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

June 10, 1996

1 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
```

2 Processor grid

```
SUBROUTINE grid_init(rank, shape, grid)
  INTEGER rank
  INTEGER shape(rank)
  INTEGER grid
```

Creates a structure describing a processor grid of the specified rank and shape, and returns a handle to it in grid. Example: the HPF declaration

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

may yield the node fragments

```
INTEGER shp_p(1)
INTEGER grd_p
...

shp_p(1) = 4
CALL grid_init(1, shp_p, grd_p)
```

3 Range creation

```
INTEGER FUNCTION new_range_distribute(g_lb, g_ub,
                                      grid, dim, distribution)
  INTEGER g_lb, g_ub
  INTEGER grid
  INTEGER dim
  INTEGER distribution

INTEGER FUNCTION new_range_collapse(g_lb, g_ub)
  INTEGER g_lb, g_ub

INTEGER FUNCTION new_range_align(g_lb, g_ub, rng_parent,
                                 offset, stride)
  INTEGER rng_parent
  INTEGER g_lb, g_ub
  INTEGER lb, stride

INTEGER FUNCTION new_range_copy(rng_parent)
  INTEGER rng_parent

SUBROUTINE delete_range(rng)
  INTEGER rng
```

The `new_range_...` calls create "stand-alone" `DIM` objects and return integer handles to them. There are four variants which create variants with different distribution or alignment formats.

`DIM` structures normally exist inside array descriptors (see section **??**). Stand-alone versions are also convenient for describing HPF templates (which don't have any associated data) and ranges of parallel loops which don't happen to be identical the index ranges of some array in the program.

`g_lb`, `g_ub` are the index limits associated with the range. `new_range_distribute` creates a range distributed over a specified grid dimension with a specified distribution format. `new_range_collapse` creates a collapsed index range. `new_range_align` creates a range aligned to some parent range with some offset and stride. `new_range_copy` creates an identical copy of some parent range.

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

Example: the HPF fragment

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

could lead to the following fragments in the node program

```
      INTEGER shp_p(1)
      INTEGER grd_p
      INTEGER rng_tx1, rng_tx2
      ...

      shp_p(1) = 4
      CALL grid_init(1, shp_p, grd_p)

      rng_tx1 = new_range_distribute(1, 200, grd_p, 1, 1)
      rng_tx2 = new_range_collapse(1, 200)
```

The DIM structures referenced by rng_tx1, rng_tx2 now constitute a run-time representation of the template TX.

Example: the parallel loop in the HPF fragment

```
      REAL X(100, 100)
!HPF$ ALIGN WITH TX :: X
      ...

      FORALL (i=26:50,j=26:50) X(i,j) = ...
```

could yield the node code

```
     INTEGER rng_i, rng_j
     ...

     rng_i = new_range_align(26, 50, rng_tx1, 0, 1)
     rng_j = new_range_align(26, 50, rng_tx2, 0, 1)

     CALL loop_bounds(rng_i, ll1, lu1, ls1)
     CALL loop_bounds(rng_j, ll2, lu2, ls2)
     DO i = ll1, lu1, ls1
       DO j = ll2, lu2, ls2
         x(i, j) = ...
       END DO
     END DO

     CALL delete_range(rng_i)
     CALL delete_range(rng_i)
```

The loop_bounds operation will be described in section **??**. The range structures rng_i, rng_j will also be used to construct temporary arrays used for communications needed in translating the loop.

4 Array creation

```
  INTEGER FUNCTION new_array_data(array,
                                  element_type, element_size,
```

```
                                rank, majority, grp)
    choice  array(...)
    INTEGER element_type
    INTEGER element_size
    INTEGER rank
    INTEGER majority
    INTEGER grp

  SUBROUTINE set_array_range_distribute(dad, dim, g_lb, g_ub,
                                        grid, dim_grid,
                                        distribution)
    INTEGER dad
    INTEGER dim, dim_grid
    INTEGER g_lb, g_ub
    INTEGER grid
    INTEGER distribution

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

  SUBROUTINE set_array_range_align(dad, dim, g_lb, g_ub,
                                   rng_parent, lb, stride)
    INTEGER dad
    INTEGER dim
    INTEGER g_lb, g_ub
    INTEGER rng_parent
    INTEGER lb, stride

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

  SUBROUTINE set_array_data_done(dad)
    INTEGER dad


  SUBROUTINE delete_array(dad)
    INTEGER dad
```

The function `new_array_data` allocates a slot for a DAD and initialises the "per-array" 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, rank and in-processor layout of the distributed array are specified by the following arguments. The final argument should be a process group (see section ??). The simplest example of a process group is a processor grid.

The initialization of the new DAD is completed by making `rank` calls to an arbitrary combination of the `set_array_range_...` routines followed by one call to `set_array_data_done`. All of these calls reference the DAD under construction as their first argument. The `set_array_range_...` calls set the "per-dimension" fields of the DAD (the `DIM` structures). All dimensions of the array must be initialised—the second arguments of these calls must be some permutation of `1, ..., rank`. Each individual dimension of the array must be distributed over a distinct dimension of the process group defined in the `new_array_data` call, or it must be collapsed. This constraint must be respected whether the dimension is distributed directly through `set_array_range_distribute` or indirectly through `set_array_range_align` or `set_array_range_copy`.

The different variants of `set_array_range_...` allow for different distribution and alignment formats. See section ?? for further details—the four `set_array_range_...` calls are directly analogous to the `new_range_...` variants described in section ??.

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

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 grd_p and rng_tx1

      dad_x = new_array_data(x, 2, 4, 1, 1, grd_p)
      CALL set_array_range_align(dad_x, 1, 2, 99, rng_tx1, 1, 2)
      CALL set_array_data_done(dad_x)

      ... Use dad_x

      CALL delete_array(dad_x)
```

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(25, 100)
      INTEGER dad_x

      ... Define grd_p

      dad_x = new_array_data(x, 2, 4, 2, 1, grd_p)
      CALL set_array_range_distribute(dad_x, 1, 1, 100, grd_p, 1, 1)
      CALL set_array_range_collapse(dad_x, 2, 1, 100)
      CALL set_array_data_done(dad_x)

      ... Use dad_x

      CALL delete_array(dad_x)
```

5 Section creation

```
INTEGER FUNCTION new_array_section(dad_parent)
  INTEGER dad_parent

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

SUBROUTINE set_array_scalar(dad, dad_parent, dim_parent, i)
  INTEGER dad, dad_parent
  INTEGER dim_parent
  INTEGER i

SUBROUTINE set_array_section_done(dad)
  INTEGER dad
```

The function `new_array_section` allocates a slot for a DAD and initialises the "per-array" fields in an array descriptor describing a *section* of some parent array. It returns an integer handle. The argument is the descriptor for the parent array.

The initialization of the new DAD is completed by making r calls to an arbitrary combination of the `set_array_triplet` and `set_array_scalar` routines to define the section subscripts, followed by one call to `set_array_section_done`. 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 `set_array_triplet` calls set the "per-dimension" fields of the new descriptor (the `DIM` structures). 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, ie the number of calls to `set_array_triplet`. The last three arguments define the lower and upper bounds and stride of a triplet subscript into the parent array dimension.

The `set_array_scalar` calls affect the base address and process group fields (see section **??**) 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 = new_array_section(dad_x)
CALL set_array_triplet(dad_xs, 1, dad_x, 1, 1, 100, 2)
CALL set_array_scalar(dad_xs, dad_x, 2, 1)
CALL set_array_section_done(dad_xs)
```

6 Array dummy arguments

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

SUBROUTINE reset_array_range_lb(dad, dim, g_lb, g_lb_parent)
  INTEGER dad
  INTEGER dim
  INTEGER g_lb
```

In general, 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 `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 `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 of array subscripts. This bound generally changes every time an array is passed to a procedure. The routine `reset_array_range_lb` changes the lower bound for dimension `dim` to `g_lb`, modifying all fields in the DAD which depend on this value. For convenience, it returns the original value of the lower bound in the `g_lb_parent`.

`reset_array_range_lb` 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(*)
INTEGER dad
INTEGER ylb1_actual, ylb2_actual, dummy

CALL reset_array_range_lb(dad_y, 1, 1, ylb1_actual)
CALL reset_array_range_lb(dad_y, 2, 1, ylb2_actual)
...

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

CALL reset_array_range_lb(dad_y, 2, ylb2_actual, dummy)
CALL reset_array_range_lb(dad_y, 1, ylb1_actual, dummy)
END
```

7 Mapping inquiry functions

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

INTEGER FUNCTION grp(dad)
  INTEGER dad

LOGICAL FUNCTION on(grp)
  INTEGER grp
```

The inquiry function `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 *process group* 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.

For orientation we will describe a possible concrete representation of a process group. It consists of a vector of boolean variables and a vector of integers. The effective size of these vectors is the rank of the *process grid*.

The boolean field associated with a process grid dimension defines whether the associated dimension of the array was restricted by a scalar subscript. If it is false the integer field is undefined. If it is true, the integer field is the `sliceCoord` value. For an unsubscripted array the logical fields are all false and the integer fields are undefined. For a section constructed with a single scalar subscript (in a distributed dimension), one of the logical fields will be true and the corresponding integer field is the coordinate of the slice in the associated process dimension.

The inquiry function `grp` returns a handle to the process group for a given DAD. The function 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(*)
  INTEGER dad_y

  INTEGER ylb1_actual, dummy

  INTEGER str_y1
  INTEGER ll, lu, ls, i

  CALL reset_array_range_lb(dad_y, 1, 1, ylb1_actual)

  IF(on(grp(dad_y)) THEN
    str_y1 = str(dad_y, 1)
    loop_bounds(rng(dad_y, 1), ll, lu, ls)
    DO i = ll, lu, ls
      y(1 + (i - 1) * str_y1) = ...
    END DO
  END IF

  CALL reset_array_range_lb(dad_y, 1, ylb1_actual, dummy)
END
```

8 Stack allocation of temporary arrays

```
REAL real_stack(...)
...

INTEGER null

INTEGER FUNCTION reset_array_alloc(dad)
  INTEGER dad

SUBROUTINE reset_array_free(dad)
  INTEGER dad
```

The one dimensional array `real_stack` is visible in every procedure which includes `pcrc.inc`. Space for `REAL` arrays can be allocated from this vector. Similar vectors will be declared for each FORTRAN 77 primitive type (and in practice these differently-typed vectors may be equivalenced to a single global stack vector).

`null` is an integer of unspecified value.

The function `reset_array_alloc` allocates space for a temporary array on the stack. Its argument is a handle to a fully initialized array descriptor. The value of the data address field in this descriptor is overwritten with an address pointing into the stack. The function returns an index into the array *type*`_stack` identifying the same address. Here *type* is the primitive data type identified by the `element_type` argument of the `new_array_data` call used to initialize the DAD.

The function `reset_array_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.

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(*), y(*)
        INTEGER dad_x, dad_y

        INCLUDE 'pcrc.inc'

        INTEGER grp_x

        INTEGER dad_tmp1
        INTEGER bas_tmp1
        INTEGER str_x_1

        INTEGER i, ll, lu, ls

        ... Calls to reset_array_range_lb

        grp_x = grp(dad_x)

        dad_tmp1 = new_array_data(null, 2, 4, 1, 1, grp_x)
        CALL set_array_range_copy(dad_tmp1, 1, rng(dad_x, 1))
```

```
    CALL set_array_data_done(dad_tmp1)

    bas_tmp1 = reset_array_alloc(dad_tmp1)

    CALL remap(dad_tmp1, dad_y)

    IF(on(grp_x)) THEN
      str_x_1  = str(dad_x, 1)

      CALL loop_bounds(rng(dad_x, 1), ll, lu, ls)

      DO i = ll, lu, ls
        x(1 + (i - 1) * str_x_1) = x(1 + (i - 1) * str_x_1) +       &
&                                  real_stack(bas_tmp1 + i - 1)
      END DO
    END IF

    CALL reset_array_free(dad_tmp1)
    CALL delete_array(dad_tmp1)

    ... Calls to reset_array_range_lb
  END
```

The DAD for tmp1 is initially set up with the same parameters as x and a meaningless value (null) for its data address. A call to reset_array_alloc allocates a suitable portion of memory in real_stack. After copying the values in y to the temporary by the remap call (see section **??**), x is updated in terms of values in real_stack.

In the example tmp1 is one-dimensional and one could safely assume that its elements were contiguous in 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.

9 Address translation

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

INTEGER FUNCTION global_to_local(rng, g_i)
  INTEGER rng
  INTEGER g_i

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

```
  INTEGER l_lb, l_ub, l_stride
```

10 Communication

```
INTEGER FUNCTION 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

SUBROUTINE shift(dad, dad_source, dim, amount)
  INTEGER dad, dad_source
  INTEGER dim, amount

SUBROUTINE remap(dad, dad_source)
  INTEGER dad, dad_source
```

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
```

```
      PROGRAM main

        INCLUDE 'pcrc.inc'

        INTEGER shp_p(1)
        INTEGER grd_p

        INTEGER rng_tx1, rng_ty1

        REAL x(100), y(100)
        REAL tmp1(100), tmp2(100), tmp3(100), tmp4(100)

        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

        CALL pcrc_init()

! Define processor arrangement

        shp_p(1) = 4
        CALL grid_init(1, shp_p, grd_p)

! Define templates

        rng_tx1 = new_range_distribute(1, 400, grd_p, 1, 1)

        rng_ty1 = new_range_distribute(-200, 199, grd_p, 1, 1)

! Define main arrays

        dad_x = new_array_data(x, 2, 4, 1, 1, grd_p)
        CALL set_array_range_align(dad_x, 1, 2, 99, rng_tx1, 1, 2)
        CALL set_array_data_done(dad_x)

        dad_y = new_array_data(y, 2, 4, 1, 1, grd_p)
        CALL set_array_range_align(dad_y, 1, 1, 100, rng_ty1, -50, 3)
        CALL set_array_data_done(dad_y)

! Do parallel loop x(i) = 0

        CALL loop_bounds(rng(dad_x, 1), ll, lu, ls)
        DO i = ll, lu, ls
```

```
          x (i) = 0
        END DO

! Do parallel loop y(i) = i

        CALL loop_bounds(rng(dad_y, 1), ll, lu, ls)
        DO i = ll, lu, ls
          y (i) = local_to_global(rng(dad_y, 1), i)
        END DO

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

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

        IF(status1 .EQ. 1) THEN
          dad_tmp1 = new_array_data(tmp1, 2, 4, 1, 1, grd_p)
          CALL set_array_range_copy(dad_tmp1, 1, rng(dad_y, 1))
          CALL set_array_data_done(dad_tmp1)

          CALL shift(dad_tmp1, dad_y, 1, amount1)
        ELSE IF(status1 .EQ. 2) THEN
          dad_tmp3 = new_array_data(tmp3, 2, 4, 1, 1, grd_p)
          CALL set_array_range_copy(dad_tmp3, 1, rng(dad_x, 1))
          CALL set_array_data_done(dad_tmp3)

          dad_ys = new_array_section(dad_y)
          CALL set_array_triplet(dad_ys, 1, dad_y, 1, 3, 100, 1)
          CALL set_array_section_done(dad_ys)

          CALL remap(dad_tmp3, dad_ys)

          CALL delete_array(dad_ys)
        END IF

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

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

        IF(status2 .EQ. 1) THEN
          dad_tmp2 = new_array_data(tmp2, 2, 4, 1, 1, grd_p)
          CALL set_array_range_copy(dad_tmp2, 1, rng(dad_y, 1))
          CALL set_array_data_done(dad_tmp2)

          CALL shift(dad_tmp2, dad_y, 1, amount2)
        ELSE IF(status2 .EQ. 2) THEN
```

```
      dad_tmp4 = new_array_data(tmp4, 2, 4, 1, 1, grd_p)
      CALL set_array_range_copy(dad_tmp4, 1, rng(dad_x, 1))
      CALL set_array_data_done(dad_tmp4)

      dad_ys = new_array_section(dad_y)
      CALL set_array_triplet(dad_ys, 1, dad_y, 1, 1, 98, 1)
      CALL set_array_section_done(dad_ys)

      CALL remap(dad_tmp4, dad_ys)

      CALL delete_array(dad_ys)
    END IF

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

    CALL loop_bounds(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 delete_array(dad_tmp1)
    ELSE IF(status1 .EQ. 2) THEN
      CALL delete_array(dad_tmp3)
    END IF
```

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

! Reclaim memory

        CALL delete_array(dad_y)
        CALL delete_array(dad_x)

        CALL delete_range(rng_ty1)
        CALL 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
```

```
      PROGRAM main

        INCLUDE 'pcrc.inc'

        INTEGER shp_p(1)
        INTEGER grd_p

        INTEGER rng_tx1, rng_ty1

        REAL x(100), y(100)
        REAL tmp1(100), tmp2(100), tmp3(100), tmp4(100)

        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

        CALL pcrc_init()

! Define processor arrangement

        shp_p(1) = 4
        CALL grid_init(1, shp_p, grd_p)

! Define template

        rng_tx1 = new_range_distribute(1, 400, grd_p, 1, 1)
        rng_ty1 = new_range_distribute(-200, 199, grd_p, 1, 1)

! Define main arrays

        dad_x = new_array_data(x, 2, 4, 1, 1, grd_p)
        CALL set_array_range_align(dad_x, 1, 2, 99, rng_tx1, 1, 2)
        CALL set_array_data_done(dad_x)

        dad_y = new_array_data(y, 2, 4, 1, 1, grd_p)
        CALL set_array_range_align(dad_y, 1, 1, 100, rng_ty1, -50, 3)
        CALL set_array_data_done(dad_y)

! Do parallel loop x(i) = 0

        CALL loop_bounds(rng(dad_x, 1), ll, lu, ls)
        DO i = ll, lu, ls
          x(i) = 0
```

```
        END DO

! Do parallel loop y(i) = i

        CALL loop_bounds(rng(dad_y, 1), ll, lu, ls)
        DO i = ll, lu, ls
          y(i) = local_to_global(rng(dad_y, 1), i)
        END DO

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

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

        IF(status1 .EQ. 1) THEN
          dad_tmp1 = new_array_data(tmp1, 2, 4, 1, 1, grd_p)
          CALL set_array_range_copy(dad_tmp1, 1, rng(dad_y, 1))
          CALL set_array_data_done(dad_tmp1)

          CALL shift(dad_tmp1, dad_y, 1, amount1)
        ELSE IF(status1 .EQ. 2) THEN
          dad_tmp3 = new_array_data(tmp3, 2, 4, 1, 1, grd_p)
          CALL set_array_range_copy(dad_tmp3, 1, rng(dad_x, 1))
          CALL set_array_data_done(dad_tmp3)

          dad_ys = new_array_section(dad_y)
          CALL set_array_triplet(dad_ys, 1, dad_y, 1, 3, 100, 1)
          CALL set_array_section_done(dad_ys)

          CALL remap(dad_tmp3, dad_ys)
        END IF

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

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

        IF(status2 .EQ. 1) THEN
          dad_tmp2 = new_array_data(tmp2, 2, 4, 1, 1, grd_p)
          CALL set_array_range_copy(dad_tmp2, 1, rng(dad_y, 1))
          CALL set_array_data_done(dad_tmp2)

          CALL shift(dad_tmp2, dad_y, 1, amount2)

          CALL delete_array(dad_ys)
        ELSE IF(status2 .EQ. 2) THEN
          dad_tmp4 = new_array_data(tmp4, 2, 4, 1, 1, grd_p)
```

```
      CALL set_array_range_copy(dad_tmp4, 1, rng(dad_x, 1))
      CALL set_array_data_done(dad_tmp4)

      dad_ys = new_array_section(dad_y)
      CALL set_array_triplet(dad_ys, 1, dad_y, 1, 1, 98, 1)
      CALL set_array_section_done(dad_ys)

      CALL remap(dad_tmp4, dad_ys)

      CALL delete_array(dad_ys)
    END IF

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

    CALL loop_bounds(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 delete_array(dad_tmp1)
    ELSE IF(status1 .EQ. 2) THEN
      CALL delete_array(dad_tmp3)
    END IF

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

```
          CALL delete_array(dad_tmp2)
        ELSE IF(status1 .EQ. 2) THEN
          CALL delete_array(dad_tmp4)
        END IF

! Define arguments and make call to 'foo'

        dad_xs = new_array_section(dad_x)
        CALL set_array_triplet(dad_xs, 1, dad_x, 1, 2, 98, 2)
        CALL set_array_section_done(dad_xs)

        dad_ys = new_array_section(dad_y)
        CALL set_array_triplet(dad_ys, 1, dad_y, 1, 1, 99, 2)
        CALL set_array_section_done(dad_ys)

        CALL foo(x, dad_xs, y, dad_ys)

        CALL delete_array(dad_ys)
        CALL delete_array(dad_xs)

! Define arguments and make call to 'foo'

        dad_xs = new_array_section(dad_x)
        CALL set_array_triplet(dad_xs, 1, dad_x, 1, 3, 99, 2)
        CALL set_array_section_done(dad_xs)

        dad_ys = new_array_section(dad_y)
        CALL set_array_triplet(dad_ys, 1, dad_y, 1, 2, 100, 2)
        CALL set_array_section_done(dad_ys)

        CALL foo(x, dad_xs, y, dad_ys)

        CALL delete_array(dad_ys)
        CALL delete_array(dad_xs)

! Reclaim memory

        CALL delete_array(dad_y)
        CALL delete_array(dad_x)

        CALL delete_range(rng_ty1)
        CALL delete_range(rng_tx1)

        CALL pcrc_finalize()
      END
```

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

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

        INCLUDE 'pcrc.inc'

        INTEGER xlb_actual, ylb_actual, dummy

        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 reset_array_range_lb(dad_x, 1, 1, xlb_actual)
        CALL reset_array_range_lb(dad_y, 1, 1, ylb_actual)

        grp_x = grp(dad_x)

! Allocate and write temporary for remapped y(i)

        dad_tmp1 = new_array_data(null, 2, 4, 1, 1, grp_x)
        CALL set_array_range_copy(dad_tmp1, 1, rng(dad_x, 1))
        CALL set_array_data_done(dad_tmp1)

        bas_tmp1 = reset_array_alloc(dad_tmp1)

        dad_ys = new_array_section(dad_y)
        CALL set_array_triplet(dad_ys, 1, dad_y, 1, 1, 49, 1)
        CALL set_array_section_done(dad_ys)

        CALL remap(dad_tmp1, dad_ys)

        CALL delete_array(dad_ys)

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

        dad_tmp2 = new_array_data(null, 2, 4, 1, 1, grp_x)
        CALL set_array_range_copy(dad_tmp2, 1, rng(dad_x, 1))
        CALL set_array_data_done(dad_tmp2)

        bas_tmp2 = reset_array_alloc(dad_tmp2)
```

```
      dad_ys = new_array_section(dad_y)
      CALL set_array_triplet(dad_ys, 1, dad_y, 1, 2, 50, 1)
      CALL set_array_section_done(dad_ys)

      CALL remap(dad_tmp2, dad_ys)

      CALL delete_array(dad_ys)

! Do parallel loop

      IF(on(grp_x)) THEN
        str_x_1  = str(dad_x, 1)

        CALL loop_bounds(rng(dad_x, 1), ll, lu, ls)

        DO i = ll, lu, ls
          x(1 + (i - 1) * str_x_1) = x(1 + (i - 1) * str_x_1) -       &
   &                                 real_stack(bas_tmp1 + i - 1) -   &
   &                                 real_stack(bas_tmp2 + i - 1)
        END DO
      END IF

! Reclaim memory

      CALL reset_array_free(dad_tmp2)
      CALL delete_array(dad_tmp2)

      CALL reset_array_free(dad_tmp1)
      CALL delete_array(dad_tmp1)

      CALL reset_array_range_lb(dad_y, 1, ylb_actual, dummy)
      CALL reset_array_range_lb(dad_x, 1, xlb_actual, dummy)
    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
```

```
      PROGRAM main

        INCLUDE 'pcrc.inc'

        INTEGER shp_p(1)
        INTEGER grd_p

        INTEGER rng_tx1, rng_tx2, rng_ty1, rng_ty2

        REAL x(25, 100), y(25, 100)

        INTEGER dad_x, dad_y

        INTEGER dad_ys, dad_xs

        INTEGER dad_tmp1, dad_tmp2

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

        CALL pcrc_init()

! Define processor arrangement

        shp_p(1) = 4
        CALL grid_init(1, shp_p, grd_p)

! Define templates

        rng_tx1 = new_range_distribute(1, 200, grd_p, 1, 1)
        rng_tx2 = new_range_collapse(1, 200)

        rng_ty1 = new_range_distribute(1, 100, grd_p, 1, 1)
        rng_ty2 = new_range_collapse(1, 100)

! Define main arrays

        dad_x = new_array_data(x, 2, 4, 2, 1, grp_p)
        CALL set_array_range_align(dad_x, 1, 1, 100, rng_tx1, 0, 2)
        CALL set_array_range_align(dad_x, 2, 1, 100, rng_tx2, 0, 2)
        CALL set_array_data_done(dad_x)

        dad_y = new_array_data(y, 2, 4, 2, 1, grp_p)
        CALL set_array_range_align(dad_y, 1, 1, 100, rng_ty1, 0, 1)
        CALL set_array_range_align(dad_y, 2, 1, 100, rng_ty2, 0, 1)
        CALL set_array_data_done(dad_y)

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

```
       CALL loop_bounds(rng(dad_x, 1), ll1, lu1, ls1)
       CALL loop_bounds(rng(dad_x, 2), ll2, lu2, ls2)
       DO i = ll1, lu1, ls1
         DO j = ll2, lu2, ls2
           x(i, j) = 0
         END
       END

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

       CALL loop_bounds(rng(dad_y, 1), ll1, lu1, ls1)
       CALL loop_bounds(rng(dad_y, 2), ll2, lu2, ls2)
       DO i = ll1, lu1, ls1
         DO j = ll2, lu2, ls2
           y(i, j) = local_to_global(rng(dad_y, 1), i) +                &
     &                local_to_global(rng(dad_y, 2), j)
         END
       END

       rng_i = new_range_align(1, 25, rng(dad_x, 1), 25, 1)
       rng_j = new_range_align(1, 25, rng(dad_x, 2), 25, 1)

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

       dad_tmp1 = new_array_data(tmp1, 2, 4, 1, 1, grp_p)
       CALL set_array_range_copy(dad_tmp1, 1, rng_i)
       CALL set_array_range_copy(dad_tmp1, 2, rng_j)
       CALL set_array_data_done(dad_tmp1)

       dad_ys = new_array_section(dad_y)
       CALL set_array_triplet(dad_ys, 1, dad_y, 1, 27, 51, 1)
       CALL set_array_triplet(dad_ys, 2, dad_y, 2, 26, 50, 1)
       CALL set_array_section_done(dad_ys)

       CALL remap(dad_tmp1, dad_ys)

       CALL delete(dad_ys)

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

       dad_tmp2 = new_array_data(tmp2, 2, 4, 1, 1, grp_p)
       CALL set_array_range_copy(dad_tmp2, 1, rng_i)
       CALL set_array_range_copy(dad_tmp2, 1, rng_j)
       CALL set_array_data_done(dad_tmp2)

       dad_ys = new_array_section(dad_y)
```

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

        CALL remap(dad_tmp2, dad_ys)

        CALL delete(dad_ys)

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

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

        CALL delete_range(rng_i)
        CALL delete_range(rng_i)

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

        rng_i = new_range_align(1, 50, rng(dad_x, 1), -1, 2)
        rng_j = new_range_align(1, 50, rng(dad_x, 2), -1, 2)

        CALL loop_bounds(rng_i, ll1, lu1, ls1)
        CALL loop_bounds(rng_j, ll2, lu2, ls2)
        DO i = ll1, lu1, ls1
          DO j = ll2, lu2, ls2
            IF(x(i, j) .NE. 0) x(i, j) = 1
          END DO
        END DO

        CALL delete_range(rng_i)
        CALL delete_range(rng_i)

        CALL delete_array(tmp2)
        CALL delete_array(tmp1)

! Define arguments and make call to 'foo'

        dad_xs = new_array_section(dad_x)
        CALL set_array_triplet(dad_xs, 1, dad_x, 1, 1, 100, 2)
        CALL set_array_scalar(dad_xs, dad_x, 2, 1)
        CALL set_array_section_done(dad_xs)
```

```
        CALL foo(x(global_to_local(rng(dad_x, 1), 1), 1), dad_xs)

        CALL delete(dad_xs)

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

        CALL delete_array(dad_y)
        CALL delete_array(dad_x)

        CALL delete_range(rng_ty1)
        CALL delete_range(rng_tx1)

        CALL pcrc_finalize()
      END


      SUBROUTINE foo(x, dad_x)
        IMPLICIT NONE

        REAL x(*)
        INTEGER dad_x

        INCLUDE 'pcrc.inc'

        INTEGER xlb_actual, dummy

        INTEGER shp_p(1)
        INTEGER grd_p

        REAL y(13)
        INTEGER dad_y

        INTEGER dad_tmp1

        INTEGER bas_tmp1

        CALL reset_array_range_lb(dad_x, 1, 1, xlb_actual)

! Define processor arrangement

        shp_p(1) = 4
        CALL grid_init(1, shp_p, grd_p)

! Define main arrays

        dad_y = new_array_data(y, 2, 4, 2, 1, grp_p)
        CALL set_array_range_distribute(dad_y, 1, 1, 50, grd_p, 1, 1)
```

```
        CALL set_array_data_done(dad_y)

! Do parallel loop  y(i) = 2

        CALL loop_bounds(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)

        dad_tmp1 = new_array_data(null, 2, 4, 1, 1, grp(dad_x))
        CALL set_array_range_copy(dad_tmp1, 1, rng(dad_x, 1))
        CALL set_array_data_done(dad_tmp1)

        bas_tmp1 = reset_array_alloc(dad_tmp1)

        CALL remap(dad_tmp1, dad_y)

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

        IF(on(grp(dad_x)) THEN
          str_x_1  = str(dad_x, 1)

          CALL loop_bounds(rng(dad_x, 1), ll, lu, ls)
          DO i = ll, lu, ls
            x(1 + (i - 1) * str_x_1) = real_stack(bas_tmp1 + i - 1) + 1
          END DO
        END IF

        CALL reset_array_free(dad_tmp1)
        CALL delete_array(dad_tmp1)

        CALL delete_array(dad_y)

        CALL reset_array_range_lb(dad_x, 1, xlb_actual, dummy)
      END
```