Proposals for the FORTRAN 77 Interface

Bryan Carpenter and Xiaoming Li

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

July 18, 1996

1 Introduction

This document combines ideas from earlier drafts, and gives a unified FORTRAN 77 interface for the part of the HPF run-time library needed to translate programs involving simple FORALL statements.

The library will be called from a node-program generated by the HPF translator. The library is responsible for maintaining the data structures which describe HPF distributed arrays, performing address translation between global (HPF-level) array indices and local (node-program) subscripts, and providing the collective communication routines that move array elements from their point of storage to the point of computation.

Although the interface given here is incomplete—it does not support translation of programs involving irregular patterns of data access or Fortran 90 array intrinsics—the essential framework is now in place. Supporting translation of these other features just involves adding new collective operations akin to the `pcrc_shift` and `pcrc_remap` operations of section 11.

1.1 Model of distributed arrays

Distributed arrays are parameterized by the Distributed Array Descriptor (DAD). This C data structure has been described elsewhere. Because FORTRAN 77 cannot directly access the fields of the data structure, the FORTRAN interface treats the DAD as an abstract data type. DAD objects are created with constructor functions which return (integer) handles. These handles are passed to other library functions that operate on the arrays. Where necessary, the fields of the DAD can be made available to the node program through inquiry functions [1]. This mechanism of accessing opaque objects through integer handles is an established practise in various FORTRAN libraries, including the FORTRAN interface to MPI.

Besides the DAD itself, the library manipulates two other kinds of data structure. These can be viewed as sub-components of the DAD. One of these classes corresponds to the `DIM` structure introduced in earlier documents describing the DAD. Throughout this document these structures will be referred to as *range* structures. The second new class will be referred to here as a *group* structure. It has some similarity to the groups (of processes) in MPI but with a more specialised röle—it describes a slice of a process grid.

To use the Fortran interface it should be sufficient to know that the distributed array descriptor has the following logical components:

1. Element type information.

[1] A FORTRAN function call is a significant overhead compared with direct access to a structure member, but in practise relatively few such inquiry functions are needed—most functions on the data structures are more complex operations. This can be viewed as an indication of a successful data abstraction.

2. Rank or dimensionality of the array—an integer `R`.

3. "Embedding" information—the (process) group structure.

4. A base address in local memory—a pointer.

5. Information on the bounds and distribution of index ranges—a vector of `R` range structures.

6. Information on layout of elements in local memory—a vector of `R` integer-valued memory strides.

1.2 Conventions for arguments

Many of the operations in the library are "constructors"—procedures which initialize structures or sub-structures. These often have many arguments which are generally laid out in the order:

```
reference to new structure (if not implicit),
    parameters of new structure,
        reference to parent structure (if any)
```

A reference to a structure is either a handle, or a handle followed by a dimension index.

1.3 This document

The remaining sections in the main body of this document typically start with a list of procedure interfaces in typewriter font, eg

```
INTEGER FUNCTION foo(x)
  REAL x
```

This list is followed by explanatory text which may include example program fragments in small typewriter font, eg

```
n = foo(1.0 + z)
```

As a matter of convention the names of all handles to DADs will be prefixed by `dad`, the names of all handles to range structures will be prefixed by `rng`, and the names of all handles to group structures will be prefixed by `grp`.

The appendix gives full translation of three HPF template codes to FORTRAN node programs.

2 Global state

```
INCLUDE 'pcrc.inc'

SUBROUTINE pcrc_init()

SUBROUTINE pcrc_finalize()
```

The file `pcrc.inc` includes declarations which must be visible in every procedure that calls the library.

The first library call must be to `pcrc_init`. It will initialize the message-passing layer, and perform other necessary global initializations. The last library call must be to `pcrc_finalize`. Both these routines should be called exactly once.

Example: a typical node program has the form

```
PROGRAM main

  INCLUDE 'pcrc.inc'

  ... Declarations

  CALL pcrc_init()

  ... Executable statements

  CALL pcrc_finalize()
END
```

3 Processor grid

```
INTEGER FUNCTION pcrc_new_group_grid(rank, shape)
  INTEGER rank
  INTEGER shape(rank)

SUBROUTINE pcrc_delete_group(grp)
  INTEGER grp
```

`pcrc_new_group_grid` creates a group structure describing a processor grid of the specified `rank` and `shape`, and returns a handle to it. Example: the HPF declaration

```
!HPF$ PROCESSORS P(4)
```

may yield the node fragments

```
INTEGER shape_p(1)
INTEGER grp_p
...

shape_p(1) = 4
grp_p = pcrc_new_group_grid(1, shape_p)
```

Any group structure created with the `pcrc_new_group_grid` call may be destroyed with a `pcrc_delete_group` call if it is no longer needed.

4 Range creation

```
INTEGER FUNCTION pcrc_new_range_distribute(g_lb, g_ub,
                                           distribution,
                                           grp_grid, dim)
  INTEGER g_lb, g_ub
  INTEGER distribution
  INTEGER grp_grid
  INTEGER dim

INTEGER FUNCTION pcrc_new_range_collapse(g_lb, g_ub)
  INTEGER g_lb, g_ub

INTEGER FUNCTION pcrc_new_range_align(g_lb, g_ub,
                                      a, b, rng_parent)
  INTEGER rng_parent
  INTEGER g_lb, g_ub
  INTEGER a, b

INTEGER FUNCTION pcrc_new_range_loop(g_lb, g_ub, g_stride,
                                     a, b, rng_parent)
  INTEGER rng_parent
  INTEGER g_lb, g_ub, g_stride,
  INTEGER a, b

INTEGER FUNCTION pcrc_new_range_copy(rng_parent)
  INTEGER rng_parent

SUBROUTINE delete_range(rng)
  INTEGER rng
```

The `pcrc_new_range_...` calls create “stand-alone” range structure and return integer handles to them.

A range structure describes a distributed index range, for example the range of indices associated with a particular dimension of a particular distributed array.

Range structures normally exist inside array descriptors (see section 5). Stand-alone versions are convenient in various places. For example, they can be used for describing HPF templates (which don't have any associated data) and for describing index ranges of parallel loops.

There are five variants which create ranges with different distribution formats, alignments, or global index ranges.

In all cases `g_lb`, `g_ub` are the global index limits associated with the range.

`pcrc_new_range_distribute` creates a range distributed over a specified grid dimension with a specified distribution format. `pcrc_new_range_collapse` creates a collapsed index range.

Example: the HPF fragment

```
!HPF$ PROCESSORS P(4)
!HPF$ TEMPLATE T(200,200)
!HPF$ DISTRIBUTE T(BLOCK,*) ONTO P
```

could lead to the following fragments in the node program

```
      INTEGER shp_p(1)
      INTEGER grp_p
      INTEGER rng_t1, rng_t2
      ...

      shp_p(1) = 4
      grp_p = pcrc_new_group_grid(1, shp_p)

      rng_t1 = pcrc_new_range_distribute(1, 200, pcrc_block, grp_p, 1)
      rng_t2 = pcrc_new_range_collapse(1, 200)
```

The range structures referenced by `rng_t1`, `rng_t2` constitute a run-time representation of the template `T`. The `distribute` argument of `pcrc_new_range_distribute` is 1 for a block distribution or 2 for a cyclic distribution—for convenience the constants `pcrc_block` and `pcrc_cyclic` are defined in `pcrc.inc`.

`pcrc_new_range_align` creates a range aligned to some parent range through some linear map. The element of the new range with global index `i` is aligned to the element of the parent range with global index `a * i + b`. The constructor `pcrc_new_range_copy` creates an identical copy of some parent range. These two operations are provided for completness: they correspond to `pcrc_set_array_` variants for creating aligned arrays, introduced in the next section.

`pcrc_new_range_loop` is similar to `pcrc_new_range_align` but allows the global index range itself to be strided. This never happens in Fortran array ranges, but is useful for describing index ranges of loops.

Example: Suppose T is defined as above. The parallel loop in the HPF fragment

```
      REAL X(200, 200)
!HPF$ ALIGN WITH TX :: X
      ...

      FORALL (i = 25 : 50 : 2, j = 50 : 100 : 4)
        X (2 * i, j + 50) = ...
```

could yield the node code

```
INTEGER rng_i, rng_j
...

rng_i = pcrc_new_range_loop(25, 50, 2, 2, 0, rng_t1)
rng_j = pcrc_new_range_loop(50, 100, 4, 1, 50, rng_t2)

... node code for the loop
```

The range structures `rng_i`, `rng_j` describe the index ranges of the distributed loop. They are aligned with `X`'s template on the assumption of an "owner computes" rule. These range structures can be used to compute the ranges of local subscripts for `X` in the node program, using the `pcrc_loop_bounds` operation described in section 10. They can also be used to construct temporary arrays for holding intermediate results generated in translating the loop—for example temporaries for communication.

All range structures created with `pcrc_new_range_...` call should be destroyed with a `delete_range` call when they are no longer needed.

5 Array creation

```
INTEGER FUNCTION pcrc_new_array_data(data,
                                     element_type, element_size,
                                     rank, grp)
  choice  data(...)
  INTEGER element_type
  INTEGER element_size
  INTEGER rank
  INTEGER grp

SUBROUTINE pcrc_set_array_distribute(dad, dim, g_lb, g_ub,
                                     distribution,
                                     grp_grid, dim_grid,
  INTEGER dad
  INTEGER dim, dim_grid
```

```
    INTEGER g_lb, g_ub
    INTEGER distribution
    INTEGER grp_grid

  SUBROUTINE pcrc_set_array_collapse(dad, dim, g_lb, g_ub)
    INTEGER dad
    INTEGER dim
    INTEGER g_lb, g_ub

  SUBROUTINE pcrc_set_array_align(dad, dim, g_lb, g_ub,
                                  a, b, rng_parent)
    INTEGER dad
    INTEGER dim
    INTEGER g_lb, g_ub
    INTEGER a, b
    INTEGER rng_parent

  SUBROUTINE pcrc_set_array_loop(dad, dim, g_lb, g_ub, g_stride,
                                 a, b, rng_parent)
    INTEGER dad
    INTEGER dim
    INTEGER g_lb, g_ub
    INTEGER a, b
    INTEGER rng_parent

  SUBROUTINE pcrc_set_array_copy(dad, dim, rng_parent)
    INTEGER dad
    INTEGER dim
    INTEGER rng_parent


  INTEGER FUNCTION pcrc_new_array_copy(data, dad_parent)
    choice  data(...)
    INTEGER dad


  SUBROUTINE pcrc_delete_array(dad)
    INTEGER dad
```

The function `pcrc_new_array_data` allocates a slot for a DAD, initialises the scalar fields in the descriptor, and returns an integer handle. The first argument is the node-program array segment; it may have any type (as flagged by the `choice` specifier) and rank. The type and rank of the distributed array are specified by the following arguments. The final argument should be a group (see section 8). The simplest example of a group is a processor grid.

The initialization of the new DAD is completed by making `rank` calls to an arbitrary combination of the `pcrc_set_array_...` procedures. These calls reference the DAD under construction as their first argument. The `pcrc_set_array_...` calls set the range structures in the DAD. All dimensions of the array must be initialised. The order of the calls defines the memory layout of the array—the first range initialised must correspond to the dimension "most rapidly varying" in local memory, and so on, with the last call initializing the least rapidly varying dimension. So for normal Fortran layout the ranges should be initialised in the natural order `1, ..., rank` for the `dim` arguments.

Every dimension of the array must be distributed over a distinct dimension of the process grid defined in the `pcrc_new_array_data` call, or collapsed.

The different variants of `pcrc_set_array_...` allow for different distribution and alignment formats. The five `pcrc_set_array_...` calls are directly analogous to the `pcrc_new_range_...` variants described in section 4. The call `pcrc_set_array_loop` allows an array to be declared with a strided subscript range. This not particularly useful for translating HPF, but it is provided for completeness.

Example: the HPF declarations

```
!HPF$ PROCESSORS P(4)
!HPF$ TEMPLATE TX(400)
!HPF$ DISTRIBUTE TX(BLOCK) ONTO P

      REAL X(2:99)
!HPF$ ALIGN X(i) WITH TX(2*i+1)
```

could translate to the node fragments

```
      REAL x(100)
      INTEGER dad_x

      ... Initialise grp_p and rng_tx1

      dad_x = pcrc_new_array_data(x, pcrc_real, pcrc_size_real, 1,      &
     &                           grp_p)
      CALL pcrc_set_array_align(dad_x, 1, 2, 99, 2, 1, rng_tx1)

      ... Use dad_x

      CALL pcrc_delete_array(dad_x)
```

Here, `pcrc_real` and `pcrc_size_real` are constants defined in `pcrc.inc`.

Example: the HPF declarations

```
      REAL X(100, 100)
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE X(BLOCK,*) ONTO P
```

could translate to the node fragments

```
      REAL x(0 : 24, 0 : 99)
      INTEGER dad_x

      ... Define grp_p

      dad_x = pcrc_new_array_data(x, pcrc_real, pcrc_size_real, 2,      &
     &                            grp_p)
      CALL pcrc_set_array_distribute(dad_x, 1, 1, 100, pcrc_block,      &
     &                               grp_p, 1)
      CALL pcrc_set_array_collapse(dad_x, 2, 1, 100)

      ... Use dad_x

      CALL pcrc_delete_array(dad_x)
```

Example: the translation of the HPF declarations

```
      REAL X(100, 100), Y(200, 100)
      ...
!HPF$ ALIGN X(i, j) WITH Y(2 * i, j)
```

could involve the node fragments

```
      ... declare arrays x and y
      INTEGER dad_x, dad_y

      ...

      dad_x = pcrc_new_array_data(x, pcrc_integer, pcrc_size_integer,   &
     &                            1, pcrc_grp(dad_y))
      CALL pcrc_set_array_align(dad_x, 1, 1, 100, 2, 0, pcrc_rng(y, 1))
      CALL pcrc_set_array_copy(dad_x, 2, pcrc_rng(y, 2))
```

The functions `pcrc_grp` and `pcrc_rng` will be introduced in section 8.

The call

```
  dad = pcrc_new_array_copy(data, dad_parent)
```

makes a DAD which is identical to `dad_parent` except that the local array segment is replaced with `data`.

All array descriptors created with `pcrc_new_array_...` call should be destroyed with a `pcrc_delete_array` call when they are no longer needed.

6 Section creation

```
  INTEGER FUNCTION pcrc_new_array_section(rank, dad_parent)
    INTEGE rank
```

```
    INTEGER dad_parent

  SUBROUTINE pcrc_set_array_triplet(dad, dim, lb, ub, stride,
                                    dad_parent, dim_parent)
    INTEGER dad, dad_parent
    INTEGER dim, dim_parent
    INTEGER lb, ub, stride

  SUBROUTINE pcrc_set_array_scalar(dad, i, dad_parent, dim_parent)
    INTEGER dad, dad_parent
    INTEGER dim_parent
    INTEGER i
```

The function `pcrc_new_array_section` allocates a slot for a DAD and initialises various fields in a descriptor representing a *section* of some parent array. It returns an integer handle. The arguments are the rank of the new section and the descriptor of the parent array.

The initialization of the new DAD is completed by making R calls to an arbitrary combination of the `pcrc_set_array_triplet` and `pcrc_set_array_scalar` routines to define the section subscripts. Here, R is the rank of the *parent array*. All dimensions of the parent array must be subscripted—the `dim_parent` arguments of these calls must be some permutation of $1, \ldots, R$.

The `pcrc_set_array_triplet` calls set the range structures in the new descriptor. All dimensions of the array must be initialised—the `dim` arguments of these calls must be some permutation of $1, \ldots, S$ where S is the rank of the section. The arguments `lb`, `ub` and `stride` define the lower and upper bounds and stride of a triplet subscript into the parent array dimension.

The `pcrc_set_array_scalar` calls affect the base address and group structure fields (see section 8) in the new descriptor. Their last argument defines a scalar subscript into the parent array dimension.

Example: Suppose the HPF code involves the section

```
X(1 : 100 : 2, 1)
```

in some context. A descriptor `dad_xs` for this section is constructed from the descriptor `dad_x` of `X` by the sequence of calls

```
dad_xs = pcrc_new_array_section(dad_x)
CALL pcrc_set_array_triplet(dad_xs, 1, 1, 100, 2, dad_x, 1)
CALL pcrc_set_array_scalar(dad_xs, 1, dad_x, 2)
```

7 Array dummy arguments

```
  INTEGER FUNCTION pcrc_str(dad, dim)
```

```
    INTEGER dad
    INTEGER dim

  SUBROUTINE pcrc_reset_array_bounds(dad, dim,
          lb, ub, stride, lb_parent, ub_parent, stride_parent)
    INTEGER dad
    INTEGER dim
    INTEGER lb, ub, stride, lb_parent, ub_parent, stride_parent
```

Often, when translating Fortran 90 to FORTRAN 77, multi-dimensional array dummy arguments must be translated to one-dimensional array dummies, and the index arithmetic for the multi-dimensional dummy must be performed explicitly in the generated code. This arithmetic requires the "memory strides" of the dimensions.

The function `pcrc_str` returns the memory stride (in units of the size of the array element) associated with a particular dimension of an array. This memory stride information should be included in the array descriptor, so that `pcrc_str` becomes a simple inquiry function.

Example: in the (sequential) Fortran 90 code

```
REAL x(10, 10, 10)

CALL foo(x(6, :, :))
...

SUBROUTINE foo(y)
REAL y(10, 10)
...

y(i, j) = ...

END
```

it is not possible to declare `y` as a two-dimensional array in the translated code. The memory layout of the section `x(6, :, :)` is different from any FORTRAN 77 two dimensional array.

An unrelated complication with array dummy arguments arises because the DAD contains information on the lower bound and stride of array subscripts. These parameters generally changes every time an array is passed to a procedure (although the *extent* implied by the bounds must be preserved). The routine `pcrc_reset_array_bounds` changes the bounds for dimension `dim` to `lb`, `ub`, `stride`, modifying all fields in the DAD which depend on this value. For convenience, it returns the original values of the bounds parameters in `lb_parent`, `ub_parent`, `stride_parent`.

`pcrc_reset_array_bounds` should be called for every dimension of every array dummy argument of every procedure, once on entry to the procedure, and once on exit. (The exit call, to restore the original value, can be eliminated only if the calling program *always* creates a temporary DAD for every array actual argument of every procedure call).

The FORTRAN 77 translation of the subroutine `foo` could be

```
      SUBROUTINE foo(y, dad_y)
        REAL y(0 : *)
        INTEGER dad
        INTEGER ylb1_actual, yub1_actual, ystride1_actual
        INTEGER ylb2_actual, yub2_actual, ystride2_actual
        INTEGER dummy1, dummy2, dummy3

        CALL pcrc_reset_array_bounds(dad_y, 1, 1, 10, 1,                &
     &           ylb1_actual, yub1_actual, ystride1_actual)
        CALL pcrc_reset_array_bounds(dad_y, 2, 1, 10, 1,                &
     &           ylb2_actual, yub2_actual, ystride2_actual)
        ...

        y((i - 1) * pcrc_str(dad_y, 1) +                                &
     &    (j - 1) * pcrc_str(dad_y, 2)) = ...

        CALL pcrc_reset_array_bounds(dad_y, 1,                          &
     &           ylb1_actual, yub1_actual, ystride1_actual,             &
     &           dummy1, dummy2, dummy3)
        CALL pcrc_reset_array_bounds(dad_y, 2,                          &
     &           ylb2_actual, yub2_actual, ystride2_actual,             &
     &           dummy1, dummy2, dummy3)
      END
```

8 Mapping inquiry functions

```
INTEGER FUNCTION pcrc_rng(dad, dim)
  INTEGER dad
  INTEGER dim

INTEGER FUNCTION pcrc_grp(dad)
  INTEGER dad

LOGICAL FUNCTION pcrc_on(grp)
  INTEGER grp
```

The inquiry function `pcrc_rng` returns a handle to a particular `DIM` structure in the DAD. Note that if the array descriptor represents an array section, in particular a scalar-subscripted array section, the `dim` field counts position in the effective dimensions of the *section*, *not* in the dimensions of the parent array.

To fully describe scalar-subscripted array sections a structure is added to the DAD describing the *group* (of processes) on which the data of the section resides. In the simplest case of an unsubscripted array, this is just the process grid on which the array is distributed. In general a section is localised to some "slice" of a process grid—a subgroup obtained by fixing one or more of the coordinates in the grid to some specified values.

The inquiry function `pcrc_grp` returns a handle to the group for a given DAD. The function `pcrc_on` returns the value `.TRUE.` if the local process is part of the process group defined by its argument, and `.FALSE.` if not. It does this by comparing the slice coordinate for each scalar subscripted dimension with the local process coordinate.

These functions are needed to translate dummy arguments with inheritted mapping. Example: the HPF subroutine

```
      SUBROUTINE foo(y)
      REAL y(10)
!HPF$ INHERIT y
      INTEGER i

      FORALL (i = 1 : 10)  y(i) = ...

      END
```

could translate to the node subroutine

```
      SUBROUTINE foo(y, dad_y)
        REAL y(0 : *)
        INTEGER dad_y

        INTEGER ylb_actual, yub_actual, ystride_actual
        INTEGER dummy1, dummy2, dummy3

        INTEGER str_y1
        INTEGER ll, lu, ls, i

        CALL pcrc_reset_array_bounds(dad_y, 1, 1, 10, 1,              &
     &           ylb_actual, yub_actual, ystride_actual)

        IF(pcrc_on(pcrc_grp(dad_y)) THEN
          str_y1 = pcrc_str(dad_y, 1)
          CALL pcrc_loop_bounds(pcrc_rng(dad_y, 1), ll, lu, ls)
          DO i = ll, lu, ls
            y(i * str_y1) = ...
          END DO
        END IF

        CALL pcrc_reset_array_bounds(dad_y, 1,                        &
     &           ylb_actual, yub_actual, ystride_actual,              &
```

```
     &           dummy1, dummy2, dummy3)
      END
```

9 Run-time allocation of temporary arrays

```
REAL pcrc_real_stack(...)
...

INTEGER FUNCTION pcrc_real_alloc(size)
  INTEGER size
...

SUBROUTINE pcrc_real_free(base)
  INTEGER base
...

INTEGER FUNCTION pcrc_size(dad)
  INTEGER dad

SUBROUTINE pcrc_array_reset_base(dad, data)
  INTEGER dad
  choice data(...)
```

The one dimensional array `pcrc_real_stack` is visible in every procedure which includes `pcrc.inc`. Space for `REAL` arrays can be allocated from this vector.

The function `pcrc_real_alloc` allocates space for a temporary array on the stack. Its argument is the size of the vector to be allocated. The function returns an index into the array `pcrc_real_stack` identifying the base address of the vector.

The function `pcrc_real_free` will free the space allocated by the matching allocation call.

These variables and functions are used to allocate temporaries whose size is not known at compile time. Similar vectors and procedures are declared for each FORTRAN 77 primitive type (in practice the differently-typed stack vectors may be equivalenced to a single global stack vector).

The inquiry function `pcrc_size` returns the size of the data vector needed to store the local segment of the array described by its argument.

Example: the translation of the HPF subroutine

```
      SUBROUTINE FOO(X, Y)
      REAL X(100), Y(100)
!HPF$ INHERIT X, Y

      FORALL (i = 1 : 100) X(i) = X(i) + Y(i)
```

```
      RETURN
      END
```

involves copying the values in `Y` to a temporary array with the same mapping as `X` before updating `X`. Because the mapping of `X` is not known at compile time, the size of the local array segments is not known either. A possible translation is

```
      SUBROUTINE foo(x, dad_x, y, dad_y)
        REAL x(0 : *), y(0 : *)
        INTEGER dad_x, dad_y

        INCLUDE 'pcrc.inc'

        INTEGER dad_tmp1
        INTEGER bas_tmp1
        INTEGER str_x_1

        INTEGER i, ll, lu, ls

        ... Calls to pcrc_reset_array_bounds

        grp_x = pcrc_grp(dad_x)

        bas_tmp1 = pcrc_real_alloc(pcrc_size(dad_x))
        dad_tmp1 = pcrc_new_array_copy(pcrc_real_stack(bas_tmp1), dad_x)

        CALL pcrc_remap(dad_tmp1, dad_y)

        IF(pcrc_on(pcrc_grp(dad_x))) THEN
          str_x_1  = pcrc_str(dad_x, 1)

          CALL pcrc_loop_bounds(pcrc_rng(dad_x, 1), ll, lu, ls)

          DO i = ll, lu, ls
            x(i * str_x_1) = x(i * str_x_1) +                          &
     &                       pcrc_real_stack(bas_tmp1 + i)
          END DO
        END IF

        CALL pcrc_delete_array(dad_tmp1)
        CALL pcrc_real_free(bas_tmp1)

        ... Calls to pcrc_reset_array_bounds
      END
```

After copying the values in **`y`** to the stack-allocated temporary by the **`pcrc_remap`** call (see section 11), **`x`** is updated in terms of values in pcrc_real_stack.

In the example tmp1 is one-dimensional and one could safely assume that its elements were contiguous in pcrc_real_stack. A multi-dimensional array would be mapped into the stack array with some strides, and elements would be accessed through the memory strides stored in the DAD, just as for dummy arguments.

In the example, the allocated array was the same shape as an existing array, and the DAD for that array could be passed to the pcrc_size inquiry. In general a new DAD is constructed before calling pcrc_size. Because the inquiry function does not depend on the value of the base address field in the DAD, an arbitrary value may be inserted initially. This can subsequently be overwritten with the operation pcrc_array_reset_base. For example

```
dad_x = pcrc_new_array_data(pcrc_real_stack,
                            pcrc_real, pcrc_size_real, 1, grp_p)
... set range(s)

bas_x = pcrc_real_alloc(pcrc_size(dad_x))
pcrc_reset_array_base(dad_x, pcrc_real_stack(bas_x))
```

It was tacitly assumed here that space will be allocated and deallocated in FIFO order. This simplifies the implementation of the allocation and deallocation calls. Note however, that the same interface equally well supports heap allocation. Eventually heap allocation will be needed in addition to, or in place of, stack allocation, to support F90 dynamically allocated arrays.

10 Address translation

```
INTEGER FUNCTION pcrc_glob(rng, l_i)
  INTEGER rng
  INTEGER l_i

INTEGER FUNCTION pcrc_loc(rng, g_i)
  INTEGER rng
  INTEGER g_i

SUBROUTINE pcrc_loop_bounds(rng, l_lb, l_ub, l_stride)
  INTEGER rng
  INTEGER l_lb, l_ub, l_stride
```

The routines pcrc_glob and pcrc_loc translate betwen global array indices and local segment subscripts. Their first argument is a handle to a DIM structure describing the relevant index range.

Local subscripts now start at 0 (*not* 1). This simplifies subscript arithmetic, especially in the case of dummy arguments or temporary arrays, where multi-dimensional array segments typically have to be "flattened" to one-dimensional arrays in the node program.

Logically, local subscripts remain subscripts into the multi-dimensional segment of the template held by the local process. They do not directly encode any information about the memory layout of an array (in particular the node program must explicitly include any memory-stride multipliers required to flatten array segments).

The function `pcrc_loop_bounds` returns the bounds of a DO loop needed to enumerate the local subscripts associated with a range.

An example given in section 4 can now be completed. The parallel loop in the HPF fragment

```
      REAL X(200, 200)
!HPF$ ALIGN WITH TX :: X
      ...

      FORALL (i = 25 : 50 : 2, j = 50 : 100 : 4)
        X (2 * i, j + 50) = ...
```

could yield the node code

```
INTEGER rng_i, rng_j
INTEGER i_lb, i_ub, i_stride, j_lb, j_ub, j_stride
...

rng_i = pcrc_new_range_loop(25, 50, 2, 0, 2, rng_t1)
rng_j = pcrc_new_range_loop(50, 100, 4, 50, 1, rng_t2)

pcrc_loop_bounds(rng_i, i_lb, i_ub, i_stride)
pcrc_loop_bounds(rng_j, j_lb, j_ub, j_stride)

DO i = i_lb, i_ub, i_stride
  DO j = j_lb, j_ub, j_stride
    x(i, j) = ...
```

11 Communication

```
SUBROUTINE pcrc_shift(dad, dad_source, dim, amount)
  INTEGER dad, dad_source
  INTEGER dim, amount

SUBROUTINE pcrc_remap(dad, dad_source)
  INTEGER dad, dad_source

INTEGER FUNCTION pcrc_detect_communication(dad, dad_source,
                                           dim, l, u, stride,
```

```
                                              a, b,
                                              m, c, amount)
   INTEGER dad, dad_source
   INTEGER dim
   INTEGER l, u, stride
   INTEGER a, b
   REAL m, c
   INTEGER amount
```

The `pcrc_shift` operation is similar of EOSHIFT in Fortran 90, with the constraint that the source and destination arrays (described by the DADs `dad_source` and `dad`) should be aligned arrays, in the HPF sense.

The `pcrc_remap` operation copies one array (or array section) to another. The two arrays must be the same shape, but their distributions need not be related.

`pcrc_detect_communication` can be used in translating certain FORALL statements to ascertain the communication pattern required. It may be that no communication is required. If a communication is required, a simple shift may be sufficient, or a general remap operation may be required. These cases are encoded in the result of the function.

A Level 1 Template

```
      PROGRAM MAIN
      REAL X(2:99), Y(100)
!HPF$ PROCESSORS P(4)
!HPF$ TEMPLATE TX(400),TY(-200:199)
!HPF$ DISTRIBUTE TX(BLOCK) ONTO P
!HPF$ DISTRIBUTE TY(BLOCK) ONTO P
!HPF$ ALIGN X(i) WITH TX(2*i+1)
!HPF$ ALIGN Y(i) WITH TY(3*i-150)

      FORALL (i=2:99) X(i) = 0
      FORALL (i=1:100) Y(i) = i

      FORALL (i=2:99) X(i) = Y(i+1) + Y(i-1)

      PRINT *, (X(i),i=2,99)
      END
```

A.1 Notes on the translation

Communications are optimised by calling `pcrc_detect_communication` for each term on the RHS of the assignment in the main forall loop.

If the returned status is 0, no communication is needed to access the array elements in the term, and the original array `y` appears in the DO loop which performs the assignment. If the returned status is 1, a simple shift is sufficient to move the array elements into the required place. They are copied to a temporary aligned with `y` by a `pcrc_shift` operation. If the returned status is 2, a simple shift is inadequate, and the general `pcrc_remap` operation is invoked to copy the elements to a temporary aligned with `x`. In this case it is also necessary to set up a temporary DAD describing the section of the `y` array from which the term originates.

```
      PROGRAM main

        IMPLICIT NONE

        INCLUDE 'pcrc.inc'

        INTEGER shp_p(1)
        INTEGER grp_p

        INTEGER rng_tx1, rng_ty1

        REAL x(0 : 99), y(0 : 99)
        REAL tmp1(0 : 99), tmp2(0 : 99), tmp3(0 : 99), tmp4(0 : 99)

        INTEGER dad_x, dad_y
        INTEGER dad_tmp1, dad_tmp2, dad_tmp3, dad_tmp4
        INTEGER dad_ys

        INTEGER i, ll, lu, ls, amount1, amount2, status1, status2
        INTEGER i1, i2
        REAL u1, v1, u2, v2

        INTEGER shp_s(1)
        INTEGER grp_s

        REAL tmp5(0 : 97)

        INTEGER dad_tmp5

        CALL pcrc_init()

! Define processor arrangement

        shp_p(1) = 4
        grp_p = pcrc_new_group_grid(1, shp_p)

! Define templates

        rng_tx1 = pcrc_new_range_distribute(1, 400, pcrc_block,         &
     &                                     grp_p, 1)

        rng_ty1 = pcrc_new_range_distribute(-200, 199, pcrc_block,      &
     &                                     grp_p, 1)

! Define main arrays

        dad_x = pcrc_new_array_data(x, pcrc_real, pcrc_size_real,       &
```

```
     &                              1, grp_p)
        CALL pcrc_set_array_align(dad_x, 1, 2, 99, 2, 1, rng_tx1)

        dad_y = pcrc_new_array_data(y, pcrc_real, pcrc_size_real,       &
     &                              1, grp_p)
        CALL pcrc_set_array_align(dad_y, 1, 1, 100, 3, -150, rng_ty1)

! Do parallel loop x(i) = 0

        CALL pcrc_loop_bounds(pcrc_rng(dad_x, 1), ll, lu, ls)
        DO i = ll, lu, ls
          x (i) = 0
        END DO

! Do parallel loop y(i) = i

        CALL pcrc_loop_bounds(pcrc_rng(dad_y, 1), ll, lu, ls)
        DO i = ll, lu, ls
          y (i) = pcrc_glob(pcrc_rng(dad_y, 1), i)
        END DO

! Define and write temporary for remapped y(i + 1)

        status1 = pcrc_detect_communication(dad_x, dad_y, 1, 2, 99,     &
     &                                      1, 1, 1, u1, v1, amount1)

        IF(status1 .EQ. 1) THEN
          dad_tmp1 = pcrc_new_array_copy(tmp1, dad_y)

          CALL pcrc_shift(dad_tmp1, dad_y, 1, amount1)
        ELSE IF(status1 .EQ. 2) THEN
          dad_tmp3 = pcrc_new_array_copy(tmp3, dad_x)

          dad_ys = pcrc_new_array_section(1, dad_y)
          CALL pcrc_set_array_triplet(dad_ys, 1, 3, 100, 1, dad_y, 1)

          CALL pcrc_remap(dad_tmp3, dad_ys)

          CALL pcrc_delete_array(dad_ys)
        END IF

! Define and write temporary for remapped y(i - 1)

        status2 = pcrc_detect_communication(dad_x, dad_y, 1, 2, 99,     &
     &                                      1, 1, -1,  u2, v2, amount2)

        IF(status2 .EQ. 1) THEN
```

```
    dad_tmp2 = pcrc_new_array_copy(tmp2, dad_y)

    CALL pcrc_shift(dad_tmp2, dad_y, 1, amount2)
  ELSE IF(status2 .EQ. 2) THEN
    dad_tmp4 = pcrc_new_array_copy(tmp4, dad_x)

    dad_ys = pcrc_new_array_section(1, dad_y)
    CALL pcrc_set_array_triplet(dad_ys, 1, 1, 98, 1, dad_y, 1)

    CALL pcrc_remap(dad_tmp4, dad_ys)

    CALL pcrc_delete_array(dad_ys)
  END IF

! Do parallel loop  y(i) = y(i+1) + y(i-1)

  CALL pcrc_loop_bounds(pcrc_rng(dad_x, 1), ll, lu, ls)
  DO i = ll, lu, ls
    i1 = NINT(u1 * i + v1)
    i2 = NINT(u2 * i + v2)

    IF(status1 .EQ. 0 .AND. status2 .EQ. 0) THEN
      x(i) = y(i1) + y(i2)
    ELSE IF(status1 .EQ. 0 .AND. status2 .EQ. 1) THEN
      x(i) = y(i1) + tmp2(i2)
    ELSE IF(status1 .EQ. 0 .AND. status2 .EQ. 2) THEN
      x(i) = y(i1) + tmp4(i2)
    ELSE IF(status1 .EQ. 1 .AND. status2 .EQ. 0) THEN
      x(i) = tmp1(i1) + y(i2)
    ELSE IF(status1 .EQ. 1 .AND. status2 .EQ. 1) THEN
      x(i) = tmp1(i1) + tmp2(i2)
    ELSE IF(status1 .EQ. 1 .AND. status2 .EQ. 2) THEN
      x(i) = tmp1(i1) + tmp4(i2)
    ELSE IF(status1 .EQ. 2 .AND. status2 .EQ. 0) THEN
      x(i) = tmp3(i1) + y(i2)
    ELSE IF(status1 .EQ. 2 .AND. status2 .EQ. 1) THEN
      x(i) = tmp3(i1) + tmp2(i2)
    ELSE
      x(i) = tmp3(i1) + tmp4(i2)
    END IF
  END DO

  IF(status1 .EQ. 1) THEN
    CALL pcrc_delete_array(dad_tmp1)
  ELSE IF(status1 .EQ. 2) THEN
    CALL pcrc_delete_array(dad_tmp3)
  END IF
```

```
        IF(status2 .EQ. 1) THEN
          CALL pcrc_delete_array(dad_tmp2)
        ELSE IF(status1 .EQ. 2) THEN
          CALL pcrc_delete_array(dad_tmp4)
        END IF

! PRINT *, (X(i),i=2,99)

        grp_s = pcrc_new_group_grid(0, shp_s)

        dad_tmp5 = pcrc_new_array_data(tmp5, pcrc_real, pcrc_size_real, &
     &                                 1, grp_s)
        CALL pcrc_set_array_collapse(dad_tmp5, 1, 0, 97)

        CALL pcrc_remap(dad_tmp5, dad_x)

        IF(pcrc_on(grp_s)) THEN
          PRINT *, (tmp5(i), i = 0, 97)
        ENDIF

        CALL pcrc_delete_array(dad_tmp5)

! Reclaim memory

        CALL pcrc_delete_array(dad_y)
        CALL pcrc_delete_array(dad_x)

        CALL pcrc_delete_range(rng_ty1)
        CALL pcrc_delete_range(rng_tx1)

        CALL pcrc_finalize()
      END
```

B Level 2 Template

```
      PROGRAM MAIN
      REAL X(2:99), Y(100)
!HPF$ PROCESSORS P(4)
!HPF$ TEMPLATE TX(400),TY(-200:199)
!HPF$ DISTRIBUTE TX(BLOCK) ONTO P
!HPF$ DISTRIBUTE TY(BLOCK) ONTO P
!HPF$ ALIGN X(i) WITH TX(2*i+1)
!HPF$ ALIGN Y(i) WITH TY(3*i-150)

      FORALL (i=2:99) X(i) = 0
      FORALL (i=1:100) Y(i) = i

      FORALL (i=2:99) X(i) = Y(i+1) + Y(i-1)

      CALL FOO(X(2:98:2),Y(1:99:2))
      CALL FOO(X(3:99:2),Y(2:100:2))

      PRINT *, (X(i),i=2,99)
      END

      SUBROUTINE FOO(X,Y)
      REAL X(49), Y(50)
!HPF$ INHERIT X,Y

      FORALL (i=1:49) X(i) = X(i) - Y(i) - Y(i+1)

      RETURN
      END
```

B.1 Notes on the translation

The new feature is the procedure call.

DAD's are initialised to describe the array sections passed to the procedure. Inside the function, temporary arrays are allocated to hold the non-local terms on the RHS of the assignment. Space for these arrays is allocated in `pcrc_real_stack`. They are aligned with `x` using `pcrc_new_array_copy`.

(The communications inside the procedure could also be optimized with `pcrc_detect_communications` to eliminate the `pcrc_remap` call if possible. For the sake of simplicity, we left out this optimization here.)

```
      PROGRAM main

        IMPLICIT NONE

        INCLUDE 'pcrc.inc'

        INTEGER shp_p(1)
        INTEGER grp_p

        INTEGER rng_tx1, rng_ty1

        REAL x(0 : 99), y(0 : 99)
        REAL tmp1(0 : 99), tmp2(0 : 99), tmp3(0 : 99), tmp4(0 : 99)

        INTEGER dad_x, dad_y
        INTEGER dad_tmp1, dad_tmp2, dad_tmp3, dad_tmp4
        INTEGER dad_xs, dad_ys

        INTEGER i, ll, lu, ls, amount1, amount2, status1, status2
        INTEGER i1, i2
        REAL u1, v1, u2, v2

        INTEGER shp_s(1)
        INTEGER grp_s

        REAL tmp5(0 : 97)

        INTEGER dad_tmp5

        CALL pcrc_init()

! Define processor arrangement

        shp_p(1) = 4
        grp_p = pcrc_new_group_grid(1, shp_p)

! Define templates

        rng_tx1 = pcrc_new_range_distribute(1, 400, pcrc_block,
     &                                     grp_p, 1)

        rng_ty1 = pcrc_new_range_distribute(-200, 199, pcrc_block,      &
     &                                     grp_p, 1)

! Define main arrays

        dad_x = pcrc_new_array_data(x, pcrc_real, pcrc_size_real,       &
```

```
     &                              1, grp_p)
        CALL pcrc_set_array_align(dad_x, 1, 2, 99, 2, 1, rng_tx1)

        dad_y = pcrc_new_array_data(y, pcrc_real, pcrc_size_real,       &
     &                              1, grp_p)
        CALL pcrc_set_array_align(dad_y, 1, 1, 100, 3, -150, rng_ty1)

! Do parallel loop x(i) = 0

        CALL pcrc_loop_bounds(pcrc_rng(dad_x, 1), ll, lu, ls)
        DO i = ll, lu, ls
          x(i) = 0
        END DO

! Do parallel loop y(i) = i

        CALL pcrc_loop_bounds(pcrc_rng(dad_y, 1), ll, lu, ls)
        DO i = ll, lu, ls
          y(i) = pcrc_glob(pcrc_rng(dad_y, 1), i)
        END DO

! Define and write temporary for remapped y(i + 1)

        status1 = pcrc_detect_communication(dad_x, dad_y, 1, 2, 99, 1,  &
     &                                      1, 1, u1, v1, amount1)

        IF(status1 .EQ. 1) THEN
          dad_tmp1 = pcrc_new_array_copy(tmp1, dad_y)

          CALL pcrc_shift(dad_tmp1, dad_y, 1, amount1)
        ELSE IF(status1 .EQ. 2) THEN
          dad_tmp3 = pcrc_new_array_copy(tmp3, dad_x)

          dad_ys = pcrc_new_array_section(1, dad_y)
          CALL pcrc_set_array_triplet(dad_ys, 1, 3, 100, 1, dad_y, 1)

          CALL pcrc_remap(dad_tmp3, dad_ys)

          CALL pcrc_delete_array(dad_ys)
        END IF

! Define and write temporary for remapped y(i - 1)

        status2 = pcrc_detect_communication(dad_x, dad_y, 1, 2, 99, 1,  &
     &                                      1, -1, u2, v2, amount2)

        IF(status2 .EQ. 1) THEN
```

```
          dad_tmp2 = pcrc_new_array_copy(tmp2, dad_y)

          CALL pcrc_shift(dad_tmp2, dad_y, 1, amount2)
        ELSE IF(status2 .EQ. 2) THEN
          dad_tmp4 = pcrc_new_array_copy(tmp4, dad_x)

          dad_ys = pcrc_new_array_section(1, dad_y)
          CALL pcrc_set_array_triplet(dad_ys, 1, 1, 98, 1, dad_y, 1)

          CALL pcrc_remap(dad_tmp4, dad_ys)

          CALL pcrc_delete_array(dad_ys)
        END IF

! Do parallel loop  y(i) = y(i+1) + y(i-1)

        CALL pcrc_loop_bounds(pcrc_rng(dad_x, 1), ll, lu, ls)
        DO i = ll, lu, ls
          i1 = NINT(u1 * i + v1)
          i2 = NINT(u2 * i + v2)

          IF(status1 .EQ. 0 .AND. status2 .EQ. 0) THEN
            x(i) = y(i1) + y(i2)
          ELSE IF(status1 .EQ. 0 .AND. status2 .EQ. 1) THEN
            x(i) = y(i1) + tmp2(i2)
          ELSE IF(status1 .EQ. 0 .AND. status2 .EQ. 2) THEN
            x(i) = y(i1) + tmp4(i2)
          ELSE IF(status1 .EQ. 1 .AND. status2 .EQ. 0) THEN
            x(i) = tmp1(i1) + y(i2)
          ELSE IF(status1 .EQ. 1 .AND. status2 .EQ. 1) THEN
            x(i) = tmp1(i1) + tmp2(i2)
          ELSE IF(status1 .EQ. 1 .AND. status2 .EQ. 2) THEN
            x(i) = tmp1(i1) + tmp4(i2)
          ELSE IF(status1 .EQ. 2 .AND. status2 .EQ. 0) THEN
            x(i) = tmp3(i1) + y(i2)
          ELSE IF(status1 .EQ. 2 .AND. status2 .EQ. 1) THEN
            x(i) = tmp3(i1) + tmp2(i2)
          ELSE
            x(i) = tmp3(i1) + tmp4(i2)
          END IF
        END DO

        IF(status1 .EQ. 1) THEN
          CALL pcrc_delete_array(dad_tmp1)
        ELSE IF(status1 .EQ. 2) THEN
          CALL pcrc_delete_array(dad_tmp3)
        END IF
```

```
        IF(status2 .EQ. 1) THEN
          CALL pcrc_delete_array(dad_tmp2)
        ELSE IF(status1 .EQ. 2) THEN
          CALL pcrc_delete_array(dad_tmp4)
        END IF

! Define arguments and make call to 'foo'

        dad_xs = pcrc_new_array_section(1, dad_x)
        CALL pcrc_set_array_triplet(dad_xs, 1, 2, 98, 2, dad_x, 1)

        dad_ys = pcrc_new_array_section(1, dad_y)
        CALL pcrc_set_array_triplet(dad_ys, 1, 1, 99, 2, dad_y, 1)

        CALL foo(x, dad_xs, y, dad_ys)

        CALL pcrc_delete_array(dad_ys)
        CALL pcrc_delete_array(dad_xs)

! Define arguments and make call to 'foo'

        dad_xs = pcrc_new_array_section(1, dad_x)
        CALL pcrc_set_array_triplet(dad_xs, 1, 3, 99, 2, dad_x, 1)

        dad_ys = pcrc_new_array_section(1, dad_y)
        CALL pcrc_set_array_triplet(dad_ys, 1, 2, 100, 2, dad_y, 1)

        CALL foo(x, dad_xs, y, dad_ys)

        CALL pcrc_delete_array(dad_ys)
        CALL pcrc_delete_array(dad_xs)

! PRINT *, (X(i),i=2,99)

        grp_s = pcrc_new_group_grid(0, shp_s)

        dad_tmp5 = pcrc_new_array_data(tmp5, pcrc_real, pcrc_size_real, &
     &                                1, grp_s)
        CALL pcrc_set_array_collapse(dad_tmp5, 1, 0, 97)

        CALL pcrc_remap(dad_tmp5, dad_x)

        IF(pcrc_on(grp_s)) THEN
          PRINT *, (tmp5(i), i = 0, 97)
        ENDIF
```

```
        CALL pcrc_delete_array(dad_tmp5)

! Reclaim memory

        CALL pcrc_delete_array(dad_y)
        CALL pcrc_delete_array(dad_x)

        CALL pcrc_delete_range(rng_ty1)
        CALL pcrc_delete_range(rng_tx1)

        CALL pcrc_finalize()
      END


      SUBROUTINE foo(x, dad_x, y, dad_y)
        IMPLICIT NONE

        REAL x(0 : *), y(0 : *)
        INTEGER dad_x, dad_y

        INCLUDE 'pcrc.inc'

        INTEGER xlb_actual, xub_actual, xstride_actual
        INTEGER ylb_actual, yub_actual, ystride_actual
        INTEGER dummy1, dummy2, dummy3

        INTEGER grp_x

        INTEGER dad_tmp1, dad_tmp2, dad_ys
        INTEGER bas_tmp1, bas_tmp2
        INTEGER str_x_1

        INTEGER i, ll, lu, ls

        CALL pcrc_reset_array_bounds(dad_x, 1, 1, 49, 1,                 &
     &           xlb_actual, xub_actual, xstride_actual)
        CALL pcrc_reset_array_bounds(dad_y, 1, 1, 50, 1,                 &
     &           ylb_actual, yub_actual, ystride_actual)

        grp_x = pcrc_grp(dad_x)

! Allocate and write temporary for remapped y(i)

        bas_tmp1 = pcrc_real_alloc(pcrc_size(dad_x))
        dad_tmp1 = pcrc_new_array_copy(pcrc_real_stack(bas_tmp1), dad_x)

        dad_ys = pcrc_new_array_section(1, dad_y)
```

```
        CALL pcrc_set_array_triplet(dad_ys, 1, 1, 49, 1, dad_y, 1)

        CALL pcrc_remap(dad_tmp1, dad_ys)

        CALL pcrc_delete_array(dad_ys)

! Allocate and write temporary for remapped y(i + 1)

        bas_tmp2 = pcrc_real_alloc(pcrc_size(dad_x))
        dad_tmp2 = pcrc_new_array_copy(pcrc_real_stack(bas_tmp2), dad_x)

        dad_ys = pcrc_new_array_section(1, dad_y)
        CALL pcrc_set_array_triplet(dad_ys, 1, 2, 50, 1, dad_y, 1)

        CALL pcrc_remap(dad_tmp2, dad_ys)

        CALL pcrc_delete_array(dad_ys)

! Do parallel loop

        IF(pcrc_on(grp_x)) THEN
          str_x_1  = pcrc_str(dad_x, 1)

          CALL pcrc_loop_bounds(pcrc_rng(dad_x, 1), ll, lu, ls)

          DO i = ll, lu, ls
            x(i * str_x_1) = x(i * str_x_1) -                          &
     &                       pcrc_real_stack(bas_tmp1 + i) -            &
     &                       pcrc_real_stack(bas_tmp2 + i)
          END DO
        END IF

! Reclaim memory

        CALL pcrc_delete_array(dad_tmp2)
        CALL pcrc_real_free(bas_tmp2)

        CALL pcrc_delete_array(dad_tmp1)
        CALL pcrc_real_free(bas_tmp1)

        CALL pcrc_reset_array_bounds(dad_y, 1,                          &
     &           ylb_actual, yub_actual, ystride_actual,                &
     &           dummy1, dummy2, dummy3)
        CALL pcrc_reset_array_bounds(dad_x, 1,                          &
     &           xlb_actual, xub_actual, xstride_actual,                &
     &           dummy1, dummy2, dummy3)
      END
```

C Level 3 Template

```
      PROGRAM MAIN
      PARAMETER (N=100)
      REAL X(N,N), Y(N,N)
!HPF$ PROCESSORS P(4)
!HPF$ TEMPLATE TX(200,200),TY(100,100)
!HPF$ DISTRIBUTE TX(BLOCK,*) ONTO P
!HPF$ DISTRIBUTE TY(BLOCK,*) ONTO P
!HPF$ ALIGN X(i,j) WITH TX(2*i,2*j)
!HPF$ ALIGN Y(i,j) WITH TY(i,j)

      FORALL (i=1:N,j=1:100) X(i,j) = 0
      FORALL (i=1:100,j=1:N) Y(i,j) = i + j

      FORALL (i=26:50,j=26:50) X(i,j) = Y(i+1,j) + Y(i,j+1)
      FORALL (i=1:100:2,j=1:N:2,X(i,j)/=0) X(i,j) = 1

      CALL FOO(X(1:100:2,1))

      PRINT *, (X(i,1),i=1,100,2)
      END

      SUBROUTINE FOO(X)
      REAL X(50)
!HPF$ INHERIT X
      REAL Y(50)
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE Y(BLOCK) ONTO P

      FORALL (i=1:50) Y(i) = 2
      FORALL (i=1:50) X(i) = Y(i) + 1

      RETURN
      END
```

C.1 Notes on the translation

The new feature is the use of multi-dimensional arrays.

In the translation here we have flattened *all* array segments to one dimension. So the `x` segment in the main program, for example, is declared as `x(0 : 2499)` rather than `x(0 : 24, 0 : 99)`. This choice was not essential, but it makes for a more uniform treatment of arrays.

Again, for simplicity, all communications are performed with `pcrc_remap`. Alternatively `pcrc_detect_communications` can be called for all dimensions of the arrays involved.

In the function call, a one dimensional section of the array is passed as the actual argument. A DAD describing the section is constructed (this time involving a call to `pcrc_set_array_scalar`). Notice that it is also necessary to subscript the local segment with the corresponding local subscript before passing it to the function.

```
      PROGRAM main
        IMPLICIT NONE

        INCLUDE 'pcrc.inc'

        INTEGER shp_p(1)
        INTEGER grp_p

        INTEGER rng_tx1, rng_tx2, rng_ty1, rng_ty2

        INTEGER rng_i, rng_j

        REAL x(0 : 9999), y(0 : 2499)

        INTEGER dad_x, dad_y

        INTEGER dad_ys, dad_xs

!        REAL tmp1(0 : 2499), tmp2(0 : 2499)
        REAL tmp1(0 : 9999), tmp2(0 : 9999)

        INTEGER dad_tmp1, dad_tmp2

        INTEGER i, j, ll1, lu1, ls1, ll2, lu2, ls2

        INTEGER shp_s(1)
        INTEGER grp_s

        REAL tmp3(0 : 49)
        INTEGER dad_tmp3


        CALL pcrc_init()

! Define processor arrangement

        shp_p(1) = 4
        grp_p = pcrc_new_group_grid(1, shp_p)

! Define templates

        rng_tx1 = pcrc_new_range_distribute(1, 200, pcrc_block,         &
     &                                      grp_p, 1)
        rng_tx2 = pcrc_new_range_collapse(1, 200)

        rng_ty1 = pcrc_new_range_distribute(1, 100, pcrc_block,         &
     &                                      grp_p, 1)
```

```
      rng_ty2 = pcrc_new_range_collapse(1, 100)

! Define main arrays

      dad_x = pcrc_new_array_data(x, pcrc_real, pcrc_size_real,       &
     &                             2, grp_p)
      CALL pcrc_set_array_align(dad_x, 1, 1, 100, 2, 0, rng_tx1)
      CALL pcrc_set_array_align(dad_x, 2, 1, 100, 2, 0, rng_tx2)

      dad_y = pcrc_new_array_data(y, pcrc_real, pcrc_size_real,       &
     &                             2, grp_p)
      CALL pcrc_set_array_align(dad_y, 1, 1, 100, 1, 0, rng_ty1)
      CALL pcrc_set_array_align(dad_y, 2, 1, 100, 1, 0, rng_ty2)

! Do parallel loop  x(i, j) = 0

      CALL pcrc_loop_bounds(pcrc_rng(dad_x, 1), ll1, lu1, ls1)
      CALL pcrc_loop_bounds(pcrc_rng(dad_x, 2), ll2, lu2, ls2)
      DO i = ll1, lu1, ls1
        DO j = ll2, lu2, ls2
          x(i + 50 * j) = 0
        END DO
      END DO

! Do parallel loop  y(i, j) = i + j

      CALL pcrc_loop_bounds(pcrc_rng(dad_y, 1), ll1, lu1, ls1)
      CALL pcrc_loop_bounds(pcrc_rng(dad_y, 2), ll2, lu2, ls2)
      DO i = ll1, lu1, ls1
        DO j = ll2, lu2, ls2
          y(i + 25 * j) = pcrc_glob(pcrc_rng(dad_y, 1), i) +          &
     &                    pcrc_glob(pcrc_rng(dad_y, 2), j)
        END DO
      END DO


      rng_i = pcrc_new_range_align(26, 50, 1, 0, pcrc_rng(dad_x, 1))
      rng_j = pcrc_new_range_align(26, 50, 1, 0, pcrc_rng(dad_x, 2))

! Define and write temporary for remapped y(i + 1, j)

      dad_tmp1 = pcrc_new_array_data(tmp1, pcrc_real, pcrc_size_real, &
     &                                2, grp_p)
      CALL pcrc_set_array_copy(dad_tmp1, 1, rng_i)
      CALL pcrc_set_array_copy(dad_tmp1, 2, rng_j)

      dad_ys = pcrc_new_array_section(2, dad_y)
```

```
        CALL pcrc_set_array_triplet(dad_ys, 1, 27, 51, 1, dad_y, 1)
        CALL pcrc_set_array_triplet(dad_ys, 2, 26, 50, 1, dad_y, 2)

        CALL pcrc_remap(dad_tmp1, dad_ys)

        CALL pcrc_delete_array(dad_ys)

! Define and write temporary for remapped y(i, j + 1)

        dad_tmp2 = pcrc_new_array_data(tmp2, pcrc_real, pcrc_size_real, &
     &                                 2, grp_p)
        CALL pcrc_set_array_copy(dad_tmp2, 1, rng_i)
        CALL pcrc_set_array_copy(dad_tmp2, 2, rng_j)

        dad_ys = pcrc_new_array_section(2, dad_y)
        CALL pcrc_set_array_triplet(dad_ys, 1, 26, 50, 1, dad_y, 1)
        CALL pcrc_set_array_triplet(dad_ys, 2, 27, 51, 1, dad_y, 2)

        CALL pcrc_remap(dad_tmp2, dad_ys)

        CALL pcrc_delete_array(dad_ys)

! Do parallel loop  x(i, j) = y(i + 1, j) + y(i, j + 1)

        CALL pcrc_loop_bounds(rng_i, ll1, lu1, ls1)
        CALL pcrc_loop_bounds(rng_j, ll2, lu2, ls2)
        DO i = ll1, lu1, ls1
          DO j = ll2, lu2, ls2
            x(i + 50 * j) = tmp1(i + 50 * j) + tmp2(i + 50 * j)
          END DO
        END DO

        CALL pcrc_delete_array(dad_tmp2)
        CALL pcrc_delete_array(dad_tmp1)

        CALL pcrc_delete_range(rng_j)
        CALL pcrc_delete_range(rng_i)

! Do parallel loop  x(i, j) = 1

        rng_i = pcrc_new_range_loop(1, 100, 2, 1, 0, pcrc_rng(dad_x, 1))
        rng_j = pcrc_new_range_loop(1, 100, 2, 1, 0, pcrc_rng(dad_x, 2))

        CALL pcrc_loop_bounds(rng_i, ll1, lu1, ls1)
        CALL pcrc_loop_bounds(rng_j, ll2, lu2, ls2)
        DO i = ll1, lu1, ls1
          DO j = ll2, lu2, ls2
```

```
          IF(x(i + 50 * j) .NE. 0) x(i + 50 * j) = 1
        END DO
      END DO

      CALL pcrc_delete_range(rng_j)
      CALL pcrc_delete_range(rng_i)

! Define arguments and make call to 'foo'

      dad_xs = pcrc_new_array_section(1, dad_x)
      CALL pcrc_set_array_triplet(dad_xs, 1, 1, 100, 2, dad_x, 1)
      CALL pcrc_set_array_scalar(dad_xs, 1, dad_x, 2)

      CALL foo(x(50 * pcrc_loc(pcrc_rng(dad_x, 1), 1)), dad_xs)

      CALL pcrc_delete_array(dad_xs)

! PRINT *, (X(i,1),i=1,100,2)

      grp_s = pcrc_new_group_grid(0, shp_s)

      dad_tmp3 = pcrc_new_array_data(tmp3, pcrc_real, pcrc_size_real, &
    &                                1, grp_s)
      CALL pcrc_set_array_collapse(dad_tmp3, 1, 0, 49)

      dad_xs = pcrc_new_array_section(1, dad_x)
      CALL pcrc_set_array_triplet(dad_xs, 1, 1, 100, 2, dad_x, 1)
      CALL pcrc_set_array_scalar(dad_xs, 1, dad_x, 2)

      CALL pcrc_remap(dad_tmp3, dad_xs)

      CALL pcrc_delete_array(dad_xs)

      IF(pcrc_on(grp_s)) THEN
        PRINT *, (tmp3(i), i = 0, 49)
      ENDIF

      CALL pcrc_delete_array(dad_tmp3)

      CALL pcrc_delete_group(grp_s)

! Reclaim memory

      CALL pcrc_delete_array(dad_y)
      CALL pcrc_delete_array(dad_x)

      CALL pcrc_delete_range(rng_ty1)
```

```
        CALL pcrc_delete_range(rng_tx1)

        CALL pcrc_finalize()
      END


      SUBROUTINE foo(x, dad_x)
        IMPLICIT NONE

        REAL x(0 : *)
        INTEGER dad_x

        INCLUDE 'pcrc.inc'

        INTEGER xlb_actual, xub_actual, xstride_actual
        INTEGER dummy1, dummy2, dummy3

        INTEGER shp_p(1)
        INTEGER grp_p

        REAL y(0 : 12)
        INTEGER dad_y

        INTEGER dad_tmp1

        INTEGER bas_tmp1
        INTEGER str_x_1

        INTEGER i, ll, lu, ls

        CALL pcrc_reset_array_bounds(dad_x, 1, 1, 50, 1,                &
     &           xlb_actual, xub_actual, xstride_actual)

! Define processor arrangement

        shp_p(1) = 4
        grp_p = pcrc_new_group_grid(1, shp_p)

! Define main arrays

        dad_y = pcrc_new_array_data(y, pcrc_real, pcrc_size_real,       &
     &                              1, grp_p)
        CALL pcrc_set_array_distribute(dad_y, 1, 1, 50, pcrc_block,     &
     &                                 grp_p, 1)

! Do parallel loop  y(i) = 2
```

```
        CALL pcrc_loop_bounds(pcrc_rng(dad_y, 1), ll, lu, ls)
        DO i = ll, lu, ls
          y(i) = 2
        END DO

! Allocate and write temporary for remapped y(i)

        bas_tmp1 = pcrc_real_alloc(pcrc_size(dad_x))
        dad_tmp1 = pcrc_new_array_copy(pcrc_real_stack(bas_tmp1), dad_x)

        CALL pcrc_remap(dad_tmp1, dad_y)

! Do parallel loop  x(i) = y(i) + 1

        IF(pcrc_on(pcrc_grp(dad_x))) THEN
          str_x_1  = pcrc_str(dad_x, 1)
          CALL pcrc_loop_bounds(pcrc_rng(dad_x, 1), ll, lu, ls)
          DO i = ll, lu, ls
            x(i * str_x_1) = pcrc_real_stack(bas_tmp1 + i) + 1
          END DO
        END IF

        CALL pcrc_delete_array(dad_tmp1)
        CALL pcrc_real_free(bas_tmp1)

        CALL pcrc_delete_array(dad_y)

        CALL pcrc_delete_group(grp_p)

        CALL pcrc_reset_array_bounds(dad_x, 1,                          &
     &           xlb_actual, xub_actual, xstride_actual,               &
     &           dummy1, dummy2, dummy3)
      END
```