimension of `a`. It implements a "spread" operation.

The general procedures described above may be supplemented by additional entry points which directly support intrinsic operations such as `CSHIFT` and `EOSHIFT`.

3.2 Gather and scatter operations

To implement parallel assignments with irregular patterns of subscripting we need gather and scatter operations.

Suppose `a` is an `n`-element array and `b` is a rank-2 array. The fragment

```
FORALL (i = 1 : n)
  a(i) = b(f(i), g(i))
END FORALL
```

where `f` and `g` are arbitrary functions, can be translated in two steps. First arrays of subscript values are computed, then a *gather* operation is invoked.

```
DIM* i = a_DAD.axes ;
DAD* subs [2] ;

/* Compute subscript arrays, 'tmp1', 'tmp2' */
for(l_i = i->l_lb ; l_i < i->l_ub ; l_i += i->l_stride) {
  int g_i ;
  locToGlob(&g_i, locCoord, l_i, i) ;

  tmp1_LOC [l_i - 1] = f(g_i) - 1 ;
  tmp2_LOC [l_i - 1] = g(g_i) - 1 ;
}

/* gather data */
subs [0] = &tmp1_DAD ;
subs [1] = &tmp2_DAD ;
gather(a_DAD, b_DAD, subs) ;
```

This assumes the compiler has set up `tmp1` and `tmp2` with the same alignment parameters as `a`.

The general `gather` operation has interface

```
void gather(DAD* gath, DAD* scat, DAD* [] subs) ;
```

Here `gath` describes the *gathered data* and `scat` describes the *scattered data*. These is followed followed by a vector of DADs describing the subscript arrays, which are integer arrays with the same shape and mapping as the *gathered data*. The number of subscript arrays should be equal to the rank of the scattered data, and the values of their elements should lie in the ranges expected for subscripts of that array. Data is copied from `scat` to `gath` with the specified subscripting pattern.

A completely analogous primitive `scatter` copies data from `gath` to `scat`:

```
void scatter(DAD* gath, DAD* scat, DAD* [] subs) ;
```

The F90 `reshape` intrinsic can be implemented by a `gather` operation, but in practise an extra library operation may be provided to implement this function directly.

3.3 Collective arithmetic operations

Reduction operations and matrix arithmetic operations (and other more complex operations in the HPF library) can be implemented directly.

For example the F90 `MATMUL` function can be supported through the procedure

```
void matmul(DAD* result, DAD* a, DAD* b) ;
```

The mappings of the three arguments need not be related.

A Glossary of Function Interfaces

```
int procs(int rank, int* shape) ;

int on(int procsHandle) ;
void crds(int* coords, int procsHandle) ;
```

Figure 3: Process coordinates, see section 2.1.

```
void globToLoc(int* coord, int* l_i, int g_i,
           int g_lb, int g_ub, int offset, int stride,
           int t_lb, int t_ub, int distCode, int numProcs) ;

void locToGlob(int* g_i, int coord, int l_i,
           int g_lb, int g_ub, int offset, int stride,
           int t_lb, int t_ub, int distCode, int numProcs) ;

void globToLoc(int* coord, int* l_i, int g_i, DIM* range) ;

void locToGlob(int* g_i, int coord, int l_i, DIM* range) ;
```

Figure 4: Address translation, see section 2.2.

```
void locBlock(int* l_lb, int* l_ub, int* l_stride,
              int g_lb, int g_ub, int offset, int stride,
              int t_lb, int t_ub, int distCode, int numProcs,
              int coord) ;

void locBlock(int* l_lb, int* l_ub, int* l_stride,
              DIM* range, int coord) ;
```

Figure 5: Parallel iteration, see section 2.3.

```
void range(DIM* range,
           int g_lb, int g_ub, int offset, int stride,
           int t_lb, int t_ub, int distCode,
           int numProcs, int locCoord) ;

void rangeSection(DIM* range, DIM* parent,
                  int l, int u, int s, int coord) ;

void arraySection(DAD* array, DAD* parent,
                  int l1, int u1, int s1,
                  int l2, int u2, int s2,
                  ...)
```

Figure 6: Constructors, see section 2.4.

```
void remap(DAD* dst, DAD* src) ;
void promote(DAD* dst, DAD* src, int* injection) ;
void broadcast(void* dst, DAD* src, int i1, int i2, ...) ;
void remapShift(DAD* dst, DAD* src, int dim) ;

void gather(DAD& gath, DAD& scat, DAD* [] subs) ;
void scatter(DAD& gath, DAD& scat, DAD* [] subs) ;

void matmul(DAD* result, DAD* a, DAD* b) ;
```

Figure 7: Collective operations, see section 3.

B Remarks on the DAD

B.1 Some shortcomings of the current definitiion

There are a few problems with the DAD, as defined in [1].

1. The DAD does not distinguish triplet subscripts of extent one from scalar subscripts. This causes problems with operations that depend on absolute dimension values.

   For example the `DIM` parameter of `CSHIFT` selects a dimension in its array (section) argument, *not* the template. If the array (section) happens to have some dimensions of extent 1, the DAD does not define a unique mapping from array (section) dimension to template dimension.

2. The DAD includes non-default lower bounds on global indices. This causes problems at the procedure boundary, where lower bounds generally change. Lower bounds are not intrinsic properties of array objects in Fortran. They are associated with array variable *names*, not the data themselves. So it is not really appropriate to have these bounds in the data representation.

   It would be preferable to standardize on zero for the base subscript of all arrays. (Of course this standard only applies in the run-time representation. The language compiler is free to emit subscript expressions implementing any lower bound for a given variable.)

3. The DAD does not have a `l_stride` field (see section 2.3). This is not a bug, but it is inconvenient to have `l_lb` and `l_ub` but *not* `l_stride`.

4. The DAD does not have a `locCoord` field. This is not a bug, but it is inconvenient to have `numProcs` but *not* `locCoord`.

B.2 An improved definition

A proposed revision of the DAD, to fix these problems:
The fields in `DIM` are changed to

```
g_extent        Global extent of section dimension.
l_lb            As existing definition, but
                           in range 0, ..., t_blockSize - 1.
l_ub             "     "          "        "
                            "    "       "       "

l_stride        Stride of local segment in local index space.

offset          As existing definition
stride                "         "
```

```
distCode                "        "
numProcs                "        "
locCoord         local process coord
                            in range 0, ..., numProcs - 1

t_extent         Global extent of template dimension
t_blockSize      Size of local block of template
```

If the template dimension is scalar-subscripted, the information in `DIM` should be

```
l_i              Location of section in local index space.
coord            Process coord at which section is held.

distCode         As above
numProcs          "   "
locCoord          "   "

t_extent          "   "
t_blockSize       "   "
```

A flag should be added to distinguish these two cases. To avoid using unions, which may cause problems in Fortran, the second case can be encoded by storing, say, `l_i` in `l_lb`, and `coord` in `l_ub`.

B.3 Clarification of problems with the DAD

[This section has been inserted following private discussions with Xiaoming Li.]

B.3.1 Scalar subscripts vs extent-1 triplet subscripts

Fortran 90 distinguishes scalar subscripts from triplet subscripts of extent one. So if `B` is a rank-1 array, then

```
B (2)
```

is a scalar, whereas

```
B (2 : 2)
```

is another rank-1 array. The data layout of the two are obviously the same (although if the two expressions were passed as arguments to a procedure, the procedure-parameter mechanisms would probably be different; the second expression would typically be accompanied by a dope vector).

As a more complex example, suppose `A` is declared by

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

Consider the two sections

```
e_1  ::=  A (:, 5 : 5, :, 5)

e_2  ::=  A (:, 5, :, 5 : 5)
```

The data layout of these two sections is identical, and the current definition of the DAD does not distinguish between them.

But the `e_1` has shape `(10, 1, 10)` whereas `e_2` has shape `(10, 10, 1)`. Consider the effect of passing these two sections to

```
SUBROUTINE FOO (C)
  INTEGER C (:, :, :)

  C = CSHIFT (C, 1, DIM = 2) ;
END
```

What should happen is that

```
CALL FOO (e_1)
```

does a circular shift in the second dimension of C. This dimension has extent 1, and a circular shift in a dimension of extent one is a null operation.

```
CALL FOO (e_2)
```

on the other hand, also does a circular shift in the second dimension of C. It now has extent 10. This is a non-trivial operation.

The claim is that the current DAD cannot distinguish these cases.

B.3.2 Lower bounds in the run-time representation

This is not strictly a bug in the DAD definition. Correct code can obviously be generated with the current approach. The observation is simply that if an actual argument has a different subscript lower bound to the dummy argument, a new DAD has to be created on entry to a subroutine. Without the lower bound the DAD created by the calling program can be used within the subroutine—a useful economy.

I also believe that most address computations in the run-time routines will be simplified by standardizing on 0 as the base for all subscripts. My experience is that the only use for the lower-bound/upper-bound pair is to compute the extent from them. Routines are likely to start off with

```
int extent = g_lb - g_ub + 1 ;
```

and subsequently ignore `g_lb` and `g_ub`. It would be more economical just to store the extent.

B.4 The data model

Apparently HPFF has now downgraded *Kernel HPF* to a set of recommendations on programming style. The data model of the HPF 2.0 language will probably include the full HPF 1.0 model.

We should consider reinstating the remaining omitted features of the HPF 1.0 model.

The absence of replicated alignments is a particular case. Several simplifications to the RTL interface could be achieved if we were supporting this feature.

References

[1] *The Distributed Array Descriptor for a PCRC HPF Compiler*, PCRC working paper, February 1996.