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 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 shift and remap operations of section 11.

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 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 class is 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 rôle—it describes a slice of a process grid.

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

\begin{verbatim}
  INTEGER FUNCTION foo(x)
  REAL x
\end{verbatim}

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

\(^1\)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.
\[ n = \text{foo}(1.0 + z) \]

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

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

## 2 Global state

\begin{verbatim}
INCLUDE 'pcrc.inc'

SUBROUTINE pcrc_init()

SUBROUTINE pcrc_finalize()
\end{verbatim}

The file \texttt{pcrc.inc} includes declarations which must be visible in every procedure that calls the library.

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

Example: a typical node program has the form

\begin{verbatim}
PROGRAM main

INCLUDE 'pcrc.inc'

... Declarations

CALL pcrc_init()

... Executable statements

CALL pcrc_finalize()
END
\end{verbatim}

## 3 Processor grid

\begin{verbatim}
INTEGER FUNCTION new_grid(rank, shape)
    INTEGER rank
    INTEGER shape(rank)

SUBROUTINE delete_grid(grp)
    INTEGER grp
\end{verbatim}
new_grid creates a 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

```fortran
INTEGER shape_p(1)
INTEGER grp_p
...
shape_p(1) = 4
grp_p = new_grid(1, shape_p)
```

Any grid structure created with the new_grid call may be destroyed with a delete_grid call if it is no longer needed.

The structure returned by new_grid is a group structure. A group structure can describe a process grid. It can also describe a “slice” of a process grid. This will be discussed further in section 8.

4 Range creation

```fortran
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, 
   offset, stride, rng_parent)
   INTEGER rng_parent
   INTEGER g_lb, g_ub
   INTEGER lb, stride

INTEGER FUNCTION new_range_copy(rng_parent)
   INTEGER rng_parent

INTEGER FUNCTION new_range_loop(g_lb, g_ub, g_stride, 
   offset, stride, rng_parent)
   INTEGER rng_parent
   INTEGER g_lb, g_ub, g_stride,
```
INTEGER lb, stride

SUBROUTINE delete_range(rng)
  INTEGER rng

The 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 for a few purposes. 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.

new_range_distribute creates a range distributed over a specified grid dimension with a specified distribution format. new_rangeCollapse creates a collapsed index range.

Example: the HPF fragment

!HPF$ PROCESSES 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 = new_grid(1, shp_p)

    rng_t1 = new_range_distribute(1, 200, grp_p, 1, 1)  
    rng_t2 = new_rangeCollapse(1, 200)

The DIM structures referenced by rng_t1, rng_t2 constitute a run-time representation of the template T. The final argument of new_range_distribute is 1 for a block distribution or 2 for a cyclic distribution.

new_range_align creates a range aligned to some parent range with some offset and stride. The element of the new range with global index i is aligned to the element of the parent range with global index offset + base * i.

new_range_copy creates an identical copy of some parent range. These two
operations are provided for completeness: they correspond to `set_array_range`
variants for creating aligned arrays, introduced in the next section.

`new_range_loop` is similar to `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

```fortran
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

```fortran
INTEGER rng_i, rng_j
...
rng_i = new_range_loop(25, 50, 2, 0, 2, rng_t1)
rng_j = new_range_loop(50, 100, 4, 50, 1, rng_t2)
...
```

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 `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 `new_range...` call should be destroyed
with a `delete_range` call when they are no longer needed.

## 5 Array creation

```fortran
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_rangeCollapse(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, 
offset, stride, rng_parent)
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_range_loop(dad, dim, g_lb, g_ub, g_stride, 
offset, stride, rng_parent)
INTEGER dad
INTEGER dim
INTEGER g_lb, g_ub
INTEGER rng_parent
INTEGER lb, stride

SUBROUTINE set_array_data_done(dad)
INTEGER dad

INTEGER FUNCTION new_array_copy(array, dad_parent)
choice array(...)
INTEGER dad
SUBROUTINE delete_array(dad)
INTEGER dad

The function new_array_data allocates a slot for a DAD, 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 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 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 range structures in the DAD. All dimensions of the array must be initialised—the dim 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 grid 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, set_array_range_copy or set_array_range_loop.

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

Example: the HPF declarations

!HPFS PROCESSORS P(4)
!HPFS TEMPLATE TX(400)
!HPFS DISTRIBUTED TX(BLOCK) ONTO P

REAL X(2:99)
!HPFS 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 = new_array_data(x, 2, 4, 1, 1, grp_p)
CALL set_array_range_align(dad_x, 1, 2, 99, 1, 2, rng_tx1)
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$ DISTRIBUT X(BLOCK,*) ONTO P

could translate to the node fragments

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

... Define grp_p

dad_x = new_array_data(x, 2, 4, 2, 1, grp_p)
CALL set_array_range_distribute(dad_x, 1, 1, 100, grp_p, 1, 1)
CALL set_array_rangeCollapse(dad_x, 2, 1, 100)
CALL set_array_data_done(dad_x)

... Use dad_x

CALL 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 = new_array_data(x, 2, 4, 2, 1, grp(dad_y))
CALL set_array_range_align(dad_x, 1, 1, 100, 0, 2, rng(y, 1))
CALL set_array_range_copy(dad_x, 2, rng(y, 2))
CALL set_array_data_done(dad_x)

The call

dad = new_array_copy(array, dad_parent)

makes a DAD which is identical to dad_parent except that the local array segment is replaced with array. It is equivalent to
dad = new_array_data(array,
    element_type, element_size, rank, majority,
    grp(dad_parent))
set_array_range_copy(dad, 1, rng(dad_parent, 1))
...
set_array_range_copy(dad, rank, rng(dad_parent, rank))
set_array_data_done(dad)

where the parameters element_type, element_size, rank and majority refer to the parent array (these parameters are always known at compile time). The functions grp and rng will be introduced in section 8.

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

6 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, ..., 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, i.e. 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 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_x`s for this section is constructed from the descriptor `dad_x` of `X` by the sequence of calls

\[
\begin{align*}
dad_x &= \text{new_array_section}(dad_x) \\
\text{CALL } & \text{set_array_triplet}(dad_x, 1, dad_x, 1, 1, 100, 2) \\
\text{CALL } & \text{set_array_scalar}(dad_x, dad_x, 2, 1) \\
\text{CALL } & \text{set_array_section_done}(dad_x)
\end{align*}
\]

### 7 Array dummy arguments

```fortran
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

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

CALL foo(x(6, :, :))
```
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 \( \text{reset_array_range}_lb \) changes the lower bound for dimension \( \text{dim} \) to \( \text{g}_lb \), modifying all fields in the DAD which depend on this value. For convenience, it returns the original value of the lower bound in the \( \text{g}_lb\_\text{parent} \).

\( \text{reset_array_range}_lb \) 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 \( \text{foo} \) could be

```fortran
SUBROUTINE foo(y, dad_y)
REAL y(10, 10)
...
\( y(i, j) = \ldots \)
END
```

8 Mapping inquiry functions

```fortran
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 `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.

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 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 inherited 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 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(i * str_y1) = ...  
    END DO
  END IF
  CALL reset_array_range_lb(dad_y, 1, ylb1_actual, dummy)
END

9 Stack allocation of temporary arrays

REAL real_stack(...)
...

INTEGER FUNCTION real_alloc(size)
  INTEGER size
...

SUBROUTINE real_free(base)
  INTEGER base
...

INTEGER FUNCTION size(dad)
  INTEGER dad

SUBROUTINE array_reset_base(dad, array)
  INTEGER dad
  choice array(...)

The one dimensional array real_stack is visible in every procedure which includes pcr.inc. Space for REAL arrays can be allocated from this vector.
The function 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 real_stack identifying the base address of the vector.

The function real_free will free the space allocated by the matching allo-
cation 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 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 reset_array_range_lb
    grp_x = grp(dad_x)
    bas_tmp1 = real_alloc(size(dad_x))
    dad_tmp1 = new_array_copy(real_stack(bas_tmp1), dad_x)
    CALL remap(dad_tmp1, dad_y)
    IF(on(grp(dad_x))) THEN

15
str_x_1 = str(dad_x, 1)

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

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

CALL delete_array(dad_tmp1)
CALL real_free(bas_tmp1)

... Calls to reset_array_range_lb

END

After copying the values in y to the stack-allocated temporary by the remap call
(see section 11), 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.

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 size inquiry. In general
a new DAD is constructed before calling 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 array_reset_base. For example

dad_x = new_array_data(real_stack, 2, 4, 1, 1, grp_p)
... set range(s)
CALL set_array_data_done(dad_x)

bas_x = real_alloc(size(dad_x))
reset_array_base(dad_x, 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 deallo-
cation 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 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

The routines local_to_global and global_to_local translate between 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 seg-
ment 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 pro-
gram must explicitly include any memory-stride multipliers required to flatten
array segments).

The function loop_bounds returns the bounds of a DO loop needed to enu-
merate 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 = new_range_loop(25, 50, 2, 0, 2, rng_t1)
    rng_j = new_range_loop(50, 100, 4, 50, 1, rng_t2)
    loop_bounds(rng_i, i_lb, i_ub, i_stride)
    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) = ...

    17
11 Communication

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

SUBROUTINE remap(dad, dad_source)
    INTEGER dad, dad_source

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

The 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 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.

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

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

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

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

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

A.1 Notes on the translation

Communications are optimised by calling `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 shift operation. If the returned status is 2, a simple shift is inadequate, and the general 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

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_y1

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

CALL pcrc_init()

! Define processor arrangement

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

! Define templates

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

! Define main arrays

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

  dad_y = new_array_data(y, 2, 4, 1, 1, grp_p)
  CALL set_array_range_align(dad_y, 1, 1, 100, -50, 3, rng_ty1)
  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, &
& u1, v1, amount1)

IF(status1 .EQ. 1) THEN
  dad_tmp1 = new_array_copy(tmp1, dad_y)
  CALL shift(dad_tmp1, dad_y, 1, amount1)
ELSE IF(status1 .EQ. 2) THEN
  dad_tmp3 = new_array_copy(tmp3, dad_x)
  dad_y = new_array_section(dad_y)
  CALL set_array_triplet(dad_y, 1, dad_y, 1, 3, 100, 1)
  CALL set_array_section_done(dad_y)
  CALL remap(dad_tmp3, dad_y)
  CALL delete_array(dad_y)
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_copy(tmp2, dad_y)
  CALL shift(dad_tmp2, dad_y, 1, amount2)
ELSE IF(status2 .EQ. 2) THEN
  dad_tmp4 = new_array_copy(tmp4, dad_x)
  dad_y = new_array_section(dad_y)
  CALL set_array_triplet(dad_y, 1, dad_y, 1, 1, 98, 1)
  CALL set_array_section_done(dad_y)
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(ii) + y(i2)
   ELSE IF(status1 .EQ. 0 .AND. status2 .EQ. 1) THEN
      x(i) = y(ii) + tmp2(i2)
   ELSE IF(status1 .EQ. 0 .AND. status2 .EQ. 2) THEN
      x(i) = y(ii) + 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_y1)
CALL delete_range(rng_x1)

CALL pcrc_finalize()
END
B Level 2 Template

PROGRAM MAIN
REAL X(2:99), Y(100)
!HPFS PROCESSORS P(4)
!HPFS TEMPLATE TX(400), TY(-200:199)
!HPFS DISTRIBUTION TX(BLOCK) ONTO P
!HPFS DISTRIBUTION TY(BLOCK) ONTO P
!HPFS ALIGN X(i) WITH TX(2*i+1)
!HPFS 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)
!HPFS 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.

DADs 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 real_stack. They are aligned with x using new_array_copy.

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

INCLUDE 'pcrc.inc'

INTEGER shp_p(1)
INTEGER grp_p

INTEGER rng_tx1, rng_ty1

REAL x(0 : 99), y(0 : 99)
REAL tsp1(0 : 99), tsp2(0 : 99), tsp3(0 : 99), tsp4(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 ii, i2
REAL uf, vi, u2, v2

CALL pcrc_init()

! Define processor arrangement

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

! Define template

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

! Define main arrays

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

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

! Do parallel loop x(i) = 0

CALL loop_bounds(rng(dad_x, i), 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), il, lu, ls)
DO i = il, 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, &
   u1, v1, amount1)

IF(status1 .EQ. 1) THEN
   dad_tmp1 = new_array_copy(tmp1, dad)

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

   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_copy(tmp2, dad_y)

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

   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)
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)

27
CALL set_array_triplet(dad_xs, 1, dad_x, 1, 2, 98, 2)
CALL set_array_section_done(dad_xs)

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

CALL foo(x, dad_xs, y, dad_y)
CALL delete_array(dad_y)
CALL delete_array(dad_xs)

! Define arguments and make call to ‘foo’

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

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

CALL foo(x, dad_x, y, dad_y)
CALL delete_array(dad_y)
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(0 : *), y(0 : *)
INTEGER dad_x, dad_y

INCLUDE 'pcrc.inc'
INTEGER xlb_actual, ylb_actual, dummy

INTEGER grp_x

INTEGER dad_tmp1, dad_tmp2, dad_y,
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)

bas_tmp1 = real_alloc(size(dad_x))
dad_tmp1 = new_array_copy(real_stack(bas_tmp1), dad_x)

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

CALL remap(dad_tmp1, dad_y)

CALL delete_array(dad_y)

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

bas_tmp2 = real_alloc(size(dad_x))
dad_tmp2 = new_array_copy(real_stack(bas_tmp2), dad_x)

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

CALL remap(dad_tmp2, dad_y)

CALL delete_array(dad_y)

! Do parallel loop

IF(on(grp_x)) THEN

str_x_1 = str(dad_x, i)

29
CALL loop_bounds(rng(dad_x, 1), ll, lu, ls)

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

! Reclaim memory

CALL delete_array(dad_tmp2)
CALL real_free(bas_tmp2)
CALL reset_array_free(dad_tmp1)
CALL real_free(bas_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$ DISTRIBUT X(BLOCK,* ) ONTO P
!HPF$ DISTRIBUT TY(BLOCK,*) ONTO P
!HPF$ ALIGN X(i,j) WITH TX(2*i,2*j)
!HPF$ ALIGN Y(i,j) WITH TY(i,j)

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

FOR ALL (i=26:50, j=26:50) X(i,j) = Y(i+1,j) + Y(i,j+1)
FOR ALL (i=1:100:2, j=1:N,2,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$ DISTRIBUT Y(BLOCK) ONTO P

FOR ALL (i=1:50) Y(i) = 2
FOR ALL (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 remap. Alternatively 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 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

INCLUDE 'pcrc.inc'

INTEGER shp_p(1)
INTEGER grp_p

INTEGER rng_tx1, rng_tx2, rng_ty1, rng_ty2
REAL x(0 : 2499), y(0 : 2499)
INTEGER dad_x, dad_y
INTEGER dad_yx, dad_y

REAL tmp1(0 : 2499), tmp2(0 : 2499)
INTEGER dad_tmp1, dad_tmp2
INTEGER i, j, l11, l1, l21, l12, l2, l22

CALL pcrc_init()

! Define processor arrangement

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

! Define templates

    rng_tx1 = new_range_distribute(1, 200, grp_p, 1, 1)
    rng_tx2 = new_range Collapse(1, 200)

    rng_ty1 = new_range_distribute(1, 100, grp_p, 1, 1)
    rng_ty2 = new_range_colla(pse(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, 0, 2, rng tx1)
    CALL set_array_range_align(dad_x, 2, 1, 100, 0, 2, rng tx2)
    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, 0, 1, rng ty1)
    CALL set_array_range_align(dad_y, 2, 1, 100, 0, 1, rng ty2)
    CALL set_array_data_done(dad_y)

33
! Do parallel loop  x(i, j) = 0

CALL loop_bounds(rng(dad_x, 1), l1l, l1u, lsi)
CALL loop_bounds(rng(dad_x, 2), l12, l1u2, lsi)
DO i = l1l, l1u, lsi
  DO j = l12, l1u2, lsi2
    x(i + 25 * j) = 0
  END
END

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

CALL loop_bounds(rng(dad_y, 1), l1l, l1u, lsi)
CALL loop_bounds(rng(dad_y, 2), l12, l1u2, lsi)
DO i = l1l, l1u, lsi
  DO j = l12, l1u2, lsi2
    y(i + 25 * j) = local_to_global(rng(dad_y, 1), i) + &
    local_to_global(rng(dad_y, 2), j)
  END
END

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

! 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, 111, lu1, ls1)
CALL loop_bounds(rng_j, 112, lu2, ls2)
DO i = 111, lu1, ls1
   DO j = 112, lu2, ls2
      x(i + 25 * j) = tmp1(i + 25 * j) + tmp2(i + 25 * j)
   END DO
END DO

CALL delete_range(rng_i)
CALL delete_range(rng_i)

! Do parallel loop  x(i, j) = 1
rang_i = new_range_align(1, 50, -1, 2, rng(dad x, 1))
rang_j = new_range_align(1, 50, -1, 2, rng(dad x, 2))

CALL loop_bounds(rng_i, 111, lu1, ls1)
CALL loop_bounds(rng_j, 112, lu2, ls2)
DO i = 111, lu1, ls1
   DO j = 112, lu2, ls2
      IF(x(i + 25 * j) .NE. 0) x(i + 25 * 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)), 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_tz1)

CALL pcrc_finalize()
END

SUBROUTINE foo(x, dad_x)
IMPLICIT NONE

REAL x(0 : *)
INTEGER dad_x

INCLUDE 'pcrc.inc'

INTEGER xlb_actual, dummy

INTEGER shp_p(1)
INTEGER grp_p

REAL y(0 : 12)
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
grp_p = new_grid(1, shp_p)

! Define main arrays
```
dad_y = new_array_data(y, 2, 4, 2, i, grp_p)
CALL set_array_range_distribute(dad_y, 1, 1, 50, grp_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)

bas_tmp1 = real_alloc(size(dad_x))
dad_tmp1 = new_array_copy(real_stack(bas_tmp1), dad_x)

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

CALL delete_array(dad_tmp1)
CALL real_free(bas_tmp1)

CALL delete_array(dad_y)

CALL delete_grid(grp_p)

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