A Manual for Multiblock PARTI

Alan Sussman*
Gagan Agrawal
Department of Computer Science, University of Maryland, College Park, MD 20742

Christian Hansen
Joel Saltz
UMIACS and Department of Computer Science, UMCP, College Park, MD 20742
{als,gagan,chansen,saltz}@cs.umd.edu

January 2004

Abstract

There exists a large class of scientific applications that are composed of irregularly coupled regular mesh (ICRM) computations. These problems are often referred to as block structured, or multiblock, problems and include the Block Structured Navier-Stokes solver developed at NASA Langley called multiblock TLNS3D.

Primitives are presented that are designed to help users efficiently program such problems on distributed memory machines. These primitives are also designed for use by compilers for distributed memory multiprocessors. Communications patterns are captured at runtime, and the appropriate send and receive messages are automatically generated. The primitives are also useful for parallelizing regular computations, since block structured computations also require all the runtime support necessary for regular computations.

*This work was supported by ARPA under contract No. NAG-1-1485 and NSF under grant No. ASC 9213821. The work was supported by NASA Contract No. NAS1-18605 while the authors were in residence at ICASE, NASA Langley Research Center.

1 C Function Descriptions

This section describes each of the C multiblock primitives. The primitives are divided into five categories: declarations, loop-bound adjustment, communication, memory management, and a group of miscellaneous primitives. These primitives are described, respectively, in Sections 1.2, 1.3, 1.4, 1.5 and 1.6. To help in our explanation we will present a very simple example of a multiblock code that will be referred to in the following sections.

The code, shown in Figures 1, 2 and 3, consists of an initialization step and an iterative computation phase. Initialization involves reading the number of blocks, the size of each block, and information about how blocks interface with each other. The interface information is obtained by the routine *readMap()*. For each block, **A**, *readMap()* reads data into **imap**[**A**] that specifies, for each subsection of **A** that interfaces with another block, **B**, three pieces of information: the coordinates of the section of **A**, the coordinates of the section of **B**, and the identity of **B**. **A** and **B** are allowed to be the same block.

The computation step calls two routines *fillBCells()* and *compute()*. Function *fillBCells()*, shown in Figure 2, uses the data structure *imap* to update the boundary cells of each block from other blocks. In line 13 of the driver code (Figure 1), array w is allocated an additional layer of cells, which we will call *external ghost cells*. External ghost cells are part of the user-defined (distributed) array, and are specific to the application (e.g. in a multiblock application, to hold data from adjacent blocks). On the other hand, we will later discuss allocation of *internal ghost cells*, which are required because the data set is divided among multiple processors (e.g. so a processor can store copies of data that is owned by other processors). Function *compute()*, shown in Figure 3, performs a simple sweep over a mesh, performing a computation at each mesh point.

For simplicity, the blocks depicted in this example are 2-dimensional. In most real problems the blocks will be 3-dimensional. *MAXBLOCS* is the maximum number of blocks allowed and *MXSIZE* is the number of data items required to describe each block interface.

1.1 Header File

There is one header file for the multiblock PARTI library, called *bsparti.h*. The header file contains the structure definitions, macro definitions and function declarations needed to use the primitives. *bsparti.h* must be included by all C programs that use the multiblock PARTI primitives. No header files need to be included by Fortran applications.

1.2 Declaration Primitives

The declaration primitives allow the user to create decompositions and distributed array descriptors. A decomposition can be thought of as an abstract problem or index space. The

```
 1  main()
 2  {
 3    int    numBlocks, numIters, size;
 4    int    i, iter, size1[MAXBLOCS], size2[MAXBLOCS];
 5    float  *w[MAXBLOCS], *deltaw[MAXBLOCS];
 6    int imap[MAXBLOCS][MXSIZE];
 7
 8    scanf("%d", &numBlocks);
 9    scanf("%d", &numIters);
10    for (i=0; i<numBlocks; i++) {
11      scanf("%d %d", &size1[i], &size2[i]);
12      size = (size1[i]+2)*(size2[i]+2);
13      w[i] = calloc(size, sizeof(float));
14      deltaw[i] = calloc(size, sizeof(float));
15      readMap(&imap[i][0]);
16    }
17
18    for (iter=0; iter<numIters; iter++) {
19      for (i=0; i<numBlocks; i++) {
20        fillBCells(w[i], imap[i], i, size1, size2);
21      }
22
23      for (i=0; i<numBlocks; i++) {
24        compute(w[i], deltaw[i], size1[bloc]+2, size2[bloc]+2);
25      }
26    }
27  }
```

Figure 1: C multiblock code - main program

```
 1 fillBCells(w, imap, dBloc, size1, size2)
 2   float  *w[MAXBLOCS];
 3   int    imap[MXSIZE], dBloc, size1[MAXBLOCS], size2[MAXBLOCS];
 4 {
 5   int  i=0; seg, nsegs;
 6
 7   nsegs = imap[i++];
 8   for (seg=1; seg<=nsegs; seg++) {
 9     dDim = imap[i++];  dVal = imap[i++];  dLo  = imap[i++];
10     dHi  = imap[i++];  sDim = imap[i++];  sVal = imap[i++];
11     sLo  = imap[i++];  sHi  = imap[i++];  sBloc = imap[i++];
12     copySegment(
13       w[dBloc],dDim,dVal,dLo,dHi,size1[dBloc],size2[dBloc],
14       w[sBloc],sDim,sVal,sLo,sHi,size1[sBloc],size2[sBloc]);
15   }
16 }
17
18 copySegment(dest,dDim,dVal,dLo,dHi,dSize1,dSize2,
19             src,sDim,sVal,sLo,sHi,sSize1,sSize2)
20   int   dDim, dVal, ........, sSize1, sSize2;
21   float src[sSize1+2][sSize2+2], dest[dSize1+2][dSize2+2];
22 {
23   int   i, j, cnt=0;
24
25   if (sDim == 0)
26     for (j=srcLo; j<=srcHi; j++)
27       tmp[cnt++] = src[sVal][j];
28   else
29     for (i=srcLo; i<==srcHi; i++)
30       tmp[cnt++] = src[i][sVal];
31
32   cnt = 0;
33   if (dDim == 0)
34     for (j=destLo; j<=destHi; j++)
35       dest[dVal][j] = tmp[cnt++];
36   else
37     for (i=destLo; i<==destHi; i++)
38       dest[i][dVal] = tmp[cnt++];
39 }
```

Figure 2: C multiblock code - fillBCells and copySegment functions

```
 1 compute(w, deltaw, size1, size2)
 2   int    size1, size2
 3   float  w[size1][size2], deltaw[size1][size2];
 4 {
 5   int i,j,k,l,m;
 6
 7   for (j=1; j<=size2-2; j++) {
 8     for (i=1; i<=size1-2; i++) {
 9       deltaw[i][j]  = k * (w[i+1][j] - w[i][j]) +
10                       l * (w[i][j+1] - w[i][j]) +
11                       m * w[i+1][j+1];
12     }
13   }
14 }
```

Figure 3: C multiblock code - compute function

library differentiates two types of decompositions, regular and irregular, each of which is created with its own set of primitives. A distributed array descriptor describes the physical and distribution characteristics of an array. This includes the number of dimensions, the local size in each dimension, the number of ghost (overlap) cells in each dimension, the distribution in each dimension, and a pointer to the decomposition with which it is associated. Every distributed array is described by a distributed array descriptor. Block to task assignments are one-to-one. One descriptor can serve for multiple distributed arrays, since arrays with identical physical characteristics (e.g. identical distribution, the same number of ghost cells in each dimension, etc.) can use the same descriptor.

1.2.1 vProc()

vProc() creates a virtual processor space. This virtual processor space can be thought of as a type of "father" decomposition that encompasses the entire problem space and all available processors. The virtual processor space in dimension i is numbered from 0 to $sizes[i] - 1$.

Synopsis

VPROC *vProc(numDims, sizes)

Parameter Declarations

int numDims number of dimensions in the virtual processor space. Currently, only one dimensional virtual processor spaces are supported.

int sizes[numDims] sizes of all the dimensions

Return Value

pointer to a descriptor for the virtual processor space

Example

Consider the initialization loop (lines 10 to 16) in Figure 1. We would like to create a virtual processor space representing the entire problem space (i.e. all blocks). The following code shows how to use the sizes of the blocks to calculate the size of the entire problem space and calls *vProc()* to create a corresponding virtual processor space.

```
totalSize[0] = 0;
for (i=0; i<numBlocks; i++) {
  scanf("%d %d", &size1[i], &size2[i]);
  size = (size1[i]+2) * (size2[i]+2);
  w[i] = calloc(size, sizeof(float));
  deltaw[i] = calloc(size, sizeof(float));
  readMap(&imap[i][0]);
  totalSize[0] += size1[i] * size2[i];
}
vProc(1, totalSize);
```

1.2.2 create_decomp()

create_decomp() creates a new decomposition with "numDims" dimensions. The size of each dimension is given by array "sizes".

Synopsis

DECOMP *create_decomp(numDims, sizes)

Parameter Declarations

int numDims number of dimensions in decomposition

int sizes[numDims] sizes of all the dimensions

Return Value

pointer to a descriptor for the decomposition

Example

The following example uses the code from Section 1.2.1 and shows how decompositions are created for each block.

```
totalSize[0] = 0;
for (i=0; i<numBlocks; i++) {
  scanf("%d %d", &size1[i], &size2[i]);
  size = (size1[i]+2) * (size2[i]+2);
  w[i] = calloc(size, sizeof(float));
  deltaw[i] = calloc(size, sizeof(float));
  readMap(&imap[i][0]);
  totalSize[0] += size1[i] * size2[i];
  sizes[0] = size1[i];  sizes[1] = size2[i];
  decomp[i] = create_decomp(2, sizes);
}
vProc(1, totalSize);
```

1.2.3 embed()

embed() embeds a decomposition into a subset of the virtual processor space. This primitive makes it possible to map a decomposition onto a specified set of virtual processors. This primitive should always be called before *distribute()*, which needs to know how many processors have been allocated to the decomposition. Currently, *embed()* supports only one dimensional virtual processor spaces.

Synopsis

void embed(decomp, vproc, startPosn, endPosn)

Parameter Declarations

DECOMP *decomp pointer to the decomposition

VPROC *vproc pointer to the virtual processor space (currently, must be one dimensional)

int startPosn start position in the virtual processor space

int endPosn end position in the virtual processor space

Return Value

none

Example

Assume a virtual processor space and a number of decompositions have already been created, as shown in the example from Section 1.2.2. The following example shows how the decompositions corresponding to each block are embedded into the virtual processor space.

```
startPosn = 0;
for (i=0; i<numBlocks; i++) {
  endPosn = startPosn + size1[i] * size2[i] - 1;
  embed(decomp[i], vp, startPosn, endPosn);
  startPosn = endPosn + 1;
}
```

1.2.4 distribute()

distribute() allows the user to specify the type of distribution for each dimension of a specified decomposition. This routine determines how the dimensions of the decomposition are mapped to the set of virtual processors specified by *embed()*. If *embed()* has not been used to specify a set of virtual processors for the decomposition, *distribute()* assumes that the decomposition is mapped to *all* processors. While *distribute()* can, in general, deal with a broad range of distributions (e.g. block, cyclic, block cyclic, undistributed and irregular), currently only block and undistributed are implemented.

Synopsis

void distribute(decomp, dist)

Parameter Declarations

DECOMP *decomp pointer to the decomposition

char *dist string of N characters, where N is the number of dimensions in the decomposition. Each character can have one of these values:

'*' Undistributed

'B' Block distribution

'C' Cyclic distribution (not yet implemented)

Return Value

none

Example

Using the subset of code from Section 1.2.3, we add a call to *distribute()* in the loop to specify that the distribution for each decomposition is "block" in the first and second dimensions.

```
startPosn = 0
for (i=0; i<numBlocks; i++) {
  endPosn = startPosn + size1[i] * size2[i] - 1;
  embed(decomp[i], vp, startPosn, endPosn);
  startPosn = endPosn + 1;
  distribute(decomp[i], "BB");
}
```

1.2.5 section()

Irregular block distributions can be handled with the *section()* call, which creates all the partitions for all the dimensions of an irregular block decomposition at once. This call does not consider a virtual processor space and instead assigns blocks to tasks based on their order in the decomposition.

Synopsis

int section(decomp, dim, cnt, idx)

Parameter Declarations

DECOMP *decomp pointer to the decomposition

int *dim the dimensions to split

int *cnt the number of partitions to create in each dimension

int **idx the limiting upper index of each partition for each dimension

Return Value

the number of blocks created

Example

Here is a use of *section()* that bisects each dimension.

```
void bisect(DECOMP* decomp, int rank) {
  int i, *dim, *cnt, **idx;
```

```
  dim = (int*)calloc(rank, sizeof(int));
  cnt = (int*)calloc(rank, sizeof(int));
  idx = (int**)calloc(rank, sizeof(int*));
  for (i = 0; i < rank; ++i) {
    dim[i] = i;
    cnt[i] = 2;
    idx[i] = (int*)calloc(2, sizeof(int));
    idx[i][0] = decomp->size[i]/2; idx[i][1] = decomp->size[i];
  }

  parti_section(decomp, dim, cnt, idx);

  for (i = 0; i < rank; ++i) {
    free(idx[i]);
  }
  free(idx);
  free(cnt);
  free(dim);
}
```

1.2.6 partition()

Irregular block distributions can also be created with the *partition()* call, which is used to recursively partition each dimension, one at a time. This allows remaining, unpartitioned dimensions to be partitioned differently. For example, if the first dimension is partitioned into two sections, *partition()* can then be called to partition the second dimension of the first section into two sections and called again to partition the second dimension of the second section into three sections. The parameter *blk* is used to indicate which section (or block) of each dimension is to be partitioned on a particular call. This call does not consider a virtual processor space and instead assigns blocks based on their order in the decomposition.

Synopsis

int partition(decomp, dim, blk, cnt, idx)

Parameter Declarations

DECOMP *decomp pointer to the decomposition

int dim the dimension to split

int *blk the block to split

int cnt the number of partitions to create

int *idx the limiting upper index of each partition

Return Value

the number of partitions created

Example

Here is a recursive use of *partition()* that bisects each dimension.

```
void bisect(DECOMP *decomp, int rank, int *size, int dim, int *blk) {
  int cnt = 2;
  int idx[2];

  if (dim < rank) {
    idx[0] = size[dim]/2; idx[1] = size[dim];
    partition(decomp, dim, blk, cnt, idx);
    blk[dim] = 0;
    bisect(decomp, rank, size, dim+1, blk);
    blk[dim] = 1;
    bisect(decomp, rank, size, dim+1, blk);
  }
}
```

1.2.7 align()

The *align()* primitive is used to map arrays onto a decomposition and create distributed array descriptors. The following parameters are associated with each array dimension: the size, the number of internal ghost cells, the number of external ghost cells at the beginning and end of each dimension, a flag for determining where to put extra data points in case the size of the dimension is not evenly divisible by the number of processors, and the decomposition dimension to which it is aligned. Internal ghost cells are those required because the data set is divided among multiple processors. External ghost cells are a part of the user-defined distributed array that are specific to the application. If a distributed array dimension will not be aligned to any decomposition dimension, then the decomposition dimension should be set to -1.

Synopsis

DARRAY *align(decomp, numDims, arrayDims, sizes, int_gcells,
ext_gcells_left, ext_gcells_right, extra_flag, decompDims)

Parameter Declarations

DECOMP *decomp pointer to the decomposition to which the array is being aligned.

int numDims the number of dimensions of the array.

int arrayDims[numDims] dimensions of the src array to be aligned. Numbering follows the C convention (i.e. 0 to numDims-1)

int sizes[numDims] sizes of each dimension (including external ghost cells)

int int_gcells[numDims] the number of internal ghost cells in each dimension

int ext_gcells_left[numDims] the number of external ghost cells at the beginning in each dimension

int ext_gcells_right[numDims] the number of external ghost cells at the end in each dimension

int extra_flag[numDims] where to put extra data items if the array size is not evenly divisible by the number of processors assigned to a dimension

- **0** : default (same as option 4)
- **1** : All on leftmost processor (the lowest numbered processor in the dimension)
- **2** : All on rightmost processor (the highest numbered processor in the dimension)
- **3** : Split equally between leftmost and rightmost processors (if an odd number the extra one is on the *leftmost*)
- **4** : Split equally between leftmost and rightmost processors (if an odd number the extra one is on the *rightmost*)

int decompDims[numDims] the decomposition dimensions to which each array dimension is being aligned. decompDims[i] should be -1 if dimension i is not aligned to any dimension of the decomposition.

Return Value

pointer to a distributed array descriptor that can be used as an argument to the schedule generating primitives described in Section 1.4

Example 1

Consider the example from Section 1.2.4. For each block we create a distributed array descriptor for a two dimensional array whose size in each dimension is the size of the block plus two (one external ghost cell on both the left and the right in each dimension), and that has one internal ghost cell in each dimension. If the number of data items in a distributed array dimension is not divisible by the number of (physical) processors assigned to that dimension (by the *distribute()* primitive), the *extra_flag* for each dimension is set so that any extra data items are split between the leftmost and rightmost processors of that dimension (with an extra data item possibly located on the rightmost processor). A distributed array descriptor contains distribution information that is used by other primitives (e.g.. laubnd(), lalbnd(), exchSched(), etc.) that require access to distribution information.

```
startPosn = 0
for (i=0; i<numBlocks; i++) {
  endPosn = startPosn + size1[i] * size2[i] - 1;
  embed(decomp[i],vp,startPosn,endPosn);
  startPosn = endPosn + 1;
  distribute(decomp[i], "BB");
  numdims = 2;
  array_dims[0] = 0;          array_dims[1] = 1;
  sizeinfo[0] = size1[i]+2;   sizeinfo[1] = size2[i]+2;
  int_gcells[0] = 1;          int_gcells[1] = 1;
  ext_gcells_left[0] = 1;     ext_gcells_right[0] = 1;
  ext_gcells_left[1] = 1;     ext_gcells_right[1] = 1;
  extra_flag[0] = 0;          extra_flag[1] = 0;
  decomp_dims[0] = 0;         decomp_dims[1] = 1;
  da[i] = align(decomp[i], numdims, array_dims, sizeinfo, int_gcells,
        ext_gcells_left, ext_gcells_right, extra_flag, decomp_dims);
}
```

Example 2

This example shows how to align an array with a decomposition of fewer dimensions. The array has 3 dimensions while the decomposition has just 2 dimensions. Also, the first and second array dimensions are transposed when mapped to the decomposition.

```
numdims = 3;
array_dims[0] = 0;  array_dims[1] = 1;  array_dims[2] = 2;
sizeinfo[0] = size1[i]+2;  sizeinfo[1] = size2[i]+2;
sizeinfo[2] = size3;
int_gcells[0] = 1;       int_gcells[1] = 1;  int_gcells[2] = 0;
ext_gcells_left[0] = 1;  ext_gcells_right[0] = 1;
ext_gcells_left[1] = 1;  ext_gcells_right[1] = 1;
ext_gcells_left[2] = 0;  ext_gcells_right[1] = 0;
extra_flag[0] = 0;       extra_flag[1] = 0;  extra_flag[2] = 0;
decomp_dims[0] = 1;  decomp_dims[1] = 0;  decomp_dims[2] = -1;
da = align(decomp, numdims, array_dims, sizeinfo, int_gcells,
         ext_gcells_left, ext_gcells_right, extra_flag, decomp_dims);
```

1.3 Loop Bound Adjustment Primitives

distribute() assigns a range of indices to each processor for each dimension of a decomposition, and *align()* then maps the elements of a distributed array to the decomposition

elements. Therefore, when a loop traverses a range of indices for a distributed array, that range may be distributed over several processors. For each processor, the start and end of the local range being traversed depends on the portion of the array residing on the processor, on the global range being traversed and on the stride. The functions *lalbnd()* and *laubnd()* compute and return the local start and stop index values for each processor, for the common situation in which

- the loop has a left hand side distributed array reference indexed by a loop iteration variable, and
- the "owner computes" rule is being used to assign computations to processors.

1.3.1 lalbnd()

Synopsis

int lalbnd(dArray, dim, start, stride)

Parameter Declarations

DARRAY *dArray pointer to distributed array descriptor

int dim array dimension being traversed. Numbering follows the C convention (i.e. 0 to N-1 where N = number of dimensions in the distributed array)

int start the (global) start of the range being traversed

int stride distance between elements, positive means counting up, negative means counting down

Return Value

an integer corresponding to the (local) start index of the range

Example

See the example in Section 1.3.2.

1.3.2 laubnd()

Synopsis

int laubnd(dArray, dim, stop, stride)

Parameter Declarations

DARRAY *dArray pointer to distributed array descriptor

int dim array dimension being traversed. Numbering follows the C convention (i.e. 0 to N-1 where N = number of dimensions in the distributed array)

int stop the (global) end of the range being traversed

int stride distance between elements, positive means counting up, negative means counting down

Return Value

an integer corresponding to the (local) last index of the range

Example

Assume that *da* has already been created, as in Example 1 from Section 1.2.7. The following example shows how *lalbnd()* and *laubnd()* are used to adjust the loop bounds in the *compute()* routine from Figure 3.

```
for(j=lalbnd(da, 1, 1, 1); j<=laubnd(da, 1, size2-2, 1); j++){
  for(i=lalbnd(da, 0, 1, 1); i<=laubnd(da, 0, size1-2, 1); i++){
    deltaw[i][j] = .......
  }
}
```

1.4 Communication Primitives

Two types of primitives for building communication schedules are provided. The *ghostFillSched()* variants and *exchSched()* are used for updating the internal ghost cells for a distributed array, while *subArraySched()* is used to move regular sections between distributed arrays (or within the same distributed array). The primitives also save the schedules in hash tables, so that the same schedule does not have to be computed more than once. If the required schedule is already available in the hash table, the primitives return a pointer to the existing schedule. Primitives for freeing the storage used by a schedule once it is no longer needed are described in Section 1.5.

1.4.1 ghostFillSched()

ghostFillSched() computes a schedule describing the data motion necessary to fill in a set of overlap/ghost cells for a distributed array with "ndims" dimensions. The physical characteristics and distribution of the array (i.e. number of dimensions, local dimension size,

number of ghost cells in each dimension, distribution in each dimension, etc.) are described by the data structure pointed to by the parameter "dArrayPtr". The parameter "fillVec", of length "ndims", describes the overlap cells to be filled, and represents the offsets in each dimension of a reference to the distributed array. This routine only fills in exactly the cells asked for, not all the possible permutations. For this routine, given a block distributed array, a processor sends to (at most) one other processor and receives from (at most) one other processor.

Synopsis

SCHED *ghostFillSched(dArrayPtr, ndims, fillVec)

Parameter declarations

DARRAY *dArrayPtr pointer to the distributed array descriptor

int ndims the number of dimensions of the distributed array , and the length of fillVec

int fillVec[ndims] the offset vector describing the ghost cells to be filled. The magnitude of the offset vector in a dimension determines the number of ghost cells filled. A positive value in a dimension means high indices are filled and a negative value means low indices are filled.

Return value

pointer to a structure of type SCHED that describes the necessary data motion, which can be used by the *DataMove()* primitives

Example

Consider the assignment statement to array *deltaw* in line 9 from the routine *compute()* shown in Figure 3. Assume the distributed array descriptor has already been created, by the code shown in Example 1 from Section 1.2.7. *da* is passed to *compute()* as a parameter and describes the distribution of *w* and *deltaw*. Off-processor fetches are required to satisfy the references to *w[i+1][j]*, *w[i][j+1]* and *w[i+1][j+1]* on the right hand side of the assignment. The following example shows how calls to the primitives *ghostFillSched()* and *fDataMove()* perform the communication necessary to ensure that the appropriate values are placed in the ghost cells of *w*. *fDataMove()*, which is discussed in detail in Section 1.4.6, uses a schedule produced by *ghostFillSched()* to perform the actual communication.

```
fillVec[0] = 1;  fillVec[1] = 0;
sched1 = ghostFillSched(da, 2, fillVec);
fillVec[0] = 0;  fillVec[1] = 1;
sched2 = ghostFillSched(da, 2, fillVec);
```

```
fillVec[0] = 1;  fillVec[1] = 1;
sched3 = ghostFillSched(da, 2, fillVec);
fDataMove(w, sched1, w);
fDataMove(w, sched2, w);
fDataMove(w, sched3, w);

for(j=lalbnd(da, 1, 1, 1); j<=laubnd(da, 1, size2-2, 1); i++){
  for(i=lalbnd(da, 0, 1, 1); i<=laubnd(da, 0, size1-2, 1); i++){
    deltaw[i][j]  = k * (w[i+1][j] - w[i][j]) +
                    l * (w[i][j+1] - w[i][j]) +
                    m * w[i+1][j+1];
  }
}
```

1.4.2 ghostFillSpanSched()

ghostFillSpanSched() computes a schedule describing the data motion necessary to fill in a set of overlap/ghost cells for a distributed array with "ndims" dimensions. The physical characteristics and distribution of the array (i.e. number of dimensions, local dimension size, number of ghost cells in each dimension, distribution in each dimension, etc.) are described by the data structure pointed to by the parameter "dArrayPtr". The parameter "fillVec", of length "ndims", describes the overlap cells to be filled, and represents the offsets in each dimension of a reference to the distributed array. This routine fills in all the possible permutations of ghost cells asked for (i.e. fill the ghost cells corresponding to all spanning vectors of fillVec). For this routine, given a block distributed array, a processor may both send to and receive from multiple other processors.

Invoking this routine is equivalent to invoking *ghostFillSched()* once for each spanning vector, and merging all the returned schedules. For example, an array reference A[i+1][j-1][k+2] would require a fill vector of [1 -1 2] to fill all the required ghost cells. This routine fills in all the possible permutations of ghost cells asked for (e.g. all face, edge and corner ghost cells of the part of the three dimensional distributed array on a given processor are filled). For this example, the spanning vectors are [1 -1 0], [1 0 2] and [0 -1 2] for the faces, [1 0 0], [0 -1 0] and [0 0 2] for the edges, and [1 -1 2] for the corner.

Synopsis

SCHED *ghostFillSpanSched(dArrayPtr, ndims, fillVec)

Parameter declarations

DARRAY *dArrayPtr pointer to the distributed array descriptor

int ndims the number of dimensions of the distributed array, and the length of fillVec

int fillVec[ndims] the offset vector describing the ghost cells to be filled. The magnitude of the offset vector in a dimension determines the number of ghost cells filled. A positive value in a dimension means high indices are filled and a negative value means low indices are filled.

Return value

pointer to a structure of type SCHED that describes the necessary data motion, which can be used by the *DataMove()* primitives

Example

Consider the same example as in Section 1.4.1. Off-processor fetches are required to satisfy the references to *w[i+1][j]*, *w[i][j+1]* and *w[i+1][j+1]* on the right hand side of the assignment. The following example shows how the three calls to *ghostFillSched()*, one for each reference, can be replaced by one call to *ghostFillSpanSched()*.

```
fillVec[0] = 1;  fillVec[1] = 1;
sched1 = ghostFillSched(da, 2, fillVec);
fDataMove(w, sched1, w);

for(j=lalbnd(da, 1, 1, 1); j<=laubnd(da, 1, size2-2, 1); i++){
  for(i=lalbnd(da, 0, 1, 1); i<=laubnd(da, 0, size1-2, 1); i++){
    deltaw[i][j]  = k * (w[i+1][j] - w[i][j]) +
                    l * (w[i][j+1] - w[i][j]) +
                    m * w[i+1][j+1];
  }
}
```

1.4.3 ghostFillAllSched()

ghostFillAllSched() computes a schedule describing the data motion necessary to fill in all of the overlap/ghost cells for a distributed array (as declared with the call to *align()* that returned "dArrayPtr"). The physical characteristics and distribution of the array (i.e. number of dimensions, local dimension size, number of ghost cells in each dimension, distribution in each dimension, etc.) are described by the data structure pointed to by the parameter "dArrayPtr". For this routine, given a block distributed array, a processor may both send to and receive from multiple other processors.

Invoking this routine is equivalent to invoking *ghostFillSched()* once for each ghost cell region, and merging all the returned schedules. For example, a two dimensional distributed array has eight such regions, corresponding to the edges and corners of the two dimensional part of the array assigned to each processor. Assuming that the distributed array has been declared with one internal ghost cell in each dimension, the fill vector arguments for *ghostFillSched()* for all the regions are [1 0], [0 1], [-1 0], [0 -1] for the edges and [1 1], [1 -1], [-1 1], [-1 -1] for the corners.

Synopsis

SCHED *ghostFillAllSched(dArrayPtr)

Parameter declarations

DARRAY *dArrayPtr pointer to the distributed array descriptor

Return value

pointer to a structure of type SCHED that describes the necessary data motion, which can be used by the *DataMove()* primitives

1.4.4 exchSched()

exchSched() computes a schedule describing the data motion necessary to update (from other processors) internal ghost cells along the given dimension of the distributed array. The physical characteristics and distribution of the array (i.e. number of dimensions, local dimension size, number of ghost cells in each dimension, distribution in each dimension, etc.) are described by the data structure pointed to by the parameter "dArrayPtr". The parameter "fill" controls the number of ghost cells being updated and whether these cells are filled at the lower or upper indices of the given array dimension. A call *exchSched(dA, dim, fill)* is equivalent to a call to *ghostFillSched()* with a fill vector of all zeros, except in dimension "dim", which would have the value "fill".

Synopsis

SCHED *exchSched(dArrayPtr, dim, fill)

Parameter declarations

DARRAY *dArrayPtr pointer to the distributed array descriptor

int dim the dimension of the distributed array whose ghost cells are being updated. Numbering follows the C convention (i.e. 0 to N-1 where N = number of dimensions)

int fill the number of ghost cells being updated. A positive value means high indices and a negative value means low indices are filled.

Return value

pointer to a structure of type SCHED that describes the necessary data motion, which can be used by the *DataMove()* primitives

Example

Consider the same example as in Section 1.4.1. Off-processor fetches are required to satisfy the references to *w[i+1][j]*, *w[i][j+1]* and *w[i+1][j+1]* on the right hand side of the assignment. The fetches for the first two references, to *w[i+1][j]* and *w[i][j+1]*, could be satisfied by calls to *exchSched()*, as shown below. *fDataMove()*, which is discussed in detail in Section 1.4.6, uses a schedule produced by *exchSched()* to perform the actual communication.

```
sched1 = exchSched(da, 0, 1);
sched2 = exchSched(da, 1, 1);
fillVec[0] = 1;  fillVec[1] = 1;
sched3 = ghostFillSched(da, 2, fillVec);
fDataMove(w, sched1, w);
fDataMove(w, sched2, w);
fDataMove(w, sched3, w);

for(j=lalbnd(da, 1, 1, 1); j<=laubnd(da, 1, size2-2, 1); i++){
  for(i=lalbnd(da, 0, 1, 1); i<=laubnd(da, 0, size1-2, 1); i++){
    deltaw[i][j]  = k * (w[i+1][j] - w[i][j]) +
                    l * (w[i][j+1] - w[i][j]) +
                    m * w[i+1][j+1];
  }
}
```

1.4.5 subArraySched()

subArraySched() creates a schedule describing the data motion necessary to move an arbitrary multi-dimensional chunk of a source distributed array to a multi-dimensional chunk of a destination distributed array, allowing data rotation and strides. The indices of the source and destination arrays should be given in global terms, as in the corresponding sequential program. The strides must be greater than zero. In a given dimension of either the source or destination array, if the range start index is greater than the end index, the stride counts backwards through the indices. The source and destination arrays are not required to have

the same number of dimensions. If the arrays do not have the same number of dimensions, then the extra dimensions of the array with fewer dimensions should specify -1 as the dimension number, and the values of the other parameters for the array in that dimension are ignored. For example, if the source array has two dimensions and the destination array has three dimensions, then srcDims[2] should be -1 (and srcLos[2], srcHis[2], srcStrides[2] don't matter), and the dimension specified by destDims[2] should contain one element (i.e. destLos[2] = destHis[2]).

Synopsis

SCHED *subArraySched(srcDArray, destDArray, numDims,
srcDims, srcLos, srcHis, srcStrides,
destDims, destLos, destHis, destStrides)

Parameter Declarations

DARRAY *srcDArray pointer to the distributed array descriptor describing the source

DARRAY *destDArray pointer to the distributed array descriptor describing the destination

int numDims the number of dimensions in the source and destination arrays

int srcDims[numDims] dimension numbers in the source

int srcLos[numDims] start of the range in the source for each dimension

int srcHis[numDims] end of the range in the source for each dimension

int srcStrides[numDims] stride in the source for each dimension

int destDims[numDims] dimension numbers in the destination

int destLos[numDims] start of the range in the destination for each dimension

int destHis[numDims] end of the range in the destination for each dimension

int destStrides[numDims] stride in the destination for each dimension

Return Value

pointer to a structure of type SCHED that describes the necessary data motion, which can be used by the *DataMove()* primitives

Example

Consider the routine *copySegment()* shown in Figure 2. Assume the distributed array descriptors *sDA* and *dDA* describe the distribution of *src* and *dest*, respectively, and are passed to *copySegment()*, along with the rest of the parameters shown in Figure 2 (sVal, sLo, sHi, dVal, dLo, dHi, etc.). The following example shows how the copies to and from *tmp* are replaced by calls to the primitives *exchSched()* and *DataMove()*.

These primitives perform the communication necessary to copy the source segment to the destination segment. *fDataMove()*, which is discussed in detail in Section 1.4.6, uses the schedule produced by *subArraySched()* to perform the actual communication.

```
numdims = 2;
sDims[0] = 0;          sDims[1] = 1;
sStrides[0] = 1;       sStrides[1] = 1;
if (sDim == 0) {
  sLos[0] = sVal;      sHis[0] = sVal;
  sLos[1] = sLo;       sHis[1] = sHi;
}
else {
  sLos[0] = sLo;       sHis[0] = sHi;
  sLos[1] = sVal;      sHis[1] = sVal;
}

dDims[0] = 0;          dDims[1] = 1;
dStrides[0] = 1;       dStrides[1] = 1;
if (dDim == 0) {
  dLos[0] = dVal;      dHis[0] = dVal;
  dLos[1] = dLo;       dHis[1] = dHi;
}
else {
  dLos[0] = dLo;       dHis[0] = dHi;
  dLos[1] = dVal;      dHis[1] = dVal;
}

sched = subArraySched(sDA, dDA, numdims,
                      sDims, sLos, sHis, sStrides,
                      dDims, dLos, dHis, dStrides);

fDataMove(src,sched,dest);
```

1.4.6 dDataMove(), iDataMove(), fDataMove(), cDataMove()

The *DataMove()* primitives take a schedule produced by one of the *ghostFillSched()* variants or *subArraySched()*, along with pointers to the source data and the destination data (*not* to the distributed array descriptors created by *align()*) and perform the data motion. For the *DataMove()* routines to work properly, the storage allocated on a processor for its part of the distributed array must be contiguous (i.e. without holes), because the routines treat the local storage as a linear address space. For the same reason, the local storage must be the

"right" size for each processor. The correct local size can be obtained using the *laSizes()* routine described in Section 1.5.1.

Synopsis

void PREFIXDataMove(srcPtr, schedPtr, destPtr)
PREFIX can be d (double precision), i (integer) , f (floating point) or c (character).

Parameter Declarations

SCHED *schedPtr pointer to the schedule describing the required data motion

type *srcPtr pointer to the source data array

type *destPtr pointer to the destination data array

Return Value

none

Example

See the examples in Sections 1.4.4 and 1.4.5.

1.5 Memory Management Primitives

1.5.1 laSizes()

laSizes() computes the local size in each dimension for a distributed array described by "dArray" (including all ghost cells on the processor) and returns them in array "sizes". This routine is used for memory allocation (e.g. using *malloc()*), to provide the exact local size of a distributed array on a processor, including the locally owned portion of the array and both external and internal ghost cells.

Synopsis

void laSizes(dArray, sizes)

Parameter Declarations

DARRAY *dArray pointer to the distributed array descriptor

int sizes[numDims] the *local size* for each dimension of the array - numDims is the number of dimensions of the array

Return Value

none

1.5.2 free_sched()

free_sched() frees the storage allocated for a schedule by either *exchSched()* or *subArraySched()*. In addition, the schedule is removed from the hash table that stores schedules for reuse.

Synopsis

void free_sched(sched)

Parameter Declarations

SCHED *sched pointer to the schedule

Return Value

none

1.5.3 remove_exch_scheds()

remove_exch_scheds() frees the storage allocated for *all* schedules created by calls to *exchSched()*, either since the beginning of program execution or since the last call to *remove_exch_scheds()*. The routine also clears the hash table of all the freed schedules. After a call to this routine, pointers to schedules previously created by *exchSched()* are invalid.

Synopsis

void remove_exch_scheds()

Parameter Declarations

none

Return Value

none

1.5.4 remove_subarray_scheds()

remove_subarray_scheds() frees the storage allocated for *all* schedules created by calls to *subArraySched()*, either since the beginning of the program execution or since the last call to *remove_subarray_scheds()*. The routine also clears the hash table of all the freed schedules. After a call to this routine, pointers to schedules previously created by *subArraySched()* are invalid.

Synopsis

void remove_subarray_scheds()

Parameter Declarations

none

Return Value

none

1.6 Miscellaneous Primitives

1.6.1 gLBnd()

As explained in Section 1.3, when a dimension of an array is distributed over several processors each processor is responsible for a subrange of the indices in that dimension. *gLBnd()* computes and returns the lower bound of the range of locally stored global indices. *gLBnd()* returns an out of bounds index (e.g. -1) if the processor does not own part of the array.

Synopsis

int gLBnd(dArray, dim)

Parameter Declarations

DARRAY *dArray pointer to the distributed array descriptor

int dim the array dimension being queried

Return Value

an integer corresponding to the lower bound of the range of global indices stored locally

Example

Suppose the one dimensional array A, with distributed array descriptor dA, has 20 elements (numbered from 0 to 19) and is distributed by "block" over 4 processors (numbered from 0 to 3). A call *gLBnd(dA, 0)* from processor 0 would return the value 0 since this processor is responsible for global indices 0 to 4. Similarly, a call *gLBnd(dA, 0)* from processor 3 would return the value 15.

1.6.2 gUBnd()

gUBnd() is similar to *gLBnd()*, as described in Section 1.6.1. However, this primitive returns the upper bound of the range of locally stored global indices. *gUBnd()* returns an out of bounds index (e.g. -1) if the processor does not own part of the array.

Synopsis

int gUBnd(dArray, dim)

Parameter Declarations

DARRAY *dArray pointer to the distributed array descriptor

int dim the array dimension being queried

Return Value

an integer corresponding to the upper bound of the range of global indices stored locally

Example

As in Section 1.6.1, suppose array A, with distributed array descriptor dA, has 20 elements distributed by "block" over 4 processors, with no ghost cells. A call *gUBnd(dA, 0)* from processor 0 would return the value 4 since this processor is responsible for global indices 0 to 4. Similarly, a call *gUBnd(dA, 0)* from processor 3 would return the value 19.

1.6.3 globalToLocal()

globalToLocal() converts a global distributed array index in a given dimension into the corresponding local index in that dimension, returning -1 if the processor does not own that index.

Synopsis

int globalToLocal(dArray, gIndex, dim)

Parameter Declarations

DARRAY *dArray pointer to the distributed array descriptor

int gIndex the global index value whose local value is being computed

int dim the array dimension being queried

Return Value

an integer representing the local index value of gIndex, or -1 if the processor does not own the index

Example

Consider again the example from Section 1.6.1. The values of A[15] to A[19] are stored locally on processor 3 as A[0] to A[4]. The call *globalToLocal(dA, 16, 0)* on processor 3 converts global distributed array index 16 in dimension 0 into the local index 1. On any other processor (0, 1, or 2), the same call would return -1, indicating that the processor does not own that index.

1.6.4 globalToLocalWithGhost()

globalToLocalWithGhost() converts a global distributed array index in a given dimension into the corresponding local index in that dimension, returning -1 if the processor both does not own that index and does not have a copy of the global index as an internal ghost cell. Unlike *globalToLocal()*, this routine will return a local index corresponding to an internal ghost cell.

Synopsis

int globalToLocalWithGhost(dArray, gIndex, dim)

Parameter Declarations

DARRAY *dArray pointer to the distributed array descriptor

int gIndex the global index value whose local value is being computed

int dim the array dimension being queried

Return Value

an integer representing the local index value of gIndex, or -1 if the processor both does not own the index and does not have a copy of the global index as an internal ghost cell

Example

Consider again the example from Section 1.6.1, but assume that distributed array *A* has been specified to have one internal ghost cell (on both ends). The values of A[15] to A[19] are stored locally on processor 3 as A[1] to A[5]. In addition, a copy of the value of A[14] can be stored on processor 3 in internal ghost cell A[0]. The call *globalToLocalWithGhost(dA, 16, 0)* on processor 3 converts global distributed array index 16 in dimension 0 into the local index 2. On any other processor (0, 1, or 2), the same call would return -1, indicating that the processor neither owns that index nor has an internal ghost cell corresponding to that index. On the other hand, the call *globalToLocalWithGhost(dA, 15, 0)* on processor 3 returns local index 1, and on processor 2 returns local index 6 (the index of the high end internal ghost cell). On processors 0 and 1, the call returns -1.

1.6.5 localToGlobal()

localToGlobal() converts a local distributed array index in a given dimension into the corresponding global index in that dimension. If the local index does not correspond to a distributed array element (e.g. a ghost cell), *localToGlobal()* returns -1.

Synopsis

int localToGlobal(dArray, lIndex, dim)

Parameter Declarations

DARRAY *dArray pointer to the distributed array descriptor

int lIndex the local index value whose global value is being computed

int dim the array dimension being queried

Return Value

an integer representing the global index value of lIndex, or -1 if lIndex does not correspond to a distributed array element owned by the processor

Example

Again consider the example from Section 1.6.1. The values of A[15] to A[19] are stored locally on processor 3 as A[0] to A[4]. The call *localToGlobal(dA, 1, 0)* on processor 3 converts local distributed array index 1 in dimension 0 into the global index 16.

1.6.6 localToGlobalWithGhost()

localToGlobalWithGhost() converts a local distributed array index in a given dimension into the corresponding global index in that dimension. If the local index does not correspond to a distributed array element or an internal ghost cell (e.g. an out of range index), *localToGlobalWithGhost()* returns -1. Unlike *localToGlobal()*, this routine will return a global index for a local internal ghost cell.

Synopsis

int localToGlobalWithGhost(dArray, lIndex, dim)

Parameter Declarations

DARRAY *dArray pointer to the distributed array descriptor

int lIndex the local index value whose global value is being computed

int dim the array dimension being queried

Return Value

an integer representing the global index value of lIndex, or -1 if lIndex is not a valid local index

Example

Consider again the example from Section 1.6.1, but assume that distributed array *A* has been specified to have one internal ghost cell (on both ends). The values of A[15] to A[19] are stored locally on processor 3 as A[1] to A[5]. In addition, a copy of the value of A[14] can be stored on processor 3 in internal ghost cell A[0]. The call *localToGlobalWithGhost(dA, 2, 0)* on processor 3 converts local distributed array index 2 in dimension 0 into the global index 16. Similarly, the call *localToGlobalWithGhost(dA, 0, 0)* on processor 3 converts the index of the internal ghost cell into global index 14. However, on processor 0 the same call will return -1, since that internal ghost cell does not correspond to any global index.

2 Calling the primitives from Fortran

This section describes each of the Fortran multiblock primitives. The primitives are divided into five categories: declarations, loop-bound adjustment, communication, memory management, and a group of miscellaneous primitives. These primitives are described, respectively, in Sections 2.1, 2.2, 2.3, 2.4 and 2.5. To help in our explanation we will present a very simple example of a multiblock code that will be referred to in the following sections. Some of the Fortran primitives differ slightly in their calling sequences from the corresponding C version.

A Fortran version of the code presented in Section 1 is shown in Figures 4, 5, 6 and 7. We now present the Fortran version of the multiblock primitives.

2.1 Declaration Primitives

2.1.1 fvproc()

fvproc() creates a virtual processor space. This virtual processor space can be thought of as a type of "father" decomposition that encompasses the entire problem space and all available processors. The virtual processor space in dimension i is numbered from 1 to $sizes(i)$.

Synopsis

subroutine fvproc(vp, numDims, sizes)

Parameter Declarations

integer vp reference to the virtual processor space created

integer numDims number of dimensions in the virtual processor space. Currently only one dimensional virtual processor spaces are supported.

integer sizes(numDims) sizes of all the dimensions

Return Value

none

Example

Consider the initialization loop (loop 10) in Figure 4. We would like to create a virtual processor space representing the entire problem space (i.e. all blocks). The following code show how to use the sizes of the blocks to calculate the size of the entire problem space and calls *fvproc()* to create a corresponding virtual processor space.

```
 1       program main
 2 c
 3       parameter (mxbloc = 10)
 4       parameter (mxsize = 10000)
 5       integer    numBlocks, numIters
 6       integer    size1(mxbloc), size2(mxbloc),
 7       integer    bptr(mxbloc), imap(mxsize)
 8       dimension  w(mxsize), deltaw(mxsize)
 9 c
10       read (5,*)  numBlocks, numIters
11       do 10 ibloc=1, numBlocks
12          read (5,*)  size1(ibloc), size2(ibloc)
13  10   continue
14 c
15       call readMap(imap)
16       bptr(1) = 1
17       do 20 i = 2, numBlocks
18          bptr(i) = bptr(i-1) + (size1(i-1)+2)*(size2(i-1)+2)
19  20   continue
20 c
21       do 40 iter=1, numIters
22         call fillBCells(w, imap, bptr, iter, size1, size2)
23 c
24         do 30 ibloc=1, numBlocks
25           call compute(w(bptr(ibloc)), deltaw(bptr(ibloc)),
26      $                 size1(ibloc)+2, size2(ibloc)+2)
27  30     continue
28  40   continue
29       stop
30       end
```

Figure 4: Fortran multiblock code - main program

```
 1       subroutine fillBCells(w, imap, bptr, dBloc, size1, size2)
 2       integer imap(mxsize), bptr(mxbloc)
 3       integer dBloc, size1(mxbloc), size2(mxbloc)
 4       dimension w(mxsize)
 5 c
 6       nsegs = imap(1)
 7       i = 2
 8       do 50 iseg = 1,nsegs
 9         dDim  = imap(i)
10         dVal  = imap(i+1)
11         dLo   = imap(i+2)
12         dHi   = imap(i+3)
13         sDim  = imap(i+4)
14         sVal  = imap(i+5)
15         sLo   = imap(i+6)
16         sHi   = imap(i+7)
17         sBloc = imap(i+8)
18         call copySegment(
19      $   w(bptr(dBloc)),dDim,dVal,dLo,dHi,size1(dBloc),size2(dBloc),
20      $   w(bptr(sBloc)),sDim,sVal,sLo,sHi,size1(sBloc),size2(sBloc))
21         i = i+9
22  50   continue
```

Figure 5: Fortran multiblock code - fillBCells subroutine

```
 1       subroutine copySegment(dest,dDim,dVal,dLo,dHi,dSize1,dSize2
 2      $                       src, sDim,sVal,sLo,sHi,sSize1,sSize2)
 3       integer   dDim, dVal, dLo, dHi, dSize1, dSize2
 4       integer   sDim, sVal, sLo, sHi, sSize1, sSize2
 5       dimension src(sSize1,sSize2), dest(dSize1,dSize2), tmp(100)
 6 c
 7 c     copy src to tmp
 8       cnt = 1
 9       if (sDim .eq. 1) then
10         do 60 j=sLo, sHi
11           tmp(cnt) = src(sVal,j)
12           cnt = cnt + 1
13  60     continue
14       else
15         do 70 i=sLo, sHi
16           tmp(cnt) = src(i,sVal)
17           cnt = cnt + 1
18  70     continue
19       endif
20 c
21 c     copy tmp to dest
22       cnt = 1
23       if (dDim .eq. 1) then
24         do 80 j=dLo, dHi
25           dest(dVal,j) = tmp(cnt)
26           cnt = cnt + 1
27  80     continue
28       else
29         do 90 i=dLo, dHi
30           dest(i,dVal) = tmp(cnt)
31           cnt = cnt + 1
32  90     continue
33       endif
```

Figure 6: Fortran multiblock code - copySegment subroutine

```
 1       subroutine compute(w, deltaw, size1, size2)
 2       dimension w(size1, size2), deltaw(size1, size2)
 3       integer   size1, size2
 4       integer   i, j, k, l, m
 5
 6       do 100 j=2, size2-1
 7       do 100 i=2, size1-1
 8         deltaw(i,j) = k * (w(i+1,j) - w(i,j)) +
 9      $                l * (w(i,j+1) - w(i,j)) +
10      $                m * w(i+1,j+1)
11  100  continue
12       end
```

Figure 7: Fortran multiblock code - compute subroutine

```
    totalSize(1) = 0
    do 10 ibloc=1, numBlocks
       read (5,*) size1(ibloc), size2(ibloc)
       totalSize(1) = totalSize(1) + (size1(ibloc) * size2(ibloc))
10  continue
    call fvproc(vp, 1, totalSize)
```

2.1.2 fdecomp()

fdecomp() creates a new decomposition with "numDims" dimensions. The size for each dimension is given by the array "sizes".

Synopsis

subroutine fdecomp(decomp, numDims, sizes)

Parameter Declarations

integer decomp reference to the decomposition created

integer numDims number of dimensions in decomposition

integer sizes(numDims) size of the decomposition in each dimension

Return Value

none

Example

The following example uses the code from Section 2.1.1 and shows how decompositions are created for each block.

```
    totalSize(1) = 0
    do 10 ibloc=1, numBlocks
       read (5,*) size1(ibloc), size2(ibloc)
       totalSize(1) = totalSize(1) + (size1(ibloc) * size2(ibloc))
       sizeinfo(1) = size1(ibloc)
       sizeinfo(2) = size2(ibloc)
       call fdecomp(decomp(ibloc), 2, sizeinfo)
10  continue
    call fvproc(vp, 1, totalSize)
```

2.1.3 fembed()

fembed() embeds a decomposition into a subset of the virtual processor space. This primitive allocates a subset of all available processors to the decomposition and decides the physical identity of these processors (i.e. processor number). This primitive should always be called before *fdistribute()*, which needs to know how many processors have been allocated to the decomposition. Currently, *fembed()* only works for one dimensional virtual processor spaces.

Synopsis

subroutine fembed(decomp, vproc, startPosn, endPosn)

Parameter Declarations

integer decomp reference to the decomposition

integer vproc reference to the virtual processor space (currently, must be one dimensional)

integer startPosn start position in the virtual processor space

integer endPosn end position in the virtual processor space

Return Value

none

Example

Assume a virtual processor space and a number of decompositions have already been created, as shown in the example from Section 2.1.2. The following example shows how the decomposition corresponding to each block is embedded into the virtual processor space.

```
    startPosn = 1
    do 20 i=1, numBlocks
       endPosn = startPosn + (size1(i) * size2(i)) - 1
       call fembed(decomp(i), vp, startPosn, endPosn)
       startPosn = endPosn + 1
20  continue
```

2.1.4 fdistribute()

fdistribute() allows the user to specify the type of distribution for each dimension of a specified decomposition. This routine determines how the dimensions of the decomposition are mapped to the set of virtual processors specified by *fembed()*. If *fembed()* has not been used to specify a set of virtual processors for the decomposition, *fdistribute()* assumes that the decomposition is mapped to *all* processors. While *fdistribute()* can, in general, deal with a broad range of distributions (e.g. block, cyclic, block cyclic, undistributed and irregular), currently only block and undistributed are implemented.

Synopsis

subroutine fdistribute(decomp, dist)

Parameter Declarations

integer decomp reference to the decomposition

integer dist an array of N integers , where N is the number of dimensions in the decomposition. Each integer can have one of the following 3 values

0 Undistributed

1 Block distribution

2 Cyclic distribution (not yet implemented)

Return Value

none

Example

Using the subset of code from Section 2.1.3, we add a call to *fdistribute()* in the loop to specify that the distribution for each decomposition is "block" in the first and second dimensions.

```
      startPosn = 1
      do 20 i=1, numBlocks
         endPosn = startPosn + (size1(i) * size2(i)) - 1
         call fembed(decomp(i), vp, startPosn, endPosn)
         startPosn = endPosn + 1
         dist(1) = 1
         dist(2) = 1
         call fdistribute(decomp(i), dist)
20    continue
```

2.1.5 ifsection()

Irregular block distributions can be handled with the *section()* call, which creates all the partitions for all the dimensions of an irregular block decomposition at once.

Synopsis

integer ifsection(decomp, dim, cnt, idx)

Parameter Declarations

integer decomp pointer to the decomposition

integer dim(ndims) the dimensions to split

integer cnt(ndims) the number of partitions to create in each dimension

integer idx(ndims)(cnt(ndims)) the limiting upper index of each partition for each dimension

Return Value

the number of blocks created

2.1.6 ifpartition()

Irregular block distributions can also be created with the *partition()* call, which is used to recursively partition each dimension, one at a time. This allows remaining, unpartitioned dimensions to be partitioned differently. For example, if the first dimension is partitioned into two sections, *partition()* can then be called to partition the second dimension of the first section into two sections and called again to partition the second dimension of the second section into three sections. The parameter *blk* is used to indicate which section (or block) of each dimension is to be partitioned on a particular call.

Synopsis

integer ifpartition(decomp, dim, blk, cnt, idx)

Parameter Declarations

integer decomp pointer to the decomposition

integer dim the dimension to split

integer blk(ndims) the block to split

integer cnt the number of partitions to create

integer idx(cnt) the limiting upper index of each partition

Return Value

the number of partitions created

2.1.7 ifalign()

ifalign() is used to map arrays onto a decomposition and create distributed array descriptors. The following parameters are associated with each array dimension: the size, the number of internal ghost cells, the number of external ghost cells at the beginning and end of each dimension, a flag for determining where to put extra data points in case the size of the dimension is not evenly divisible by the number of processors, and the decomposition dimension to which it is aligned. Internal ghost cells are those required because the data set is divided among multiple processors. External ghost cells are a part of the user-defined distributed array that are specific to the application. If a distributed array dimension will not be aligned to any decomposition dimension, then the decomposition dimension should be set to -1. This primitive creates a distributed array descriptor and stores it in a table. *ifalign()* returns a reference to the newly created distributed array descriptor. All Fortran primitives requiring distribution information (e.g. *ifexch_sched(), iflalbnd(), iflaubnd(),* etc.) use the reference to access the appropriate distributed array descriptor.

Synopsis

integer ifalign(decomp, numDims, sizes, int_gcells,
ext_gcells_left, ext_gcells_right, extra_flag, decompDims)

Parameter Declarations

integer decomp reference to the decomposition to which the array is being aligned

integer numDims number of dimensions of the array

integer sizes(numDims) the sizes of the array in each dimension (including external ghost cells)

integer int_gcells(numDims) the number of internal ghost cells in each dimension

integer ext_gcells_left(numDims) the number of external ghost cells at the beginning in each dimension

integer ext_gcells_right(numDims) the number of external ghost cells at the end in each dimension

integer extra_flag(numDims) where to put extra data items if the array size is not evenly divisible by number of processors assigned to a dimension

0 : default (same as option 4)

1 : All on leftmost processor (the lowest numbered processor in the dimension)

2 : All on rightmost processor (the highest numbered processor in the dimension)

3 : Split equally between leftmost and rightmost processors (if an odd number the extra one is on the *leftmost*)

4 : Split equally between leftmost and rightmost processors (if an odd number the extra one is on the *rightmost*)

integer decompDims(numDims) the decomposition dimensions to which array dimensions are aligned. decompDims(i) should be -1 if dimension i is not aligned to any dimension of the decomposition.

Return Value

reference to a distributed array descriptor that can be used as an argument to the schedule generating primitives described in Section 2.3

Example

Consider the example from Section 2.1.4. For each block we call *ifalign()*, which creates a distributed array descriptor for a two dimensional array whose size in each dimension is the size of the block plus two (one external ghost cell on both the left and the right in each dimension), and that has one internal ghost cell in each dimension. If the number of data items in a distributed array dimension is not divisible by the number of (physical) processors assigned to that dimension (by the *fdistribute()* primitive), the *extra_flag* for

each dimension is set so that any extra data items are split between the leftmost and rightmost processors of that dimension (with an extra data item possibly located on the rightmost processor).

```
      startPosn = 1
      do 20 ibloc=1, numBlocks
        endPosn = startPosn + (size1(ibloc) * size2(ibloc))
        call fembed(decomp(ibloc), vp, startPosn, endPosn)
        startPosn = endPosn + 1
        dist(1) = 1                    dist(2) = 1
        call fdistribute(decomp(ibloc), dist)
        numdims = 2
        sizeinfo(1) = size1(ibloc)+2   sizeinfo(2) = size2(ibloc)+2
        int_gcells(1) = 1              int_gcells(2) = 1
        ext_gcells_left(1)  = 1        ext_gcells_left(2) = 1
        ext_gcells_right(1) = 1        ext_gcells_right(2) = 1
        extra_flag(1) = 0              extra_flag(2) = 0
        decomp_dim(1) = 1              decomp_dim(2) = 2
        da(ibloc) = ifalign(decomp(ibloc), numdims, sizeinfo, int_gcells,
     $     ext_gcells_left, ext_gcells_right, extra_flag, decomp_dim)
 20   continue
```

2.2 Loop Bound Adjustment Primitives

fdistribute() assigns a range of indices to each processor for each dimension of a decomposition, and *ifalign()* then maps the elements of a distributed array to the decomposition elements. Therefore, when a loop traverses a range of indices for a distributed array, that range may be distributed over several processors. For each processor, the start and end of the local range being traversed depends on the portion of the array residing on the processor, on the global range being traversed and on the stride. The functions *iflalbnd()* and *iflaubnd()* compute and return the local start and stop index values for each processor, for the common situation in which

- the loop has a left hand side distributed array reference indexed by a loop iteration variable, and
- the "owner computes" rule is being used to assign computations to processors.

2.2.1 iflalbnd()

Synopsis

integer iflalbnd(darray, dim, start, stride)

Parameter Declarations

integer darray reference to the distributed array descriptor

integer dim array dimension being traversed. Numbering follows the Fortran convention (i.e. 1 to number of dimensions)

integer start the (global) start of the range being traversed

integer stride distance between elements, positive means counting up, negative means counting down

Return Value

an integer corresponding to the (local) start index of the range

Example

See the example in Section 2.2.2.

2.2.2 iflaubnd()

Synopsis

integer iflaubnd(darray, dim, stop, stride)

Parameter Declarations

integer darray reference to the distributed array descriptor

integer dim array dimension being traversed. Numbering follows the Fortran convention (i.e. 1 to number of dimensions)

integer stop the (global) end of the range being traversed

integer stride distance between elements, positive means counting up, negative means counting down

Return Value

an integer corresponding to the (local) last index of the range

Example

The following example shows how *iflalbnd()* and *iflaubnd()* are used to adjust the loop bounds in routine *compute()* from Figure 7. Assume that the distributed array descriptor *da* has already been created, for either *w* or *deltaw* (since they have the same alignment), as in the example from Section 2.1.7.

```
      do 100 j=iflalbnd(da, 2, 2, 1), iflaubnd(da, 2, size2-1, 1)
      do 100 i=iflalbnd(da, 1, 2, 1), iflaubnd(da, 1, size1-1, 1)
         deltaw(i,j) = .......
100   continue
```

2.3 Communication Primitives

Two types of primitives for building communication schedules are provided. The *ifghostfill_sched* variants and *ifexch_sched()* are used for updating the internal ghost cells for a distributed array, while *ifsubarray_sched()* is used to move regular sections between distributed arrays (or within the same distributed array). The primitives also save the schedules in hash tables, so that the same schedule does not have to be computed more than once. If the required schedule is already available in the hash table, the primitives return a reference to the existing schedule. Primitives for freeing the storage used by a schedule once it is no longer needed are described in Section 2.4.

2.3.1 ifghostfill_sched()

ifghostfill_sched() computes a schedule describing the data motion necessary to fill in a set of overlap/ghost cells for a distributed array with "ndims" dimensions. The parameter "fillVec", of length "ndims", describes the overlap cells to be filled, and represents the offsets in each dimension of a reference to the distributed array. This routine only fills in exactly the cells asked for, not all the possible permutations. For this routine, given a block distributed array, a processor sends to (at most) one other processor and receives from (at most) one other processor.

Synopsis

integer ifghostfill_sched(darray, ndims, fillVec)

Parameter declarations

integer darray reference to the distributed array descriptor

integer ndims the number of dimensions of the distributed array, and the length of fillVec

integer fillVec(ndims) the offset vector describing the ghost cells to be filled. The magnitude of the offset vector in a dimension determines the number of ghost cells filled. A positive value in a dimension means high indices are filled and a negative value means low indices are filled.

Return value

reference to a schedule that describes the necessary data motion, which can be used by the *fdata_move()* primitives

Example

Consider the assignment statement to array *deltaw* in line 8 from the routine *compute()* shown in Figure 7. Assume the distributed array descriptor has already been created, by the code shown in the example from Section 2.1.7. *da* is passed to *compute()* as a parameter and describes the distribution of *w* and *deltaw*. Off-processor fetches are required to satisfy the references to *w(i+1,j)*, *w(i,j+1)* and *w(i+1,j+1)* on the right hand side of the assignment. The following example shows how calls to the primitives *ifghostfill_sched()* and *ffdata_move()* perform the communication necessary to ensure that the appropriate values are placed in the ghost cells of *w*. *ffdata_move()*, which is discussed in detail in Section 2.3.6, uses a schedule produced by *ghostFillSched()* to perform the actual communication.

```
      fillVec(0) = 1;  fillVec(1) = 0;
      sched1 = ifghostfill_sched(da, 2, fillVec);
      fillVec(0) = 0;  fillVec(1) = 1;
      sched2 = ifghostfill_sched(da, 2, fillVec);
      fillVec(0) = 1;  fillVec(1) = 1;
      sched3 = ifghostfill_sched(da, 2, fillVec);
      ffdata_move(w, sched1, w);
      ffdata_move(w, sched2, w);
      ffdata_move(w, sched3, w);

      do 100 j=iflalbnd(da, 2, 2, 1), iflaubnd(da, 2, size2-1, 1)
      do 100 i=iflalbnd(da, 1, 2, 1), iflaubnd(da, 1, size1-1, 1)
        deltaw(i,j)  = k * (w(i+1,j) - w(i,j)) +
     $                 l * (w(i,j+1) - w(i,j))
     $                 m * w(i+1,j+1)
100   continue
```

2.3.2 ifghostfillspan_sched()

ifghostfillspan_sched() computes a schedule describing the data motion necessary to fill in a set of overlap/ghost cells for a distributed array with "ndims" dimensions. The parameter "fillVec", of length "ndims", describes the overlap cells to be filled, and represents the offsets in each dimension of a reference to the distributed array. This routine fills in all the possible

permutations of ghost cells asked for (i.e. fill the ghost cells corresponding to all spanning vectors of fillVec). For this routine, given a block distributed array, a processor may both send to and receive from multiple other processors.

Invoking this routine is equivalent to invoking *ifghostfill_sched()* once for each spanning vector, and merging all the returned schedules. For example, an array reference A(i+1,j-1,k+2) would require a fill vector of [1 -1 2] to fill all the required ghost cells. This routine fills in all the possible permutations of ghost cells asked for (e.g. all face, edge and corner ghost cells of the part of the three dimensional distributed array on a given processor are filled). For this example, the spanning vectors are [1 -1 0], [1 0 2] and [0 -1 2] for the faces, [1 0 0], [0 -1 0] and [0 0 2] for the edges, and [1 -1 2] for the corner.

Synopsis

integer ifghostfillspan_sched(darray, ndims, fillVec)

Parameter declarations

integer darray reference to the distributed array descriptor

integer ndims the number of dimensions of the distributed array, and the length of fillVec

integer fillVec(ndims) the offset vector describing the ghost cells to be filled. The magnitude of the offset vector in a dimension determines the number of ghost cells filled. A positive value in a dimension means high indices are filled and a negative value means low indices are filled.

Return value

reference to a schedule that describes the necessary data motion, which can be used by the *fdata_move()* primitives

Example

Consider the same example as in Section 2.3.1. Off-processor fetches are required to satisfy the references to *w(i+1,j)*, *w(i,j+1)* and *w(i+1,j+1)* on the right hand side of the assignment. The following example shows how the three calls to *ifghostfill_sched()*, one for each reference, can be replaced by one call to *ifghostfillspan_sched()*.

```
fillVec(0) = 1;  fillVec(1) = 1;
sched1 = ifghostfillspan_sched(da, 2, fillVec);
ffdata_move(w, sched1, w);

do 100 j=iflalbnd(da, 2, 2, 1), iflaubnd(da, 2, size2-1, 1)
do 100 i=iflalbnd(da, 1, 2, 1), iflaubnd(da, 1, size1-1, 1)
```

```
         deltaw(i,j)  = k * (w(i+1,j) - w(i,j)) +
     $                  l * (w(i,j+1) - w(i,j))
     $                  m * w(i+1,j+1)
100    continue
```

2.3.3 ifghostfillall_sched()

ifghostfillall_sched() computes a schedule describing the data motion necessary to fill in all of the overlap/ghost cells for a distributed array (as declared with the call to *ifalign()* that returned "darray"). The physical characteristics and distribution of the array (i.e. number of dimensions, local dimension size, number of ghost cells in each dimension, distribution in each dimension, etc.) are described by the data structure referred to by the parameter "darray". For this routine, given a block distributed array, a processor may both send to and receive from multiple other processors.

Invoking this routine is equivalent to invoking *ifghostfill_sched()* once for each ghost cell region, and merging all the returned schedules. For example, a two dimensional distributed array has eight such regions, corresponding to the edges and corners of the two dimensional part of the array assigned to each processor. Assuming that the distributed array has been declared with one internal ghost cell in each dimension, the fill vector arguments for *ifghostfill_sched()* for all the regions are [1 0], [0 1], [-1 0], [0 -1] for the edges and [1 1], [1 -1], [-1 1], [-1 -1] for the corners.

Synopsis

integer ifghostfillall_sched(darray)

Parameter declarations

integer darray reference to the distributed array descriptor

Return value

reference to a schedule that describes the necessary data motion, which can be used by the *fdata_move()* primitives

2.3.4 ifexch_sched()

ifexch_sched() computes a schedule describing the data motion necessary to update internal ghost cells along the given dimension of the distributed array. The physical characteristics

and distribution of the array (i.e. number of dimensions, local dimension size, number of ghost cells in each dimension, distribution in each dimension, etc.) are described by the data structure referred to by the parameter "darray". The parameter "fill" controls the number of ghost cells being updated in and whether these cells are filled at the lower or upper indices of the given array dimension. A call *ifexch_sched(dA, dim, fill)* is equivalent to a call to *ifghostfill_sched()* with a fill vector of all zeros, except in dimension "dim", which would have the value "fill".

Synopsis

integer ifexch_sched(darray, dim, fill)

Parameter declarations

integer darray reference to the distributed array descriptor

integer dim the dimension of the array whose ghost cells are being updated. Numbering follows the Fortran convention (i.e. 1 to number of dimensions)

integer fill the number of cells being updated. A positive value means high indices and a negative value means low indices are filled.

Return value

reference to a schedule that describes the necessary data motion, which can be used by the *fdata_move()* primitives

Example

Consider the same example as in Section 2.3.1. Off-processor fetches are required to satisfy the references to *w(i+1,j)*, *w(i,j+1)* and *w(i+1,j+1)* on the right hand side of the assignment. The fetches for the first two references, to *w(i+1,j)* and *w(i,j+1)*, could be satisfied by calls to *ifexch_sched()*, as shown below. *ffdata_move()*, which is discussed in detail in Section 2.3.6, uses a schedule produced by *ifexch_sched()* to perform the actual communication.

```
sched1 = ifexch_sched(da, 1, 1)
sched2 = ifexch_sched(da, 2, 1)
fillVec(0) = 1;  fillVec(1) = 1;
sched3 = ifghostfill_sched(da, 2, fillVec);
call ffdata_move(w, sched1, w)
call ffdata_move(w, sched2, w)
call ffdata_move(w, sched3, w)

do 100 j=iflalbnd(da, 2, 2, 1), iflaubnd(da, 2, size2-1, 1)
```

```
      do 100 i=iflalbnd(da, 1, 2, 1), iflaubnd(da, 1, size1-1, 1)
        deltaw(i,j)  = k * (w(i+1,j) - w(i,j)) +
     $                 l * (w(i,j+1) - w(i,j))
     $                 m * w(i+1,j+1)
100   continue
```

2.3.5 ifsubarray_sched()

ifsubarray_sched() creates a schedule describing the data motion necessary to move an arbitrary multi-dimensional chunk of a source distributed array to a multi-dimensional chunk of a destination distributed array, allowing data rotation and strides. The indices of the source and destination arrays should be given in global terms, as in the corresponding sequential program. The strides must be greater than zero. In a given dimension of either the source or destination array, if the range start index is greater than the end index, the stride counts backwards through the indices. The source and destination arrays are not required to have the same number of dimensions. If the arrays do not have the same number of dimensions, then the extra dimensions of the array with fewer dimensions should specify 0 as the dimension number, and the values of the other parameters for the array in that dimension are ignored. For example, if the source array has two dimensions and the destination array has three dimensions, then srcDims(3) should be 0 (and srcLos(3), srcHis(3), srcStrides(3) don't matter), and the dimension specified by destDims(3) should contain one element (i.e. destLos(3) = destHis(3)).

Synopsis

integer ifsubarray_sched(srcDA, destDA, numDims,
srcDims, srcLos, srcHis, srcStrides,
destDims, destLos, destHis, destStrides)

Parameter Declarations

integer srcDA reference to the source distributed array descriptor

integer destDA reference to the destination distributed array descriptor

integer numDims the number of dimensions in the source and destination arrays

integer srcDims(numDims) dimension numbers in the source

integer srcLos(numDims) start of the range in the source for each dimension

integer srcHis(numDims) end of the range in the source for each dimension

integer srcStrides(numDims) stride in the source for each dimension

integer destDims(numDims) dimension numbers in the destination

integer destLos(numDims) start of the range in the destination for each dimension

integer destHis(numDims) end of the range in the destination for each dimension

integer destStrides(numDims) stride in the destination for each dimension

Return Value

reference to a schedule describing the necessary data motion, which can be used by the *fdata_move()* primitives

Example

Consider the routine *copySegment()* shown in Figure 6. Assume the distributed array descriptors *srcDA* and *destDA* describe the distribution of *src* and *dest*, respectively, and are passed to *copySegment()*, along with the rest of the parameters shown in Figure 6 (sVal, sLo, sHi, dVal, dLo, dHi, etc.). The following example shows how the copy to and from *temp* are replaced by calls to the primitives *ifsubarray_sched()* and *ffdata_move()*. These primitives perform the communication necessary to copy the source segment to the destination segment. *ffdata_move()*, which is discussed in detail in Section 2.3.6, uses the schedule produced by *ifsubarray_sched()* to perform the actual communication.

```
numDims = 2
sDims(1) = 1            sDims(2) = 2
sStrides(1) = 1         sStrides(2) = 1
if (sDim .eq. 1) then
  sLos(1) = sVal        sHis(1) = sVal
  sLos(2) = sLo         sHis(2) = sHi
else
  sLos(2) = sVal        sHis(2) = sVal
  sLos(1) = sLo         sHis(1) = sHi
endif

dDims(1) = 1            dDims(2) = 2
dStrides(1) = 1         dStrides(2) = 1
if (dDim .eq. 1) then
  dLos(1) = dVal        dHis(1) = dVal
  dLos(2) = dLo         dHis(2) = dHi
else
  dLos(2) = dVal        dHis(2) = dVal
  dLos(1) = dLo         dHis(1) = dHi
endif

sched = ifsubarray_sched(srcDA, destDA, numDims,
```

```
$                          sDims, sLos, sHis, sStrides,
$                          dDims, dLos, dHis, dStrides)
 call ffdata_move(src, sched, dest)
```

2.3.6 fddata_move(), fidata_move(), ffdata_move(), fcdata_move()

The *fdata_move* primitives take a schedule produced by one of the *ifghostfill_sched()* variants or *ifsubarray_sched()*, along with references to the source data and the destination data (*not* to the distributed array descriptors created by *ifalign()*) and perform the data motion. For the *fdata_move()* routines to work properly, the storage allocated on a processor for its part of the distributed array must be contiguous (i.e. without holes), because the routines treat the local storage as a linear address space. For the same reason, the local storage must be the "right" size for each processor. The correct local size can be obtained using the *flasizes()* routine described in Section 2.4.1.

Synopsis

subroutine fPREFIXdata_move(srcArray, sched, destArray)
PREFIX can be d (double precision), i (integer) , f (floating point) or c (character).

Parameter Declarations

integer sched reference to the schedule describing the required data motion

integer srcArray reference to the source distributed array

integer destArray reference to the destination distributed array

Return Value

none

Example

See the examples in Sections 2.3.4 and 2.3.5.

2.4 Memory Management Primitives

2.4.1 flasizes()

flasizes() computes the local size of all dimensions for a distributed array (including all ghost cells on the processor) and returns them in array "sizes". This routine is used for memory

allocation (e.g. setting up block pointers into one large, statically allocated array, as in the example from Figure 4) and reshaping arrays (i.e. for passing arrays into subroutines). *flasizes()* provides the exact local size of a distributed array on a processor, including the locally owned portion of the array and both external and internal ghost cells.

Synopsis

subroutine flasizes(darray, sizes)

Parameter Declarations

integer darray reference to the distributed array descriptor

integer sizes(numDims) the *local size* for each dimension of the array - numDims is the number of dimensions of the array

Return Value

none

2.4.2 free_sched()

free_sched() frees the storage allocated for a schedule by either *ifexch_sched()* or *ifsubarray_sched()*. In addition, the schedule is removed from the hash table that stores schedules for reuse.

Synopsis

subroutine free_sched(sched)

Parameter Declarations

integer sched reference to the schedule

Return Value

none

2.4.3 remove_exch_scheds()

remove_exch_scheds() frees the storage allocated for *all* schedules created by calls to *ifexch_sched()*, either since the beginning of program execution or since the last call to *remove_exch_scheds()*. The routine also clears the hash table of all the freed schedules. After a call to this routine, references to schedules previously created by *ifexch_sched()* are invalid.

Synopsis

subroutine remove_exch_scheds()

Parameter Declarations

none

Return Value

none

2.4.4 remove_subarray_scheds()

remove_subarray_scheds() frees the storage allocated for *all* schedules created by calls to *ifsubarray_sched()*, either since the beginning of program execution or since the last call to *remove_subarray_scheds()*. The routine also clears the hash table of all the freed schedules. After a call to this routine, references to schedules previously created by *ifsubarray_sched()* are invalid.

Synopsis

subroutine remove_subarray_scheds()

Parameter Declarations

none

Return Value

none

2.5 Miscellaneous Primitives

2.5.1 ifglbnd()

As explained in Section 2.2, when a dimension of an array is distributed over several processors each processor is responsible for a subrange of the indices in that dimension. *ifglbnd()* computes and returns the lower bound of the range of locally stored global indices. *ifglbnd()* returns an out of bounds index (e.g. 0) if the processor does not own part of the array.

Synopsis

integer ifglbnd(darray, dim)

Parameter Declarations

integer darray reference to the distributed array descriptor

integer dim the array dimension being queried

Return Value

an integer corresponding to the lower bound of the range of global indices stored locally

Example

Suppose the one dimensional array A, with distributed array descriptor dA, has 20 elements (numbered from 1 to 20) and is distributed by "block" over 4 processors (numbered from 0 to 3). A call *ifglbnd(dA, 1)* from processor 0 would return the value 1 since this processor is responsible for global indices 1 to 5. Similarly, a call *ifglbnd(dA, 1)* from processor 3 would return the value 16.

2.5.2 ifgubnd()

ifgubnd() is similar to *ifglbnd()*, as described in Section 2.5.1. However, this primitive returns the upper bound of the range of locally stored global indices. *ifgubnd()* returns an out of bounds index (e.g. 0) if the processor does not own part of the array.

Synopsis

integer ifgubnd(darray, dim)

Parameter Declarations

integer darray reference to the distributed array descriptor

integer dim the array dimension being queried

Return Value

an integer corresponding to the upper bound of the range of global indices stored locally

Example

As in the example from Section 2.5.1, suppose the one dimensional array *A*, with distributed array descriptor *dA*, has 20 elements distributed by "block" over 4 processors, with no ghost cells. A call *ifgubnd(dA, 1)* from processor 0 would return the value 5 since this processor is responsible for global indices 1 to 5. Similarly, a call *ifgubnd(dA, 1)* from processor 3 would return the value 20.

2.5.3 ifglobal_to_local()

ifglobal_to_local() converts a global distributed array index in a given dimension into the corresponding local index in that dimension, returning -1 if the processor does not own that index.

Synopsis

integer ifglobal_to_local(darray, gindex, dim)

Parameter Declarations

integer darray reference to the distributed array descriptor

int gindex the global index value whose local value is being computed

int dim the array dimension being queried

Return Value

an integer representing the local index value of gindex, or -1 if the processor does not own the index

Example

Consider again the example from Section 2.5.1. The values of A(16) to A(20) are stored locally on processor 3 as A(1) to A(5). The call *ifglobal_to_local(dA, 17, 1)* on processor 3 converts global distributed array index 17 in dimension 1 into the local index 2. On any other processor (0, 1, or 2), the same call would return -1, indicating that the processor does not own that index.

2.5.4 ifglobal_to_local_with_ghost()

ifglobal_to_local_with_ghost() converts a global distributed array index in a given dimension into the corresponding local index in that dimension, returning -1 if the processor both does not own that index and does not have a copy of the global index as an internal ghost cell. Unlike *ifglobal_to_local()*, this routine will return a local index corresponding to an internal ghost cell.

Synopsis

integer ifglobal_to_local_with_ghost(darray, gindex, dim)

Parameter Declarations

integer darray reference to the distributed array descriptor

int gindex the global index value whose local value is being computed

int dim the array dimension being queried

Return Value

an integer representing the local index value of gindex, or -1 if the processor both does not own the index and does not have a copy of the global index as an internal ghost cell

Example

Consider again the example from Section 2.5.1, but assume that distributed array *A* has been specified to have one internal ghost cell (on both ends). . The values of A(16) to A(20) are stored locally on processor 3 as A(2) to A(6). In addition, a copy of the value of A(15) can be stored on processor 3 in internal ghost cell A(1). The call *ifglobal_to_local_with_ghost(dA, 17, 1)* on processor 3 converts global distributed array index 17 in dimension 1 into the local index 3. On any other processor (0, 1, or 2), the same call would return -1, indicating that the processor neither owns that index nor has an internal ghost cell corresponding to that index. On the other hand, the call *ifglobal_to_local_with_ghost(dA, 16, 1)* on processor 3 returns local index 2, and on processor 2 returns local index 7 (the index of the high end internal ghost cell). On processors 0 and 1, the call returns -1.

2.5.5 iflocal_to_global()

iflocal_to_global() converts a local distributed array index in a given dimension into the corresponding global index in that dimension. If the local index does not correspond to a distributed array element (e.g. a ghost cell), *iflocal_to_global()* returns -1.

Synopsis

integer if_local_to_global(darray, lindex, dim)

Parameter Declarations

integer darray reference to the distributed array descriptor

integer lindex the local index value whose global value is being computed.

integer dim the array dimension being queried

Return Value

an integer representing the global index value of lindex, or -1 if lindex does not correspond to a distributed array element owned by the processor

Example

Again consider the example from Section 2.5.1. The values of A(16) to A(20) are stored locally on processor 3 as A(1) to A(5). The call *iflocal_to_global(dA, 2, 1)* on processor 3 converts local distributed array index 2 in dimension 1 into the global index 17.

2.5.6 iflocal_to_global_with_ghost()

iflocal_to_global_with_ghost() converts a local distributed array index in a given dimension into the corresponding global index in that dimension. If the local index does not correspond to a distributed array element or an internal ghost cell (e.g. an out of range index), *iflocal_to_global_with_ghost()* returns -1. Unlike *iflocal_to_global()*, this routine will return a global index for a local internal ghost cell.

Synopsis

integer if_local_to_global_with_ghost(darray, lindex, dim)

Parameter Declarations

integer darray reference to the distributed array descriptor

integer lindex the local index value whose global value is being computed.

integer dim the array dimension being queried

Return Value

an integer representing the global index value of lindex, or -1 if lindex is not a valid local index

Example

Consider again the example from Section 2.5.1, but assume that distributed array *A* has been specified to have one internal ghost cell (on both ends). The values of A(16) to A(20) are stored locally on processor 3 as A(2) to A(6). The call *iflocal_to_global_with_ghost(dA, 3, 1)* on processor 3 converts local distributed array index 3 in dimension 1 into the global index 17. Similarly, the call *iflocal_to_global_with_ghost(dA, 1, 1)* on processor 3 converts the index of the internal ghost cell into global index 15. However, on processor 0 the same call will return -1, since that internal ghost cell does not correspond to any global index.

Acknowledgments

Kay Crowley did the initial implementation of the library at ICASE, NASA Langley Research Center. S. Gupta contributed to the redesign and reimplementation of many parts of the library, also at ICASE.