Proposals for the C Interface

D.B. Carpenter

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

May 31, 1996

The semantics of the interface presented here would be identical to the FORTRAN interface, except for the handling of index ranges. In this draft we have adopted the convention that all subscript ranges start at 0, and ranges are parametrized by their extents rather than by `lb`, `ub` *pairs.*

Arguably, this is a natural choice here, if we assume that the C interface would typically be used to translate data-parallel C dialects (because, unlike FORTRAN 77, C subscripts always start at 0).

1 Global state

```
#include "pcrc.h"

void pcrcInit(int argc, char* argv []) ;

void pcrcFinalize() ;
```

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

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

Example: a typical node program has the form

```
#include pcrc.h

void main(int argc, char* argv []) {

  ... Declarations

  pcrcInit() ;

  ... Executable statements

  pcrcFinalize() ;
}
```

2 Processor grid

```
void gridInit(int rank, int shape [], GRP* 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

```
int shp_p [1] ;
GRP grd_p ;
...

shp_p [0] = 4 ;
gridInit(1, shp_p, &grd_p) ;
```

3 Range initialization

```
void setRangeDistribute(DIM* rng, int extent,
                        int distribution, GRP* grid, int dim) ;

void setRangeCollapse(DIM* rng, int extent) ;

void setRangeAlign(DIM* rng, int extent,
                   int offset, int stride, DIM* rng_parent) ;

void setRangeCopy(DIM* rng, DIM* rng_parent) ;
```

The `setRange...` calls initialize "stand-alone" `DIM` objects. There are four variants which create variants with different distribution or alignment formats.

`DIM` structures normally exist inside array descriptors (see section 4). 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.

`extent` is the size of the range. `setRangeDistribute` initializes a range distributed over a specified grid dimension with a specified distribution format. `setRangeCollapse` initializes a collapsed index range. `setRangeAlign` initializes a range aligned to some parent range with some offset and stride. `setRangeCopy` makes an identical copy of some parent range.

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

```
int shp_p [1] ;
GRP grd_p ;
DIM rng_tx1, rng_tx2 ;
...

shp_p [0] = 4 ;
gridInit(1, shp_p, &grd_p) ;

newRangeDistribute(&rng_tx1, 200, 1, grd_p, 1) ;
newRangeCollapse(&rng_tx2, 200) ;
```

The `DIM` structures `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

```
DIM rng_i, rng_j ;
...

setRangeAlign(&rng_i, 25, &rng_tx1, 25, 1) ;
setRangeAlign(&rng_j, 25, &rng_tx2, 25, 1) ;

loopBounds(&rng_i, &le1, &ll1, &ls1) ;
loopBounds(&rng_j, &le1, &ll2, &ls2) ;
for(i = 0, l_i = ll1 ; i < le1 ; i++, l_i += ls1)
  for(j = 0, l_j = ll2 ; j < le2 ; j++, l_j += ls2)
    x [l_i] [l_j] = ... ;
```

The `loopBounds` operation will be described in section 9. 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 initialization

```
void setArrayData(DAD* dad, void* array,
                  int element_type, int element_size,
                  int rank, int majority, GRP* grp) ;

void setArrayRangeDistribute(DAD* dad, int dim, int extent,
                             int distribution, GRP* grid, int dim) ;

void setArrayRangeCollapse(DAD* dad, int dim, int extent) ;

void setArrayRangeAlign(DAD* dad, int dim, int extent,
                        int offset, int stride, DIM* rng_parent) ;

void setArrayRangeCopy(DAD* dad, int dim, DIM* rng_parent) ;

void setArrayDataDone(DAD* dad) ;
```

The function `setArrayData` initialises the "per-array" fields in the descriptor. The second argument is the node-program array segment. 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 7). 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 `setArrayRange...` routines followed by one call to `setArrayDataDone`. All of these calls reference the DAD under construction as their first argument. The `setArrayRange...` 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 `setArrayData` call, or it must be collapsed. This constraint must be respected whether the dimension is distributed directly through `setArrayRangeDistribute` or indirectly through `setArrayRangeAlign` or `setArrayRangeCopy`.

The different variants of `setArrayRange...` allow for different distribution and alignment formats. See section 3 for further details—the four `setArrayRange...` calls are directly analogous to the `setRange...` variants described in section 3[1].

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

```
  float x [100] ;
  DAD dad_x ;

  ... Define grd_p and rng_tx1

  setArrayData(&dad_x, x, 2, 4, 1, 1, &grd_p) ;
  setArrayRangeAlign(&dad_x, 1, 98, &rng_tx1, 4, 2) ;
  setArrayDataDone(&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

```
  float x [25] [100] ;
  DAD dad_x ;
```

[1]Too all intents, `setArrayRange...(dad, dim, ...)` is equivalent to `setRange...(rng(dad, dim), ...)`.

```
... Define grd_p

setArrayData(&dad_x, x, 2, 4, 2, 2, &grd_p) ;
setArrayRangeDistribute(&dad_x, 1, 100, 1, &grd_p, 1) ;
setArrayRangeCollapse(&dad_x, 2, 100) ;
setArrayDataDone(&dad_x) ;
```

5 Section creation

```
void setArraySection(DAD* dad, DAD* dad_parent) ;

void setArrayTriplet(DAD* dad, int dim,
                     int extent, int offset, int stride
                     DAD* dad_parent, int dim_parent) ;

void setArrayScalar(DAD* dad,
                    int i,
                    DAD* dad_parent, int dim_parent) ;

void setArraySectionDone(DAD* dad) ;
```

The function `setArraySection` initializes the "per-array" fields in an array descriptor describing a *section* of some parent array. The `dad_parent` 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 `setArrayTriplet` and `setArrayScalar` routines to define the section subscripts, followed by one call to `setArraySectionDone`. 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 `setArrayTriplet` 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 `setArrayTriplet`. The last three arguments define the lower and upper bounds and stride of a triplet subscript into the parent array dimension.

The `setArrayScalar` calls affect the base address and process group fields (see section 7) 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

```
setArraySection(&dad_xs, &dad_x) ;
setArrayTriplet(&dad_xs, 1, 50, 0, 2, &dad_x, 1) ;
setArrayScalar(&dad_xs, 0, &dad_x, 2) ;
setArraySectionDone(dad_xs) ;
```

6 Array dummy arguments

```
int str(DAD* dad, int dim) ;
```

In general, when translating Fortran 90 to C multi-dimensional array dummy arguments must be translated to one-dimensional array formal parameters, 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 C two-dimensional array.

(Moreover, even FORTRAN 77 allows any extent of an array dummy argument to be a non-constant expressions, whereas C only allows non-constant array extents for the first dimension of a multi-dimensional array.)

The C translation of the subroutine `foo` could be

```
void foo(float* y, DAD* dad_y) {
  ...

  y [(i - 1) * str(dad_y, 1) + (j - 1) * str(dad_y, 2)] = ... ;
}
```

7 Mapping inquiry functions

```
DIM* rng(DAD* dad, int dim) ;

GRP* grp(DAD* dad)

int on(GRP* grp) ;
```

The inquiry function `rng` returns a pointer 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 a non-zero value if the local process is part of the process group defined by its argument, and zero 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, 10)
!HPF$ INHERIT y
      INTEGER i

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

      END
```

could translate to the node subroutine

```
void foo(float* y, DAD* dad_y) {

  int str_y1 ;
  int le, lu, le, i, li ;

  if(on(grp(dad_y)) {
    str_y1 = str(dad_y, 1) ;
    loopBounds(rng(dad_y, 1), &le, &ll, &ls) ;
    for(i = 0, li = ll ; i < le ; i++, li += ls)
      y [li * str_y1] = ...  ;
  }
}
```

The call

```
rng(dad, dim)
```

is equivalent to

```
dad->distInfo + dim - 1
```

This function could be be replaced by a macro.

8 Dynamic allocation of temporary arrays

```
void* resetArrayAlloc(DAD* dad) ;

void resetArrayFree(DAD* dad) ;
```

The function `resetArrayAlloc` allocates space for a temporary array on the stack. Its argument is a pointer to a fully initialized array descriptor. The value of the data address field in this descriptor is overwritten with a pointer to the allocated space. The function also returns this pointer.

The function `resetArrayFree` 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

```
void foo(float* x, DAD* dad_x, float* y, DAD* dad_y) {

  GRP* grp_x ;

  DAD dad_tmp1 ;
  float* bas_tmp1 ;

  ... Calls to reset_array_range_lb

  grp_x = grp(dad_x) ;

  setArrayData(&dad_tmp1, NULL, 2, 4, 1, 1, grp_x) ;
  setArrayRangeCopy(&dad_tmp1, 1, rng(dad_x, 1)) ;
  setArrayDataDone(&dad_tmp1) ;

  bas_tmp1 = (float*) resetArrayAlloc(&dad_tmp1) ;

  remap(&dad_tmp1, dad_y)

  if(on(grp_x)) {
    int i, li, le, ll, ls ;
    int str_x_1 = str(dad_x, 1) ;

    loopBounds(rng(dad_x, 1), &le, &ll, &ls) ;
    for(i = 0, li = ll ; i < le ; i++, li += ls)
      x [li * str_x_1] = x [li * str_x_1] + bas_tmp1 [li] ;
  }

  resetArrayFree(&dad_tmp1) ;
}
```

The DAD for `tmp1` is initially set up with the same parameters as `x` and a data address of `NULL`. A call to `resetArrayAlloc` allocates a suitable portion of memory. After copying the values in `y` to the temporary by the `remap` call (see section 10), `x` is updated in terms of values in the temporary.

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

9 Address translation

```
int localToGlobal(DIM* rng, int l_i) ;

int globalToLocal(DIM* rng, int g_i) ;

void loopBounds(DIM*, int* l_ext, int* l_lb, int* l_stride)
```

The call

```
loopBounds(rng, le, ll, ls) ;
```

is equivalent to

```
*le = rng->l_extent ;
*ll = rng->l_lb - 1 ;
*ls = rng->l_stride ;
```

This function could be be replaced by a macro.

10 Communication

```
int detectCommunication(DAD* dad, DAD* dad_source,
                        int dim, int l, int u, int stride,
                        int a, int b,
                        float m, float c, int status) ;

void shift(DAD* dad, DAD* dad_source, int dim, int amount) ;

void remap(DAD* dad, 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-50)

      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
```

```
#include "pcrc.h"

void main(int argc, char* argv[]) {
  int shp_p [1] ;
  GRP grd_p ;

  DIM rng_tx1, rng_ty1 ;

  float x [100], y [100] ;
  float tmp1 [100], tmp2 [100], tmp3 [100], tmp4 [100] ;

  DAD dad_x, dad_y ;
  DAD dad_tmp1, dad_tmp2, dad_tmp3, dad_tmp4 ;
  DAD dad_ys ;

  int i, li, le, ll, ls, amount1, amount2, status1, status2
  int i1, i2
  float u1, v1, u2, v2

  DIM* rng ;

  pcrcInit(argc, argv) ;

/* Define processor arrangement */

  shp_p [0] = 4 ;
  gridInit(1, shp_p, &grd_p) ;

/* Define templates */

  setRangeDistribute(&rng_tx1, 400, 1, &grd_p, 1) ;

  setRangeDistribute(&rng_ty1, 400, 1, &grd_p, 2) ;

/* Define main arrays */

  setArrayData(&dad_x, x, 2, 4, 1, 1, &grd_p) ;
  setArrayRangeAlign(&dad_x, 1, 98, 4, 1, &rng_tx1) ;
  setArrayDataDone(&dad_x) ;

  setArrayData(&dad_y, y, 2, 4, 1, 1, &grd_p) ;
  setArrayRangeAlign(&dad_y, 1, 100, 153, 3, &rng_ty1) ;
  setArrayDataDone(&dad_y) ;

/* Do parallel loop x(i) = 0 */

  loopBounds(rng(dad_x, 1), &le, &ll, &ls)
```

```
  for(i = 0, li = ll ; i < le ; i++, li += ls)
    x [li] = 0 ;

/* Do parallel loop y(i) = i */

  loopBounds(rng(dad_y, 1), &le, &ll, &ls)
  for(i = 0, li = ll ; i < le ; i++, li += ls)
    y [li] = localToGlobal(rng(dad_y, 1), li) ;

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

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

  if(status1 == 1) {
    setArrayData(&dad_tmp1, tmp1, 2, 4, 1, 1, &grd_p) ;
    setArrayRangeCopy(&dad_tmp1, 1, rng(dad_y, 1)) ;
    setArrayDataDone(dad_tmp1) ;

    shift(&dad_tmp1, &dad_y, amount1) ;
  }
  else if(status1 == 2) {
    setArrayData(&dad_tmp3, tmp3, 2, 4, 1, 1, &grd_p) ;
    setArrayRangeCopy(&dad_tmp3, 1, rng(dad_x, 1)) ;
    setArrayDataDone(&dad_tmp3) ;

    setArraySection(&dad_ys, &dad_y) ;
    setArrayTriplet(&dad_ys, 1, 98, 2, 1, &dad_y, 1) ;
    setArraySectionDone(&dad_ys) ;

    remap(&dad_tmp3, &dad_ys) ;
  }

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

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

  if(status2 == 1) {
    setArrayData(&dad_tmp2, &tmp2, 2, 4, 1, 1, &grd_p) ;
    setArrayRangeCopy(&dad_tmp2, 1, rng(&dad_y, 1)) ;
    setArrayDataDone(dad_tmp2) ;

    shift(&dad_tmp2, &dad_y, amount2) ;
  }
  else if(status2 == 2) {
    setArrayData(&dad_tmp4, &tmp4, 2, 4, 1, 1, grd_p) ;
```

```
    setArrayRangeCopy(&dad_tmp4, 1, rng(&dad_x, 1)) ;
    setArrayDataDone(&dad_tmp4) ;

    setArraySection(&dad_ys, &dad_y) ;
    setArrayTriplet(&dad_ys, 1, 98, 0, 1, &dad_y, 1) ;
    setArraySectionDone(&dad_ys) ;

    remap(&dad_tmp4, &dad_ys) ;
  }

/* Do parallel loop  y(i) = y(i+1) + y(i-1) */

  loopBounds(rng(&dad_x, 1), &le, &ll, &ls) ;
  for(i = 0, li = ll ; i < le ; i++, li += ls) {
    i1 = (int) (u1 * i + v1 + 0.5) ;
    i2 = (int) (u2 * i + v2 + 0.5) ;

    if(status1 == 0 && status2 == 0)
      x [li] = y [i1] + y [i2] ;
    else if(status1 == 0 && status2 == 1)
      x [li] = y [i1] + tmp2 [i2] ;
    else if(status1 == 0 && status2 == 2)
      x [li] = y [i1] + tmp4 [i2] ;
    else if(status1 == 1 && status2 == 0)
      x [li] = tmp1 [i1] + y [i2] ;
    else if(status1 == 1 && status2 == 1)
      x [li] = tmp1 [i1] + tmp2 [i2] ;
    else if(status1 == 1 && status2 == 2)
      x [li] = tmp1 [i1] + tmp4 [i2] ;
    else if(status1 == 2 && status2 == 0)
      x [li] = tmp3 [i1] + y [i2] ;
    else if(status1 == 2 && status2 == 1)
      x [li] = tmp3 [i1] + tmp2 [i2] ;
    else
      x [li] = tmp3 [i1] + tmp4 [i2] ;
  }

  pcrcFinalize() ;
}
```

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-50)

      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
```

```
#include "pcrc.h"

void main(int argc, char* argv[]) {
  int shp_p [1] ;
  GRP grd_p ;

  DIM rng_tx1, rng_ty1 ;

  float x [100], y [100] ;
  float tmp1 [100], tmp2 [100], tmp3 [100], tmp4 [100] ;

  DAD dad_x, dad_y ;
  DAD dad_tmp1, dad_tmp2, dad_tmp3, dad_tmp4 ;
  DAD dad_ys, dad_xs ;

  int i, li, le, ll, ls, amount1, amount2, status1, status2
  int i1, i2
  float u1, v1, u2, v2

  DIM* rng ;

  pcrcInit(argc, argv) ;

/* Define processor arrangement */

  shp_p [0] = 4 ;
  gridInit(1, shp_p, &grd_p) ;

/* Define templates */

  setRangeDistribute(&rng_tx1, 400, 1, &grd_p, 1) ;

  setRangeDistribute(&rng_ty1, 400, 1, &grd_p, 2) ;

/* Define main arrays */

  setArrayData(&dad_x, x, 2, 4, 1, 1, &grd_p) ;
  setArrayRangeAlign(&dad_x, 1, 98, 4, 1, &rng_tx1) ;
  setArrayDataDone(&dad_x) ;

  setArrayData(&dad_y, y, 2, 4, 1, 1, &grd_p) ;
  setArrayRangeAlign(&dad_y, 1, 100, 153, 3, &rng_ty1) ;
  setArrayDataDone(&dad_y) ;

/* Do parallel loop x(i) = 0 */

  loopBounds(rng(dad_x, 1), &le, &ll, &ls)
```

```
  for(i = 0, li = ll ; i < le ; i++, li += ls)
    x [li] = 0 ;

/* Do parallel loop y(i) = i */

  loopBounds(rng(dad_y, 1), &le, &ll, &ls)
  for(i = 0, li = ll ; i < le ; i++, li += ls)
    y [li] = localToGlobal(rng(dad_y, 1), li) ;

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

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

  if(status1 == 1) {
    setArrayData(&dad_tmp1, tmp1, 2, 4, 1, 1, &grd_p) ;
    setArrayRangeCopy(&dad_tmp1, 1, rng(dad_y, 1)) ;
    setArrayDataDone(dad_tmp1) ;

    shift(&dad_tmp1, &dad_y, amount1) ;
  }
  else if(status1 == 2) {
    setArrayData(&dad_tmp3, tmp3, 2, 4, 1, 1, &grd_p) ;
    setArrayRangeCopy(&dad_tmp3, 1, rng(dad_x, 1)) ;
    setArrayDataDone(&dad_tmp3) ;

    setArraySection(&dad_ys, &dad_y) ;
    setArrayTriplet(&dad_ys, 1, 98, 2, 1, &dad_y, 1) ;
    setArraySectionDone(&dad_ys) ;

    remap(&dad_tmp3, &dad_ys) ;
  }

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

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

  if(status2 == 1) {
    setArrayData(&dad_tmp2, &tmp2, 2, 4, 1, 1, &grd_p) ;
    setArrayRangeCopy(&dad_tmp2, 1, rng(&dad_y, 1)) ;
    setArrayDataDone(dad_tmp2) ;

    shift(&dad_tmp2, &dad_y, amount2) ;
  }
  else if(status2 == 2) {
    setArrayData(&dad_tmp4, &tmp4, 2, 4, 1, 1, grd_p) ;
```

```
    setArrayRangeCopy(&dad_tmp4, 1, rng(&dad_x, 1)) ;
    setArrayDataDone(&dad_tmp4) ;

    setArraySection(&dad_ys, &dad_y) ;
    setArrayTriplet(&dad_ys, 1, 98, 0, 1, &dad_y, 1) ;
    setArraySectionDone(&dad_ys) ;

    remap(&dad_tmp4, &dad_ys) ;
  }

/* Do parallel loop  y(i) = y(i+1) + y(i-1) */

  loopBounds(rng(&dad_x, 1), &le, &ll, &ls) ;
  for(i = 0, li = ll ; i < le ; i++, li += ls) {
    i1 = (int) (u1 * li + v1 + 0.5) ;
    i2 = (int) (u2 * li + v2 + 0.5) ;

    if(status1 == 0 && status2 == 0)
      x [li] = y [i1] + y [i2] ;
    else if(status1 == 0 && status2 == 1)
      x [li] = y [i1] + tmp2 [i2] ;
    else if(status1 == 0 && status2 == 2)
      x [li] = y [i1] + tmp4 [i2] ;
    else if(status1 == 1 && status2 == 0)
      x [li] = tmp1 [i1] + y [i2] ;
    else if(status1 == 1 && status2 == 1)
      x [li] = tmp1 [i1] + tmp2 [i2] ;
    else if(status1 == 1 && status2 == 2)
      x [li] = tmp1 [i1] + tmp4 [i2] ;
    else if(status1 == 2 && status2 == 0)
      x [li] = tmp3 [i1] + y [i2] ;
    else if(status1 == 2 && status2 == 1)
      x [li] = tmp3 [i1] + tmp2 [i2] ;
    else
      x [li] = tmp3 [i1] + tmp4 [i2] ;
  }

/* Define arguments and make call to 'foo' */

  setArraySection(&dad_xs, &dad_x) ;
  setArrayTriplet(&dad_xs, 1, 49, 0, 2, &dad_x, 1) ;
  setArraySectionDone(dad_xs) ;

  setArraySection(&dad_ys, &dad_y) ;
  setArrayTriplet(&dad_ys, 1, 50, 0, 2, &dad_y, 1) ;
  setArraySectionDone(&dad_ys) ;
```

```
  foo(&dad_xs, &dad_ys) ;

/* Define arguments and make call to 'foo' */

  setArraySection(&dad_x) ;
  setArrayTriplet(&dad_xs, 1, 49, 1, 2, &dad_x, 1) ;
  setArraySectionDone(&dad_xs) ;

  setArraySection(&dad_ys, &dad_y) ;
  setArrayTriplet(&dad_ys, 1, 50, 1, 2, &dad_y, 1) ;
  setArraySectionDone(&dad_ys) ;

  foo(&dad_xs, &dad_ys)

  pcrcFinalize() ;
}


void foo(DAD* dad_x, DAD* dad_y) {
  float *x, *y ;

  GRP* grp_x ;

  DAD dad_tmp1, dad_tmp2, dad_ys ;
  float* bas_tmp1, bas_tmp2 ;
  int str_x_1 ;

  int i, li, le, ll, ls ;

  x = (float*) dad_x->baseAddress ;
  y = (float*) dad_y->baseAddress ;

  grp_x = grp(dad_x)

/* Allocate and write temporary for remapped y(i) */

  setArrayData(&dad_tmp1, NULL, 2, 4, 1, 1, &grp_x) ;
  setArrayRangeCopy(&dad_tmp1, 1, rng(dad_x, 1)) ;
  setArrayDataDone(&dad_tmp1) ;

  bas_tmp1 = (float*) resetArrayAlloc(&dad_tmp1) ;

  setArraySection(&dad_ys, dad_y) ;
  setArrayTriplet(&dad_ys, 1, 49, 0, 1, dad_y, 1) ;
  setArraySectionDone(&dad_ys) ;

  remap(&dad_tmp1, &dad_ys) ;
```

```
/* Allocate and write temporary for remapped y(i + 1) */

  setArrayData(&dad_tmp2, NULL, 2, 4, 1, 1, &grp_x) ;
  setArrayRangeCopy(&dad_tmp2, 1, rng(dad_x, 1)) ;
  setArrayDataDone(&dad_tmp2) ;

  bas_tmp2 = (float*) resetArrayAlloc(&dad_tmp2) ;

  setArraySection(&dad_ys, dad_y) ;
  setArrayTriplet(&dad_ys, 1, 49, 1, 1, dad_y, 1) ;
  setArraySectionDone(&dad_ys) ;

  remap(&dad_tmp2, &dad_ys) ;

/* Do parallel loop */

  if(on(grp_x)) {
     str_x_1  = str(&dad_x, 1) ;

     loopBounds(rng(&dad_x, 1), &le, &ll, &ls) ;
     for(i = 0, li = ll ; i < le ; i++, li += ls)
       x [li * str_x_1] = x [li * str_x_1] - bas_tmp1 [li] - bas_tmp2 [li] ;
  }
}
```

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
```

```
#include "pcrc.h"

void main(int argc, char* argv[]) {
  int shp_p [1] ;
  GRP grd_p ;

  DIM rng_tx1, rng_tx2, rng_ty1, rng_ty2 ;

  float x [25] [100], y [25] [100] ;

  DAD dad_x, dad_y ;

  DIM* rng ;

  DIM rng_i, rng_j ;

  DAD dad_ys, dad_xs ;

  DAD dad_tmp1, dad_tmp2 ;

  int i, j, li, le1, ll1, ls1, lj, le2, ll2, ls2 ;

  pcrcInit(argc, argv) ;

/* Define processor arrangement */

  shp_p [0] = 4 ;
  gridInit(1, shp_p, &grd_p) ;

/* Define templates */

  setRangeDistribute(&rng_tx1, 1, 200, 1, &grd_p, 1) ;
  setRangeCollapse(&rng_tx2, 1, 200) ;

  setRangeDistribute(&rng_ty1, 1, 100, 1, &grd_p, 1) ;
  setRangeCollapse(&rng_ty2, 1, 100) ;

/* Define main arrays */

  setArrayData(&dad_x, x, 2, 4, 2, 1, &grp_p) ;
  setArrayRangeAlign(&dad_x, 1, 1, 100, 0, 2, &rng_tx1) ;
  setArrayRangeAlign(&dad_x, 2, 1, 100, 0, 2, &rng_tx2) ;
  setArrayDataDone(&dad_x)

  setArrayData(&dad_y, y, 2, 4, 2, 1, &grp_p) ;
  setArrayRangeAlign(&dad_y, 1, 1, 100, 0, 1, &rng_ty1) ;
  setArrayRangeAlign(&dad_y, 2, 1, 100, 0, 1, &rng_ty2) ;
```

```
  setArrayDataDone(&dad_y) ;

/* Do parallel loop  x(i, j) = 0 */

  loopBounds(rng(&dad_x, 1), &le1, &ll1, &ls1) ;
  loopBounds(rng(&dad_x, 2), &le2, &ll2, &ls2) ;
  for(i = 0, li = ll1 ; i < le1 ; i++, li += ls1)
    for(j = 0, lj = ll2 ; j < le2 ; j++, lj += ls2)
      x [li] [lj] = 0 ;

/* Do parallel loop  y(i, j) = i + j */

  loopBounds(rng(&dad_y, 1), &le1, &ll1, &ls1) ;
  loopBounds(rng(&dad_y, 2), &le2, &ll2, &ls2) ;
  for(i = 0, li = ll1 ; i < le1 ; i++, li += ls1)
    for(j = 0, lj = ll2 ; j < le2 ; j++, lj += ls2)
      y [li] [lj] = localToGlobal(rng(&dad_y, 1), li) +
                    localToGlobal(rng(&dad_y, 2), lj) ;

  setRangeAlign(&rng_i, 25, 25, 1, rng(&dad_x, 1)) ;
  setRangeAlign(&rng_j, 25, 25, 1, rng(&dad_x, 2)) ;

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

  setArrayData(&dad_tmp1, tmp1, 2, 4, 1, 1, &grp_p) ;
  setArrayRangeCopy(&dad_tmp1, 1, &rng_i) ;
  setArrayRangeCopy(&dad_tmp1, 2, &rng_j) ;
  setArrayDataDone(&dad_tmp1) ;

  setArraySection(&dad_ys, &dad_y) ;
  setArrayTriplet(&dad_ys, 1, 25, 26, 1, &dad_y, 1) ;
  setArrayTriplet(&dad_ys, 2, 25, 25, 1, &dad_y, 2) ;
  setArraySectionDone(&dad_ys) ;

  remap(&dad_tmp1, &dad_ys) ;

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

  setArrayData(&dad_tmp2, tmp2, 2, 4, 1, 1, &grp_p) ;
  setArrayRangeCopy(&dad_tmp2, 1, &rng_i) ;
  setArrayRangeCopy(&dad_tmp2, 1, &rng_j) ;
  setArrayDataDone(&dad_tmp2) ;

  setArraySection(&dad_ys, &dad_y) ;
  setArrayTriplet(&dad_ys, 1, 25, 25, 1, &dad_y, 1) ;
  setArrayTriplet(&dad_ys, 2, 25, 26, 1, &dad_y, 2) ;
  setArraySectionDone(&dad_ys) ;
```

```
  remap(&dad_tmp2, &dad_ys) ;

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

  loopBounds(&rng_i, &le1, &ll1, &ls1) ;
  loopBounds(&rng_j, &le2, &ll2, &ls2) ;
  for(i = 0, li = ll1 ; i < le1 ; i++, li += ls1)
    for(j = 0, lj = ll2 ; j < le2 ; j++, lj += ls2)
      x [li] [lj] = tmp1 [li] [lj] + tmp2 [li] [lj] ;

/* Do parallel loop  x(i, j) = 1 */

  setRangeAlign(&rng_i, 50, 0, 2, rng(dad_x, 1)) ;
  setRangeAlign(&rng_j, 50, 0, 2, rng(dad_x, 2)) ;

  loopBounds(&rng_i, &le1, &ll1, &ls1) ;
  loopBounds(&rng_j, &le2, &ll2, &ls2) ;
  for(i = 0, li = ll1 ; i < le1 ; i++, li += ls1)
    for(j = 0, lj = ll2 ; j < le2 ; j++, lj += ls2)
      if(x [li] [lj] != 0) x [li] [lj] = 1 ;

/* Define arguments and make call to ‘foo’ */

  setArraySection(&dad_xs, &dad_x) ;
  setArrayTriplet(&dad_xs, 1, 50, 0, 2, &dad_x, 1) ;
  setArrayScalar(&dad_xs, 1, &dad_x, 2) ;
  setArraySectionDone(&dad_xs) ;

  foo(&dad_xs) ;

  pcrcFinalize() ;
}


void foo(DAD* dad_x) {
  float *x ;

  int shp_p [1] ;
  GRP grd_p ;

  float y [13] ;
  DAD dad_y ;

  DAD dad_tmp1 ;

  float* bas_tmp1 ;
```

```
  x = (float*) dad_x->baseAddress ;

/* Define processor arrangement */

  shp_p [0] = 4 ;
  CALL grid_init(1, shp_p, &grd_p) ;

/* Define main arrays */

  setArrayData(&dad_y, y, 2, 4, 2, 1, &grp_p) ;
  setArrayRangeDistribute(&dad_y, 1, 50, 1, &grd_p, 1) ;
  setArrayDataDone(&dad_y) ;

/* Do parallel loop  y(i) = 2 */

  loopBounds(rng(&dad_y, 1), &le, &ll, &ls) ;
  for(i = 0, li = ll ; i < le ; i++, li += ls)
    y [li] = 2 ;

/* Allocate and write temporary for remapped y(i) */

  setArrayData(&dad_tmp1, NULL, 2, 4, 1, 1, grp(&dad_x)) ;
  setArrayRangeCopy(&dad_tmp1, 1, rng(&dad_x, 1)) ;
  setArrayDataDone(&dad_tmp1) ;

  bas_tmp1 = (float*) resetArrayAlloc(&dad_tmp1) ;

  remap(&dad_tmp1, &dad_y) ;

/* Do parallel loop  x(i) = y(i) + 1 */

  if(on(grp(&dad_x))) {
    str_x_1  = str(&dad_x, 1) ;

    loopBounds(rng(&dad_x, 1), &le, &ll, &ls) ;
    for(i = 0, li = ll ; i < le ; i++, li += ls)
      x [li * str_x_1] = bas_tmp1 [li] + 1 ;
   }

   resetArrayFree(&dad_tmp1) ;
}
```