Virtual Topologies

Introduction

- Many computational science and engineering problems reduce at the end to either a series of matrix or some form of grid operations, be it through differential, integral or other methods. The dimensions of the matrices or grids are often determined by the physical problems.
- Frequently in multiprocessing, these matrices or grids are partitioned, or domain-decomposed, so that each partition (or subdomain) is assigned to a process.
- One such example is an *m x n* matrix decomposed into *p* *q x n* submatrices with each assigned to be worked on by one of the *p* processes.

Introduction

- In this case, each process represents one distinct submatrix in a straightforward manner. However, an algorithm might dictate that the matrix be decomposed into a *pxq* logical grid, whose elements are themselves each an *r x s* matrix. This requirement might be due to a number of reasons: efficiency considerations, ease in code implementation, code clarity, to name a few.
- Although it is still possible to refer to each of these *pxq* subdomains by a linear rank number, it is obvious that a mapping of the linear process rank to a 2D virtual rank numbering would facilitate a much clearer and natural computational representation.
- To address the needs of this and other topological layouts, the MPI library provides two types of topology routines: Cartesian and graph topologies. Only Cartesian topology and the associated routines will be discussed in this chapter.

MPI Topology Routines

Virtual Topology MPI Routines

- Some of the MPI topology routines are
  - MPI_CART_CREATE
  - MPI_CART_COORDS
  - MPI_CART_RANK
  - MPI_CART_SUB
  - MPI_CARTDIM_GET
  - MPI_CART_GET
  - MPI_CART_SHIFT
- These routines are discussed in the following sections.

MPI_CART_CREATE

- Definition of MPI_CART_CREATE
  - Used to create Cartesian coordinate structures, of arbitrary dimensions, on the processes. The new communicator receives no cached information.
- The MPI_CART_CREATE routine creates a new communicator using a Cartesian topology.

```
int MPI_Cart_create(MPI_Comm old_comm, int ndims,
    int *dim_size, int *periods, int reorder, MPI_Comm
    *new_comm)
```

- The function returns an int error flag.

MPI_CART_CREATE

Variable Name	C Type	In/Out	Description
old_comm	MPI_Comm	Input	Communicator handle
ndims	int	Input	Number of dimensions
dim_size	int *	Input	Array of size ndims providing length in each dimension
periods	int *	Input	Array of size ndims specifying periodicity status of each dimension
reorder	int	Input	whether process rank reordering by MPI is permitted
new_comm	MPI_Comm *	Output	Communicator handle

MPI_CART_CREATE

```
#include "stdio.h"
#include "mpi.h"
MPI_Comm old_comm, new_comm;
int ndims, reorder, periods[2], dim_size[2];

old_comm = MPI_COMM_WORLD;
ndims = 2;          /*  2D matrix/grid */
dim_size[0] = 3;    /* rows */
dim_size[1] = 2;    /* columns */
periods[0] = 1;     /* row periodic (each column forms a ring) */
periods[1] = 0;     /* columns non-periodic */
reorder = 1;        /* allows processes reordered for efficiency */

MPI_Cart_create(old_comm, ndims, dim_size, periods, reorder, &new_comm);
```

MPI_CART_CREATE

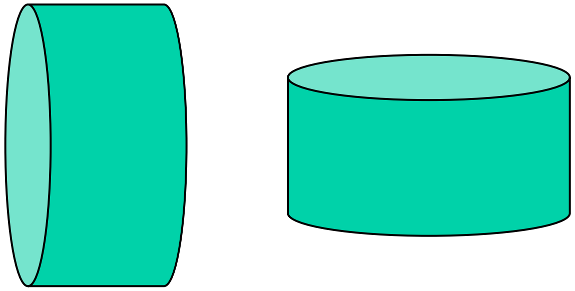
- In the above example we use MPI_CART_CREATE to map (or rename) 6 processes from a linear ordering (0,1,2,3,4,5) into a two-dimensional matrix ordering of 3 rows by 2 columns (*i.e.,* (0,0), (0,1), ..., (2,1)).
- Figure 8.1 (a) depicts the resulting Cartesian grid representation for the processes. The index pair "i,j" represent row "i" and column "j". The corresponding (linear) rank number is enclosed in parentheses.

0,0 (0)	*0,1 (1)*
1,0 (2)	*1,1 (3)*
2,0 (4)	*2,1 (5)*

Figure 8.1 (a). Cartesian Grid

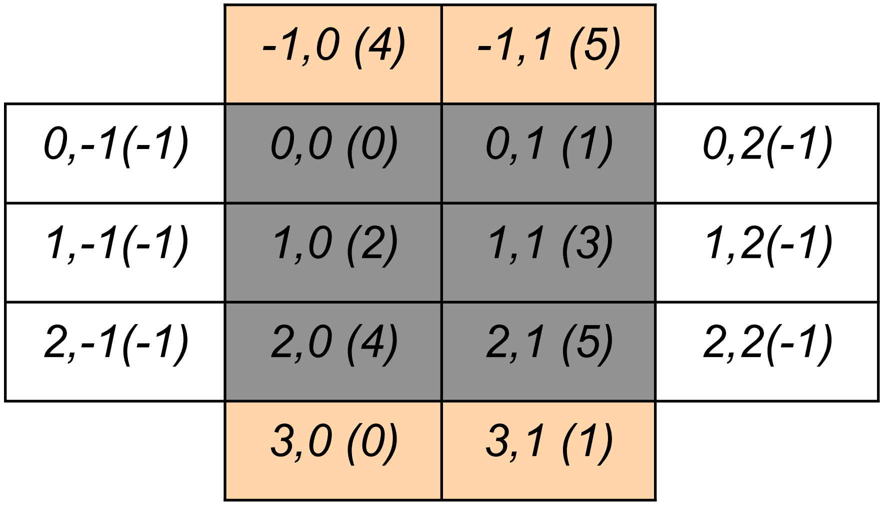
MPI_CART_CREATE

- With processes renamed in a 2D grid topology, we are able to assign or distribute work, or distinguish among the processes by their grid topology rather than by their linear process ranks.
- Additionally, we have imposed periodicity along the first dimension (periods[0]=1), which means that any reference beyond the first or last entry of any column will be wrapped around cyclically.
- For example, row index $i = -1$, due to periodicity, corresponds to $i = 2$. Similarly, $i = -2$ maps onto $i = 1$. Likewise, $i = 3$ is the same as $i = 0$.
- No periodicity is imposed on the second dimension (periods[1]=0). Any reference to the column index outside of its defined range (in this case 0 to 1) will result in a negative process rank (equal to MPI_PROC_NULL which is -1), which signifies that it is out of range.
- Similarly, if periodicity was defined only for the column index (i.e., periods[0]=0; periods[1]=1), each row would wrap around itself.



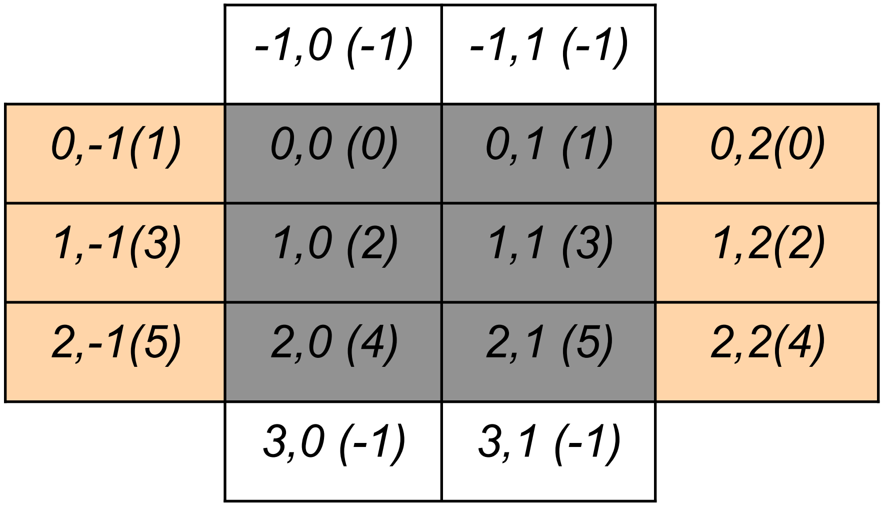
MPI_CART_CREATE

- Each of the above two 2D cases may be viewed graphically as a cylinder; the periodic dimension forms the circumferential surface while the non-periodic dimension runs parallel to the cylindrical axis.
- If both dimensions are periodic, the grid resembles a torus. The effects of periodic columns and periodic rows are depicted in Figures 8.1 (b) and (c), respectively. The tan-colored cells indicate cyclic boundary condition in effect.



	-1,0 (4)	-1,1 (5)	
0,-1(-1)	0,0 (0)	0,1 (1)	0,2(-1)
1,-1(-1)	1,0 (2)	1,1 (3)	1,2(-1)
2,-1(-1)	2,0 (4)	2,1 (5)	2,2(-1)
	3,0 (0)	3,1 (1)	

Figure 8.1 (b). periods[0]=1;periods[1]=0



	-1,0 (-1)	-1,1 (-1)	
0,-1(1)	0,0 (0)	0,1 (1)	0,2(0)
1,-1(3)	1,0 (2)	1,1 (3)	1,2(2)
2,-1(5)	2,0 (4)	2,1 (5)	2,2(4)
	3,0 (-1)	3,1 (-1)	

Figure 8.1 (c). periods[0]=0;periods[1]=1

MPI_CART_CREATE

- Finally, note that while the processes are arranged logically as a cartesian topology, the processors corresponding to these processes may in fact be scattered physically - even within a shared-memory machine.
  - If reorder is set to “1” in C, MPI may reorder the process ranks in the new communicator (for potential gain in performance due to, say, the physical proximities of the processes assigned).
  - If reorder is “0” in C, the process rank in the new communicator is identical to its rank in the old communicator.

MPI_CART_CREATE

- While having the processes laid out in the Cartesian topology help you write code that's conceivably more readable, many MPI routines recognize only rank number and hence knowing the relationship between ranks and Cartesian coordinates (as shown in the figures above) is the key to exploit the topology for computational advantages. In the following sections, we will discuss two subroutines that provide this information. They are
  - MPI_CART_COORDS
  - MPI_CART_RANK

MPI_CART_CREATE

- **Note:**
  - MPI_CART_CREATE is a collective communication function (see Chapter 6 - Collective Communications). It must be called by all processes in the group. Like other collective communication routines, MPI_CART_CREATE uses blocking communication. However, it is not required to be synchronized among processes in the group and hence is implementation dependent.
  - If the total size of the Cartesian grid is smaller than available processes, those processes not included in the new communicator will return MPI_COMM_NULL.
  - If the total size of the Cartesian grid is larger than available processes, the call results in error.

MPI_CART_COORDS

- Definition of MPI_CART_COORDS
  - Used to translate the coordinates of the process from rank, the inverse of MPI_CART_RANK.
- The MPI_CART_COORDS routine returns the corresponding Cartesian coordinates of a (linear) rank in a Cartesian communicator.

  int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
- The function returns an int error flag.

MPI_CART_COORDS

Variable Name	C Type	In/Out	Description
comm	MPI_Comm	Input	Communicator handle
rank	int	Input	Calling process rank
maxdims	int	Input	Number of dimensions in cartesian topology
coords	int *	Output	Corresponding cartesian coordinates of rank

MPI_CART_COORDS

```
MPI_Cart_create(old_comm, ndims, dim_size, periods,
   reorder, &new_comm);  /* creates communicator */

if(Iam == root) {   /* only want to do this on one process */
   for (rank=0; rank<p; rank++) {
        MPI_Cart_coords(new_comm, rank, ndims, &coords);
        printf("%d, %d\n ",rank, coords);
   }
}
```

MPI_CART_COORDS

- In the above example, a Cartesian communicator is created first.
- Repeated applications of MPI_CART_COORDS for all process ranks (input) produce the mapping table, shown in Figure 8.2, of process ranks and their corresponding Cartesian coordinates (output).

0,0 (0)	*0,1 (1)*
1,0 (2)	*1,1 (3)*
2,0 (4)	*2,1 (5)*

Figure 8.2. Cartesian Grid

MPI_CART_COORDS

- Note:
  - This routine is the reciprocal of MPI_CART_RANK.
  - Querying for coordinates of ranks in new_comm is not robust; querying for an out-of-range rank results in error.
- Definition of MPI_CART_RANK
  - Used to translate logical process coordinates to the ranks of the process in point-to-point routines.

MPI_CART_RANK

- Definition of MPI_CART_RANK
  - Used to translate logical process coordinates to the ranks of the process in point-to-point routines.
- The MPI_CART_RANK routine returns the corresponding process rank of the Cartesian coordinates of a Cartesian communicator.

```
int MPI_Cart_rank( MPI_Comm comm, int *coords, int
   *rank )
```

- The function returns an int error flag.

MPI_CART_RANK

Variable Name	C Type	In/Out	Description
comm	MPI_Comm	Input	Cartesian Communicator handle
coords	int *	Input	Array of size ndims specifying Cartesian coordinates
rank	int	Output	Process rank of process specified by its Cartesian coordinates, coords

MPI_CART_RANK

```
MPI_Cart_create(old_comm, ndims, dim_size, periods, reorder,
   &new_comm);

if(lam == root) {        /* only want to do this on one process */
   for (i=0; i<nv; i++) {
        for (j=0; j<mv; j++)  {
                coords[0] = i;
                coords[1] = j;
                MPI_Cart_rank(new_comm, coords, &rank);
                printf("%d, %d, %d\n",coords[0],coords[1],rank);
        }
   }
}
```

MPI_CART_RANK

- Once a Cartesian communicator has been established, repeated applications of MPI_CART_RANK for all possible values of the Cartesian coordinates produce a correlation table of the Cartesian coordinates and their corresponding process ranks.
- Shown in Figure 8.3 below is the resulting Cartesian topology (grid) where the index pair "i,j" represent row "i" and column "j". The number in parentheses represents the rank number associated with the Cartesian coordinates.

0,0 (0)	*0,1 (1)*
1,0 (2)	*1,1 (3)*
2,0 (4)	*2,1 (5)*

Figure 8.3. Cartesian Grid

MPI_CART_RANK

- **Note:**
  - This routine is the reciprocal of MPI_CART_COORDS.
  - Querying for rank number of out-of-range coordinates along the dimension in which periodicity is not enabled is not safe (i.e., results in error).

MPI_CART_SUB

- Definition of MPI_CART_SUB
  - Used to partition a communicator group into subgroups when MPI_CART_CREATE has been used to create a Cartesian topology.
- The MPI_CART_SUB routine creates new communicators for subgrids of up to (N-1) dimensions from an N-dimensional Cartesian grid.
- Often, after we have created a Cartesian grid, we wish to further group elements of this grid into subgrids of lower dimensions. Typical operations requiring subgrids include reduction operations such as the computation of row sums, column extremums.
  - For instance, the subgrids of a 2D Cartesian grid are 1D grids of the individual rows or columns. Similarly, for a 3D Cartesian grid, the subgrids can either be 2D or 1D.

```
int MPI_Cart_sub( MPI_Comm old_comm, int *belongs, MPI_Comm
    *new_comm )
```

- The function returns an int error flag.

MPI_CART_SUB

Variable Name	C Type	In/Out	Description
old_comm	MPI_Comm	Input	Cartesian Communicator handle
belongs	int *	Input	Array of size ndims specifying whether a dimension belongs to new_comm
new_comm	MPI_Comm	Output	Cartesian Communicator handle

MPI_CART_SUB

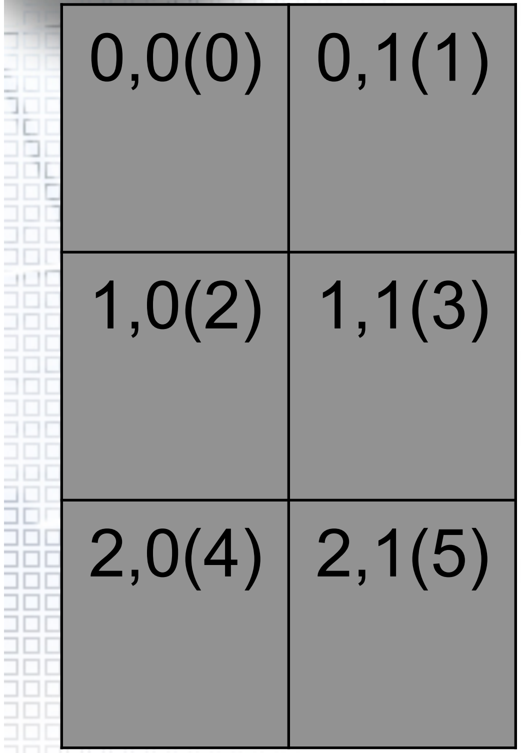
- For a 2D Cartesian grid, create subgrids of rows and columns. Create Cartesian topology for processes.

```
/* Create 2D Cartesian topology for processes */
MPI_Cart_create(MPI_COMM_WORLD, ndim, dims, period, reorder,
    &comm2D);
MPI_Comm_rank(comm2D, &id2D);
MPI_Cart_coords(comm2D, id2D, ndim, coords2D);
/* Create 1D row subgrids */
belongs[0] = 0;
belongs[1] = 1;     ! this dimension belongs to subgrid
MPI_Cart_sub(comm2D, belongs, &commrow);
/* Create 1D column subgrids */
belongs[0] = 1;      /* this dimension belongs to subgrid */
belongs[1] = 0;
MPI_Cart_sub(comm2D, belongs, &commcol);
```

MPI_CART_SUB

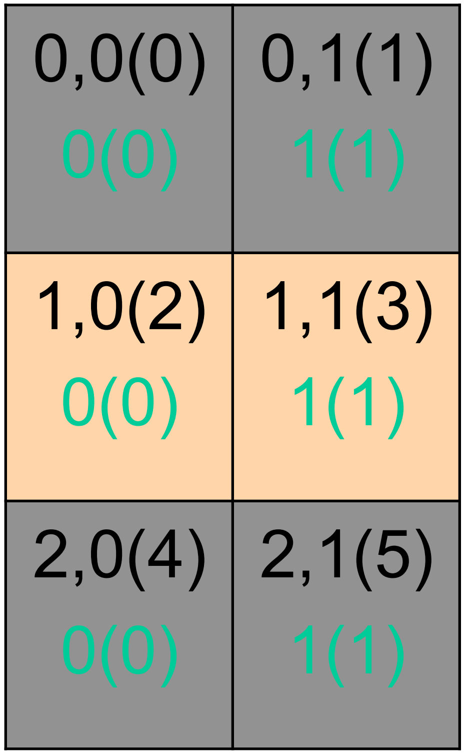
- Shown in Figure 8.4 (a) below is a 3-by-2 Cartesian topology. Figure 8.4 (b) shows the resulting row subgrids, while Figure 8.4 (c) shows the corresponding column subgrids. In black, the first row of numbers in each cell lists the 2D Cartesian coordinate index pair "i,j" and the associated rank number. On the second row, and in green, are shown the 1D subgrid Cartesian coordinates and the subgrid rank number (in parentheses). Their order is counted relative to their respective subgrid communicator groups.

MPI_CART_SUB



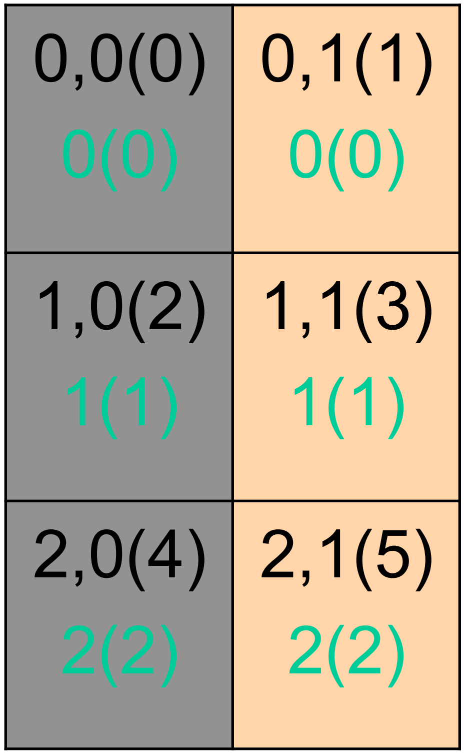
0,0(0)	0,1(1)
1,0(2)	1,1(3)
2,0(4)	2,1(5)

Figure 8.4 (a). 2D Cartesian Grid



0,0(0) 0(0)	0,1(1) 1(1)
1,0(2) 0(0)	1,1(3) 1(1)
2,0(4) 0(0)	2,1(5) 1(1)

Figure 8.4 (b). 3 Row Subgrids



0,0(0) 0(0)	0,1(1) 0(0)
1,0(2) 1(1)	1,1(3) 1(1)
2,0(4) 2(2)	2,1(5) 2(2)

Figure 8.4 (c). 2 Column Subgrids

MPI_CART_SUB

- Note:
  - MPI_CART_SUB is a collective routine. It must be called by all processes in old_comm.
  - MPI_CART_SUB generated subgrid communicators are derived from Cartesian grid created with MPI_CART_CREATE.
  - Full length of each dimension of the original Cartesian grid is used in the subgrids.
  - Each subgrid has a corresponding communicator. It inherits properties of the parent Cartesian grid; it remains a Cartesian grid.
  - It returns the communicator to which the calling process belongs.
  - There is a comparable MPI_COMM_SPLIT to perform similar function.
  - MPI_CARTDIM_GET and MPI_CART_GET can be used to acquire structural information of a grid (such as dimension, size, periodicity)

MPI_CART_SUB

- Definition of MPI_CART_CREATE
  - Used to create Cartesian coordinate structures, of arbitrary dimensions, on the processes. The new communicator receives no cached information.
- Definition of MPI_COMM_SPLIT
  - Used to partition old_comm into separate subgroups. It is similar to MPI_CART_CREATE.
- Definition of MPI_CART_GET
  - Used to retrieve the Cartesian topology previously cached with "comm".

MPI_CART_SUB Example

- This example demonstrates the usage of MPI_CART_SUB. We will work with six (6) processes.
- First, form a 2D (3x2) cartesian grid. Each element of this grid corresponds to one entry, A(i,j), of a matrix A. Furthermore, A(i,j) is defined as

  A(i,j) = (i+1)*10 + j + 1; i=0,1,2; j=0,1
- With this definition, A(0,0), for instance, has the value 11 while A(2,1) = 32.
- Next, create 2 column subgrids via MPI_CART_SUB. Each of the subgrids is a 3x1 vector. We then let the last member of each column (subgrid) to gather data, A(i,j), from their respective members.

MPI_CART_SUB Example

```
#include "stdio.h"
#include "mpi.h"
void main(int argc, char *argv[])
{
    int nrow, mcol, i, lastrow, p, root;
    int Iam, id2D, colID, ndim;
    int coords1D[2], coords2D[2], dims[2], aij[1], alocal[3];
    int belongs[2], periods[2], reorder;
    MPI_Comm comm2D, commcol;
    /* Starts MPI processes ... */
    MPI_Init(&argc, &argv);                                /* starts
    MPI */
    MPI_Comm_rank(MPI_COMM_WORLD, &Iam);          /* get current
    process id */
    MPI_Comm_size(MPI_COMM_WORLD, &p);            /* get number of
    processes */
```

MPI_CART_SUB Example

```
nrow = 3; mcol = 2; ndim = 2;
root = 0; periods[0] = 1; periods[1] = 0; reorder = 1;

/* create cartesian topology for processes */
dims[0] = nrow;           /* number of rows */
dims[1] = mcol;           /* number of columns */
MPI_Cart_create(MPI_COMM_WORLD, ndim, dims, periods, reorder, &comm2D);
MPI_Comm_rank(comm2D, &id2D);
MPI_Cart_coords(comm2D, id2D, ndim, coords2D);

/* Create 1D column subgrids */
belongs[0] = 1;           /* this dimension belongs to subgrid */
belongs[1] = 0;
MPI_Cart_sub(comm2D, belongs, &commcol);
MPI_Comm_rank(commcol, &colID);
MPI_Cart_coords(commcol, colID, 1, coords1D);
```

MPI_CART_SUB Example

```
MPI_Barrier(MPI_COMM_WORLD);

/* aij = (i+1)*10 + j + 1; 1 matrix element to each proc */
aij[0] = (coords2D[0]+1)*10 + coords2D[1]+1;

if(Iam == root) {
        printf("\n     MPI_Cart_sub example:");
        printf("\n 3x2 cartesian grid ==> 2 (3x1) column subgrids\n");
        printf("\n   Iam     2D       2D          1D       1D      aij");
        printf("\n  Rank   Rank     coords.     Rank  coords.\n");
}

/* Last element of each column gathers elements of its own column */
for ( i=0; i<=nrow-1; i++) {
        alocal[i] = -1;
}
```

MPI_CART_SUB Example

```
    lastrow = nrow - 1;
    MPI_Gather(aij, 1, MPI_INT, alocal, 1, MPI_INT, lastrow, commcol);

    MPI_Barrier(MPI_COMM_WORLD);

    printf("%6d|%6d|%6d %6d|%6d|%8d|",
    Iam,id2D,coords2D[0],coords2D[1],colID,coords1D[0]);
    for (i=0; i<=lastrow; i++) {
        printf("%6d ",alocal[i]);
    }
    printf("\n");

    MPI_Finalize();       /* let MPI finish up ...  */
}
```

MPI_CART_SUB Example

- Output

```
$ mpirun -np 6 ch08_cart_sub_example
MPI_Cart_sub example:
3x2 cartesian grid ==> 2 (3x1) column subgrids

  lam    2D        2D          1D      1D        aij
  Rank   Rank      coords.     Rank    coords.
  0|     0|      0     0|      0|       0|   -1   -1   -1
  1|     1|      0     1|      0|       0|   -1   -1   -1
  2|     2|      1     0|      1|       1|   -1   -1   -1
  3|     3|      1     1|      1|       1|   -1   -1   -1
  4|     4|      2     0|      2|       2|   11   21   31
  5|     5|      2     1|      2|       2|   12   22   32
```

MPI_CART_SUB Example

- Shown below are the two column subgrids resulting from the application of MPI_CART_SUB to a 3x2 cartesian grid.
- As before, the 2D cartesian grid coordinates are represented by the "i,j" pair of numbers, the rank numbers corresponding to the grid processes of the 2D grid are parenthesized and in black.
- The numbers below them, in green, are the rank numbers in the two respective column subgrids. The content of each element of the 2D grid is shown as $a_{i,j}$.

0,0 (0) $a_{0,0}(0)$	0,1 (1) $a_{0,1}(0)$
1,0 (2) $a_{1,0}(1)$	1,1 (3) $a_{1,1}(1)$
2,0 (4) $a_{2,0}(2)$	2,1 (5) $a_{2,1}(2)$

Figure a.
Column Subgrids

MPI_CARTDIM_GET

- Definition of MPI_CARTDIM_GET
  - An inquiry function used to determine the number of dimensions of the Cartesian structure.
- The MPI_CARTDIM_GET routine determines the number of dimensions of a subgrid communicator.
- On occasions, a subgrid communicator may be created in one routine and subsequently used in another routine. If the dimension of the subgrid is not available, it can be determined by MPI_CARTDIM_GET.

```
int MPI_Cartdim_get( MPI_Comm comm, int* ndims )
```

- The function returns an int error flag.

MPI_CARTDIM_GET

Variable Name	C Type	In/Out	Description
comm	MPI_Comm	Input	Cartesian communicator handle
ndims	int *	Output	Number of dimensions

```
/* create column subgrids */
belongs[0] = 1;
belongs[1] = 0;
MPI_Cart_sub(grid_comm, belongs, &col_comm);
/* queries number of dimensions of cartesan grid */
MPI_Cartdim_get(col_comm, &ndims);
```

MPI_CARTDIM_GET

- On occasions, detailed information about a grid may not be available, as in the case where a communicator is created in one routine and is used in another. In such a situation, MPI_CARTDIM_GET may be used to find the dimension of the grid associated with the communicator. Armed with this value, additional information may be obtained by calling MPI_CART_GET, which is discussed in the next section.
- Definition of MPI_CART_GET
  - Used to retrieve the Cartesian topology previously cached with "comm".

MPI_CART_GET

- Definition of MPI_CART_GET
  - Used to retrieve the Cartesian topology previously cached with "comm".
- Definition of MPI_CARTDIM_GET
  - An inquiry function used to determine the number of dimensions of the Cartesian structure.
- The MPI_CART_GET routine retrieves properties such as periodicity and size of a subgrid.
- On occasions, a subgrid communicator may be created in one routine and subsequently used in another routine. If only the communicator is available in the latter, this routine, along with MPI_CARTDIM_GET, may be used to determine the size and other pertinent information about the subgrid.

```
int MPI_Cart_get( MPI_Comm subgrid_comm, int ndims, int *dims, int
    *periods, int *coords )
```

- The function returns an int error flag.

MPI_CART_GET

Variable Name	C Type	In/Out	Description
subgrid_comm	MPI_Comm	Input	Communicator handle
ndims	int	Input	Number of dimensions
dims	int *	Output	Array of size ndims providing length in each dimension
periods	int *	Output	Array of size ndims specifying periodicity status of each dimension
coords	int *	Output	Array of size ndims providing Cartesian coordinates of calling process

MPI_CART_GET

```
/* create Cartesian topology for processes */
dims[0] = nrow;
dims[1] = mcol;
MPI_Cart_create(MPI_COMM_WORLD, ndim, dims, period, reorder,
   &grid_comm);
MPI_Comm_rank(grid_comm, &me);
MPI_Cart_coords(grid_comm, me, ndim, coords);
/* create row subgrids */
belongs[0] = 1;
belongs[1] = 0;
MPI_Cart_sub(grid_comm, belongs, &row_comm);
/* Retrieve subgrid dimensions and other info */
MPI_Cartdim_get(row_comm, &mdims);
MPI_Cart_get(row_comm, mdims, dims, period, row_coords);
```

MPI_CART_GET

- Shown in Figure 8.5 below is a 3-by-2 Cartesian topology (grid) where the index pair "i,j" represents row "i" and column "j". The number in parentheses represents the rank number associated with the Cartesian grid.
- This example demonstrated the use of MPI_CART_GET to retrieve information on a subgrid communicator. Often, MPI_CARTDIM_GET needs to be called first since ndims, the dimensions of the subgrid, is needed as input to MPI_CART_GET.

0,0 (0)	*0,1 (1)*
1,0 (2)	*1,1 (3)*
2,0 (4)	*2,1 (5)*

Figure 8.5. Cartesian Grid

MPI_CART_SHIFT

- Definition of MPI_CART_SHIFT
  - Used to return ranks of source and destination processes for a following call to MPI_SENDRCV that shifts data in a coordinate direction in the Cartesian communicator. Specified by the coordinate of the shift and by the size of the negative or positive shift step.
- The MPI_CART_SHIFT routine finds the resulting source and destination ranks, given a shift direction and amount.

```
int MPI_Cart_shift( MPI_Comm comm, int direction, int displ, int
    *source, int *dest )
```

- The function returns an int error flag.

MPI_CART_SHIFT

- Loosely speaking, MPI_CART_SHIFT is used to find two "nearby" neighbors of the calling process along a specific direction of an N-dimensional Cartesian topology.
- This direction is specified by the input argument, direction, to MPI_CART_SHIFT. The two neighbors are called "source" and "destination" ranks, and the proximity of these two neighbors to the calling process is determined by the input parameter displ.
  - If displ = 1, the neighbors are the two adjoining processes along the specified direction and the source is the process with the lower rank number, while the destination rank is the process with the higher rank.
  - On the other hand, if displ = -1, the reverse is true. A simple code fragment and a complete sample code are shown below to demonstrate the usage. A more practical example of an application is given in Section “*Iterative Solvers*”.

MPI_CART_SHIFT

Variable Name	C Type	In/Out	Description
comm	MPI_Comm	Input	Communicator handle
direction	int	Input	The dimension along which shift is to be in effect
displ	int	Input	Amount and sense of shift (<0; >0; or 0)
source	int *	Output	The source of shift (a rank number)
dest	int *	Output	The destination of shift (a rank number)

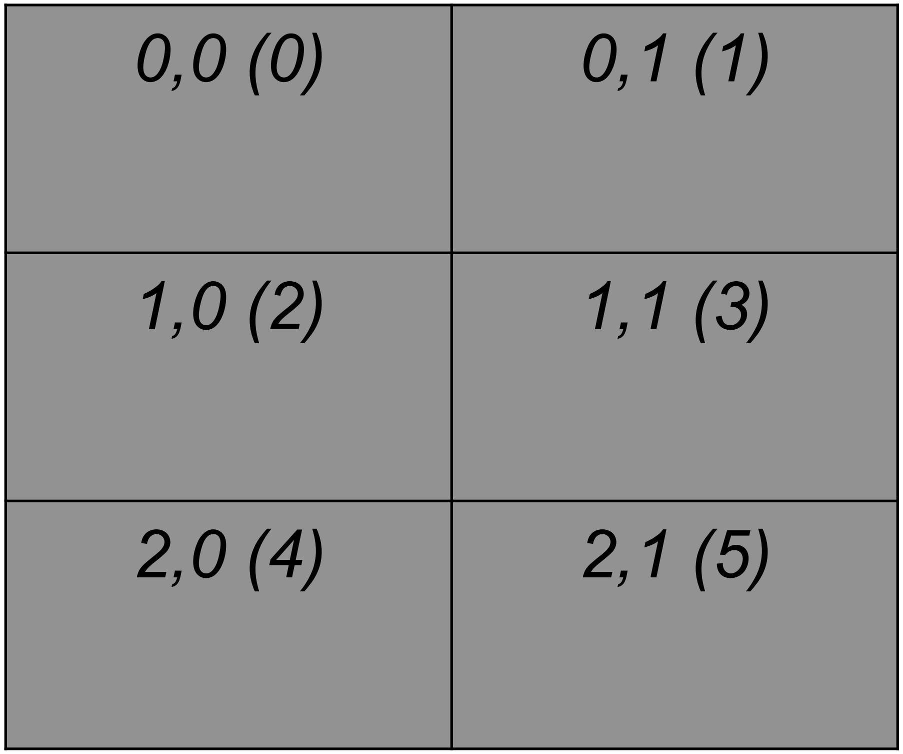
MPI_CART_SHIFT

```
/* create Cartesian topology for processes */
dims[0] = nrow;     /* number of rows     */
dims[1] = mcol;     /* number of columns  */
period[0] = 1;      /* cyclic in this direction */
period[1] = 0;      /* no cyclic in this direction */
MPI_Cart_create(MPI_COMM_WORLD, ndim, dims, period, reorder,
   &comm2D);
MPI_Comm_rank(comm2D, &me);
MPI_Cart_coords(comm2D, me, ndim, coords);

index =  0;    /* shift along the 1st index (out of 2) */
displ =  1;    /* shift by  1 */
MPI_Cart_shift(comm2D, index, displ, &source, &dest1);
```

MPI_CART_SHIFT

- In the above example, we demonstrate the application of MPI_CART_SHIFT to obtain the source and destination rank numbers of the calling process, me, resulting from shifting it along the first direction of the 2D Cartesian grid by one.
- Shown in Figure 8.6 below is a 3x2 Cartesian topology (grid) where the index pair "i,j" represent row "i" and column "j". The number in parentheses represents the rank number associated with the Cartesian coordinates.



0,0 (0)	*0,1 (1)*
1,0 (2)	*1,1 (3)*
2,0 (4)	*2,1 (5)*

Figure 8.6. Cartesian Grid

MPI_CART_SHIFT

- With the input as specified above and 2 as the calling process, the source and destination rank would be 0 and 4, respectively, as a result of the shift. Similarly, if the calling process were 1, the source rank would be 5 and destination rank would be 3. The source rank of 5 is the consequence of period(0) = 1. More examples are included in the sample code.

MPI_CART_SHIFT

- Note:
  - Direction, the Cartesian grid dimension index, has range (0, 1, ..., ndim-1). For a 2D grid, the two choices for direction are 0 and 1.
  - MPI_CART_SHIFT is a query function. No action results from its call.
  - A negative returned value (MPI_UNDEFINED) of source or destination signifies the respective value is out of bound. It also implies that there is no periodicity along that direction.
  - If periodic condition is enabled along the shift direction, an out of bound does not result. (See sample code).
- Definition of MPI_UNDEFINED
  - A flag value returned by MPI when an integer value to be returned by MPI is not meaningfully defined.

MPI_CART_SHIFT Example

- In this example, we demonstrate the usage of MPI_CART_SHIFT. With six (6) active processes, a 2D Cartesian topology is created for these 6 processes. This results in a 3x2 Cartesian topology representation for the 6 processes.
- Furthermore, a cyclic boundary condition is imposed down the rows -- but not the columns - of this 2D grid. Given the calling process rank number in the Cartesian grid communicator, upon calling MPI_CART_SHIFT the source and destination ranks of the calling process rank are returned as follows:

MPI_CART_SHIFT Example

1. along the rows and a displacement of +1, the source is the rank above the calling rank and the destination is the rank below it.
2. along the rows and a displacement of -2, the source is two ranks above the calling rank and the destination is two ranks below it.
3. along the rows and a displacement of +3, the source is three ranks above the calling rank and the destination is three rank below it.
4. across the columns and a displacement of +1, the source is the rank to the left of the calling rank and the destination is the rank to the right.
5. across the columns and a displacement of -1, the source is the rank to the right of the calling rank and the destination is the rank to the left.

MPI_CART_SHIFT Example

```
#include "stdio.h"
#include "mpi.h"
void main(int argc, char *argv[])
{
    int nrow, mcol, irow, jcol, lam, me, ndim;
    int p, ierr, root, direct, displ;
    int source1, source2, source3, source4, source5;
    int dest1, dest2, dest3, dest4, dest5;

    int coords[2], dims[2];
    int periods[2], reorder;
    MPI_Comm comm2D;

    /* Starts MPI processes ... */
    MPI_Init(&argc, &argv);                                    /* starts MPI */
    MPI_Comm_rank(MPI_COMM_WORLD, &lam);        /* get current process id */
    MPI_Comm_size(MPI_COMM_WORLD, &p);                     /* get number of
    processes */
```

MPI_CART_SHIFT Example

```
nrow = 3; mcol = 2; ndim = 2;
root = 0; periods[0] = 1; periods[1] = 0; reorder = 1;

if (Iam == root)
{
     printf("                (       along the rows        )(   across columns    )\n");
     printf("                 <== +1 ==>  <== -2 ==>  <== +3 ==>  <== +1 ==>  <== -1 ==>\n");
     printf("   2D   Row   Col  From    To  From    To  From    To  From    To  From    To\n");
     printf(" Rank     i     j   Src  Dest   Src  Dest   Src  Dest   Src  Dest   Src  Dest\n");
}
MPI_Barrier(MPI_COMM_WORLD);

/* create cartesian topology for processes */
dims[0] = nrow;              /* number of rows */
dims[1] = mcol;              /* number of columns */
MPI_Cart_create(MPI_COMM_WORLD, ndim, dims, periods, reorder, &comm2D);
MPI_Comm_rank(comm2D, &me);
MPI_Cart_coords(comm2D, me, ndim, coords);
```

MPI_CART_SHIFT Example

```
    direct = 0;          /* shift along the 1st direction (0; not 1) */
    displ = 1;           /* shift by  2 */
    MPI_Cart_shift(comm2D, direct, displ, &source1, &dest1);
    direct = 0;          /* shift along the 1st direction */
    displ = -2;          /* shift by -2 */
    MPI_Cart_shift(comm2D, direct, displ, &source2, &dest2);
    direct = 0;          /* shift along the 1st direction (0; not 1) */
    displ = 3;           /* shift by  3 */
    MPI_Cart_shift(comm2D, direct, displ, &source3, &dest3);
    direct = 1;          /* shift along the 2nd direction (1; not 2) */
    displ = 1;           /* shift by  1 */
    MPI_Cart_shift(comm2D, direct, displ, &source4, &dest4);
    direct = 1;          /* shift along the 2nd direction */
    displ = -1;          /* shift by -1 */
    MPI_Cart_shift(comm2D, direct, displ, &source5, &dest5);

    printf("%5d %5d %5d %5d %5d %5d %5d %5d %5d %5d %5d %5d %5d\n", me, coords[0],
    coords[1], source1, dest1, source2, dest2, source3, dest3, source4, dest4, source5, dest5);
    MPI_Finalize();    /* let MPI finish up ...  */
}
```

MPI_CART_SHIFT Example

- Output
  - Note that some of the returned ranks in the last four columns are negative because they are out of bounds and are assigned the value MPI_UNDEFINED. The value of MPI_UNDEFINED is implementation dependent. In this case, it is -1.

```
$ mpirun -np 6 ch08_cart_shift_example
                  (         along the rows           )(   across columns     )
                   <== +1 ==>  <== -2 ==>  <== +3 ==>  <== +1 ==>  <== -1 ==>
   2D   Row   Col  From    To  From    To  From    To  From    To  From    To
 Rank     i     j   Src  Dest   Src  Dest   Src  Dest   Src  Dest   Src  Dest
    0     0     0     4     2     4     2     0     0    -1     1     1    -1
    1     0     1     5     3     5     3     1     1     0    -1    -1     0
    2     1     0     0     4     0     4     2     2    -1     3     3    -1
    3     1     1     1     5     1     5     3     3     2    -1    -1     2
    4     2     0     2     0     2     0     4     4    -1     5     5    -1
    5     2     1     3     1     3     1     5     5     4    -1    -1     4
```

MPI_CART_SHIFT Example

– The 3x2 Cartesian topology grid, shown in Figure 8.7 below, illustrates what MPI_CART_SHIFT does under the four different sets of input parameters . The index pair "i,j" represents row "i" and column "j". The number in parentheses represents the rank number associated with the Cartesian coordinates.

	-3,0(0)	-3,0(1)	
	-2,0(2)	-2,1(3)	
	-1,0(4)	-1,1(5)	
0,-1(-1)	0,0(0)	0,1(1)	0,2(-1)
1,-1(-1)	1,0(2)	1,1,(3)	1,2(-1)
2,-1(-1)	2,0(4)	2,1(5)	2,2(-1)
	3,0(0)	3,1(1)	
	4,0(2)	4,1,(3)	
	5,0(4)	5,1(5)	

Figure 8.7.
periods[0]=1;periods[1]=0

Practical Applications

Practical Applications of Virtual Topologies

- The practical applications of virtual topologies listed below are discussed in the following sections.
  - Matrix Transposition
  - Iterative Solvers

Matrix Transposition

- This section demonstrates the use of virtual topologies by way of a matrix transposition. The matrix algebra for a matrix transposition is demonstrated in the following example.
- Consider a ***3 x 3*** matrix ***A***. This matrix is blocked into sub-matrices ***A*$_{11}$, *A*$_{12}$, *A*$_{21}$**, and ***A*$_{22}$** as follows:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

where

$$A_{11} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}; A_{12} = \begin{bmatrix} a_{13} \\ a_{23} \end{bmatrix}$$

$$A_{21} = \begin{bmatrix} a_{31} & a_{32} \end{bmatrix}; A_{22} = \begin{bmatrix} a_{33} \end{bmatrix}$$

Matrix Transposition

- Next, let ***B*** represent the transpose of ***A***.
- According to Equation on the right, the element $\boldsymbol{B_{ij}}$ is the blocked submatrix $\boldsymbol{A_{ji}^{T}}$. For instance,

$$B_{12} = A_{21}^{T} = \begin{bmatrix} a_{31} \\ a_{32} \end{bmatrix}$$

$$B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = A^{T}$$

$$= \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}^{T}$$

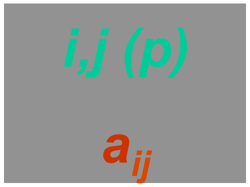
$$= \begin{bmatrix} A_{11}^{T} & A_{21}^{T} \\ A_{12}^{T} & A_{22}^{T} \end{bmatrix}$$

Matrix Transposition

- The parallel algorithm is
  - Select ***p*** and ***q*** such that the total number of processes, ***nprocs* = *p* x *q***.
  - Partition the ***n* x *m*** matrix into a (blocked) ***p* x *q*** matrix whose elements are themselves matrices of size ***(n/p)* x *(m/q)***.
  - Perform a transpose on each of these sub-matrices. These are performed serially because the entire sub-matrix resides locally on a process. No inter-process communication is required.
  - Formally, the ***p* x *q*** matrix needs to be transposed to obtain the final result. However, in reality this step is often not necessary. If you need to access the element (or sub-matrix) "***p,q***" of the transposed matrix, all you need to do is access the element "***q,p***", which has already been transposed locally. Depending on what comes next in the calculation, unnecessary message passing may be avoided.

Matrix Transposition

- As an example (see Figure 8.8), a ***9 x 4*** matrix with 6 processes is defined. Next, that matrix is mapped into a ***3 x 2*** virtual Cartesian grid, i.e., ***p=3, q=2***. Coincidentally, each element of this Cartesian grid is, in turn, a ***3 x 2*** matrix.
- For the physical grid, each square box represents one entry of the matrix. The pair of indices, "***i,j***", on the first row gives the global Cartesian coordinates, while "***(p)***" is the process associated with the virtual grid allocated by calling MPI_CART_CREATE or MPI_COMM_SPLIT. On the second row, ***a_{ij}***, is the value of the matrix element.
- The ***3 x 2*** virtual grid is depicted on the right of Figure 8.8. Each box in this grid represents one process and contains one ***3 x 2*** submatrix. Finally, another communicator is created for the transposed virtual grid with dimensions of ***2 x 3***. For instance, the element at "***1,0***" of the transposed virtual grid stores the value sent by the element at "***0,1***" of the virtual grid.



Matrix Transposition

0,0(0) 100	0,1(0) 101	0,2(1) 102	0,3(1) 103
1,0(0) 110	1,1(0) 111	1,2(1) 112	1,3(1) 113
2,0(0) 120	2,1(0) 121	2,2(1) 122	2,3(1) 123
3,0(2) 130	3,1(2) 131	3,2(3) 132	3,3(3) 133
4,0(2) 140	4,1(2) 141	4,2(3) 142	4,3(3) 143
5,0(2) 150	5,1(2) 151	5,2(3) 152	5,3(3) 153
6,0(4) 160	6,1(4) 161	6,2(5) 162	6,3(5) 163
7,0(4) 170	7,1(4) 171	7,2(5) 172	7,3(5) 173
8,0(4) 180	8,1(4) 181	8,2(5) 182	8,3(5) 183

Virtual Grid

0,0(0)	0,1(1)
1,0(2)	1,1(3)
2,0(4)	2,1(5)

Transposed Virtual Grid

0,0(0)	0,1(1)	0,2(2)
1,0(3)	1,1(4)	1,2(5)

Figure 8.8. An example of the use of virtual topologies by way of a matrix transposition using a ***9 x 4*** matrix with 6 processes.

Matrix Transposition

```
#include "stdio.h"
#include "mpi.h"     /* This brings in pre-defined MPI constants, ... */

void asemble(int at[][], int ml, int nl, MPI_Comm comm, int b[][], int m, int n, int p);

void main(int argc, char *argv[])
{
     int n, m, nv, nl, mv, ml, i, il, iv, j, jl, jv;
     int p, ndim, reorder, ierr;
     int master, me, Iam, source, dest, tag;
     int dims[2], coord[2];
     int period[2];
     int a[3][2], at[2][3], b[4][9];
     MPI_Status status;
     MPI_Request req;
     MPI_Comm grid_comm;
```

Matrix Transposition

```
/* Starts MPI processes ... */
MPI_Init(&argc, &argv);                                              /* starts MPI */
MPI_Comm_rank(MPI_COMM_WORLD, &lam); /* get current process id */
MPI_Comm_size(MPI_COMM_WORLD, &p);                /* get number of processes */

master = 0;              /* 0 is defined as the master processor */
period[0] = 0; period[1] = 0;        /* no cyclic boundary in either index */
tag = 0;  /* a tag is not required in this case, set it to zero */
dest = 0;                /* results are sent back to master */

n = 9; m = 4; nv = 3; mv = 2;
nl = n/nv; ml = m/mv;
ndim = 2; reorder = 1;

/* create cartesian topology for matrix */
dims[0] = nv;
dims[1] = mv;
MPI_Cart_create(MPI_COMM_WORLD, ndim, dims, period, reorder, &grid_comm);
MPI_Comm_rank(grid_comm, &me);
MPI_Cart_coords(grid_comm, me, ndim, coord);
iv = coord[0];
jv = coord[1];
```

Matrix Transposition

```
/* define local matrix according to virtual grid coordinates, (iv,jv) */
for (il=0; il<nl; il++)
{
        for (jl=0; jl<ml; jl++)
        {
                i = il + iv*nl;
                j = jl + jv*ml;
                a[il][jl] = i*10 + j;
        }
}

printf("%d: Before Transpose:\n", Iam);
for (i=0; i<nl; i++)
{
        for (j=0; j<ml; j++)
        {
                printf("%5d", a[i][j]);
        }
        printf("\n");
}
```

Matrix Transposition

```
    /* perform transpose on local matrix */
    for (il=0; il<nl; il++)
    {
            for (jl=0; jl<ml; jl++)
            {
                          at[jl][il] = a[il][jl];
            }
    }

    /* send "at" to Master for asembly and printing */
    MPI_Isend(at, ml*nl, MPI_INT, master, tag, grid_comm, &req);

    /* Master asembles all local transposes into final matrix and print */
    if(lam == master) {
            asemble(at, ml, nl, grid_comm, b, m, n, p);
            MPI_Wait(&req, &status);        /* make sure all sends done */
    }
    MPI_Finalize();                /* let MPI finish up ... */
}
```

Matrix Transposition

```
void asemble(int at[2][3], int ml, int nl, MPI_Comm comm, int b[4][9], int m, int n, int p)
{
    int tag, source, ierr, ndim;
    int iv, jv, i, j, il, jl, coord[2];
    MPI_Status status;
    tag = 0;

    /* The Master asembles the final (transposed) matrix from local copies and print */
    for (source=0; source<p; source++)
    {
          MPI_Cart_coords(comm, source, ndim, coord);
          MPI_Recv(at, ml*nl, MPI_INT, source, tag, comm, &status);
          iv = coord[0];
          jv = coord[1];
          for (jl=0; jl<nl; jl++)
          {
                    j = jl + iv*nl;                    /* swap iv and jv for transpose */
                    for (il=0; il<ml; il++)
                    {
                              i = il + jv*ml;
                              b[i][j] = at[il][jl];
                    }
          }
    }
```

Matrix Transposition

```
    printf("\nAfter Transpose:\n");
    for (i=0; i<m; i++)
    {
        for (j=0; j<n; j++)
        {
                printf("%5d", b[i][j]);
        }
        printf("\n");
    }
}
```

Iterative Solvers

- In this example, we demonstrate an application of the Cartesian topology by way of a simple elliptic (Laplace) equation.
- **Fundamentals:** The Laplace equation, along with prescribed boundary conditions, are introduced. Finite Difference Method is then applied to discretize the PDE to form an algebraic system of equations.
- **Jacobi Scheme:** A very simple iterative method, known as the Jacobi Scheme, is described. A single-process computer code is shown. This program is written in Fortran 90 for its concise but clear array representations. (Parallelism and other improvements will be added to this code as you progress through the example.)

Iterative Solvers

- **Parallel Jacobi Scheme:** A parallel algorithm for this problem is discussed. Simple MPI routines, without the invocations of Cartesian topology, are inserted into the basic, single-process code to form the parallel code.
- **SOR Scheme:** The Jacobi scheme, while simple and hence desirable for demonstration purposes, is impractical for "real" applications because of its slow convergence. Enhancements to the basic technique are introduced leading to the Successive Over Relaxation (SOR) scheme.
- **Parallel SOR Scheme:** With the introduction of a "red-black" algorithm, the parallel algorithm used for Jacobi is employed to parallelize the SOR scheme.
- **Scalability:** The performance of the code for a number of processes is shown to demonstrate its scalability.

Fundamentals

- First, some basics.
  Equation (1)

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

- where *u=u(x,y)* is an unknown scalar potential subjected to the following boundary conditions:

  Equation (2)

$$u\left(x,0\right) = \sin(nx) \qquad 0 \le \mathrm{x} \le 1$$

$$u\left(x,1\right) = \sin(nx)e^{-x} \qquad 0 \le \mathrm{x} \le 1$$

$$u\left(0,y\right) = u(1,y) = 0 \qquad 0 \le \mathrm{y} \le 1$$

Fundamentals

- Discretize the equation numerically with centered difference results in the algebraic equation

  Equation 3:

$$u_{i,j}^{n+1} \cong \frac{u_{i+1,j}^{n} + u_{i-1,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n}}{4} ; i = 1, \ldots, m; j = 1, \ldots, m$$

- where ***n*** and ***n+1*** denote the current and the next time step, respectively, while $u_{i-1,j}^{n}$ represents

  Equation 4:

$$\begin{aligned} u_{i-1,j}^{n} &= u^{n}\left(x_{i-1}, y_{j}\right); i = 1, \ldots, m; j = 1, \ldots, m \\ &= u^{n}\left((i-1)\Delta x, j\Delta y\right) \end{aligned}$$

Fundamentals

- and for simplicity, we take $\Delta x = \Delta y = \dfrac{1}{m+1}$

- Note that the analytical solution for this boundary value problem can easily be verified to be

  Equation (5):

  $$u(x, y) = \sin(\pi x)e^{-xy}; 0 \le x \le 1; 0 \le y \le 1$$

- and is shown below in a contour plot with x pointing from left to right and y going from bottom to top.

Fundamentals

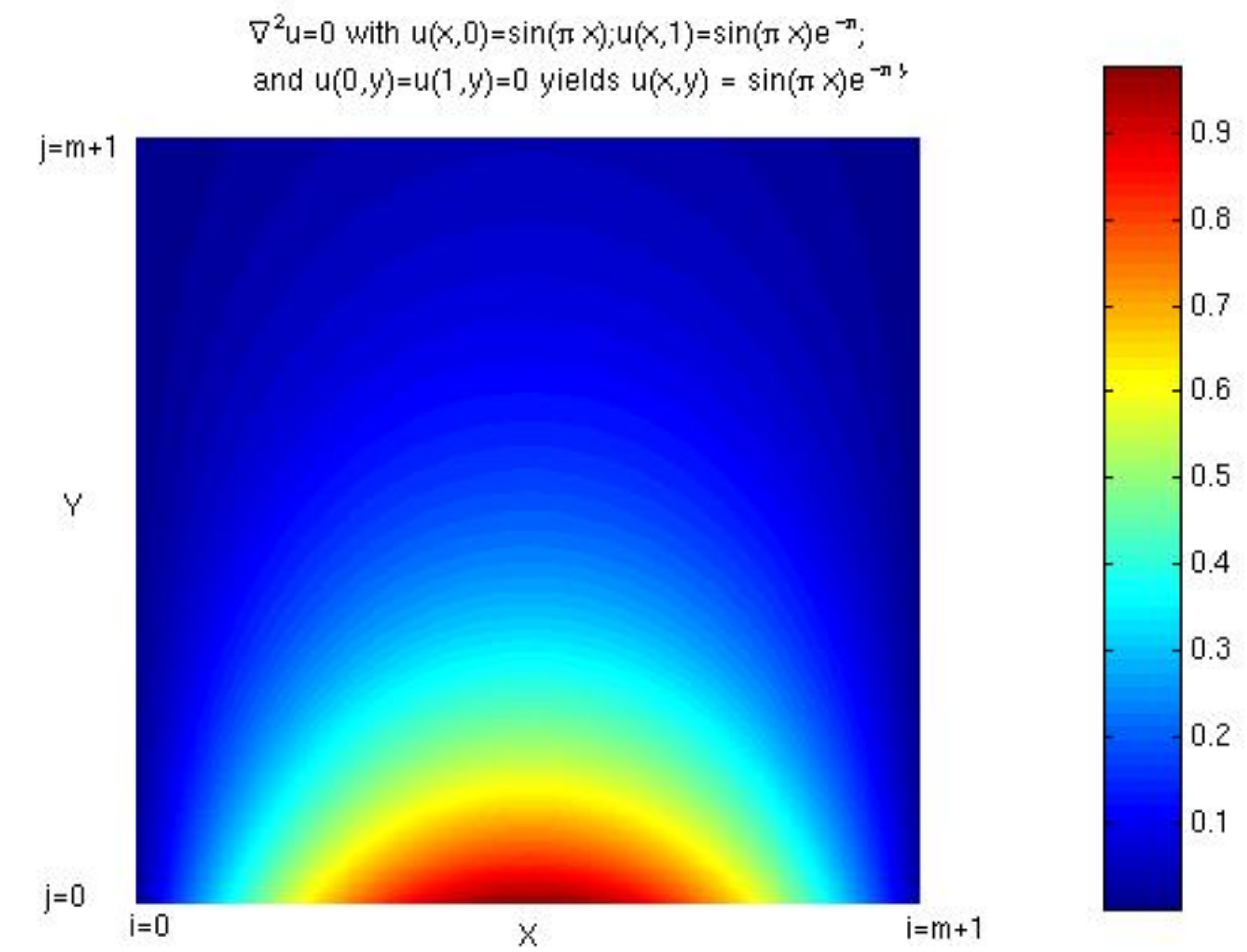


Figure 8.9. Contour plot showing the analytical solution for the boundary value problem.

Jacobi Scheme

- While numerical techniques abound to solve PDEs such as the Laplace equation, we will focus on the use of two iterative methods. These methods will be shown to be readily parallelizable, as well as lending themselves to the opportunity to apply MPI Cartesian topology introduced above. The simplest of iterative techniques is the Jacobi scheme, which may be stated as follows:

1. Make initial guess for $u_{i,j}$ at all interior points (i,j) for all $i=1:m$ and $j=1:m$.
2. Use Equation 3 to compute $u^{n+1}_{i,j}$ at all interior points (i,j).
3. Stop if the prescribed convergence threshold is reached, otherwise continue on to the next step.

4. $u^{n}_{i,j} = u^{n+1}_{i,j}$.

5. Go to Step 2.

Serial Jacobi Iterative Scheme

- A single-process implementation of the Jacobi Scheme as applied to the Laplace equation is given below. Note that
  - Program is written in C.
  - System size, m, is determined at run time.
  - Boundary conditions are handled by subroutine bc.
  - This scheme is very slow to converge and is not used in practice.
  - This example provides a starting point for later introduction of parallelization and convergence rate improvement concepts.

sjacobi.c

```
#include "solvers.h"

INT main() {
/********** MAIN PROGRAM ********************************
 * Solve Laplace equation using Jacobi iteration method      *
 * Kadin Tseng, Boston University, August, 2000               *
 *******************************************************/
    INT iter, m, mi, mp;
    REAL gdel;
    CHAR line[10];
    REAL **u, **un;

    fprintf(OUTPUT,"Enter size of interior points, mi :");
    (void) fgets(line, sizeof(line), stdin);
    (void) sscanf(line, "%d", &mi);
    fprintf(OUTPUT,"mi = %d\n",mi);
```

sjacobi.c

```
m = mi + 2;   /* interior points plus 2 b.c. points */
mp = m/P;

u      = allocate_2D(m, mp);  /* allocate mem for 2D array */
un = allocate_2D(m, mp);

gdel = 1.0;
iter = 0;

bc(m, mp, u, K, P); /* initialize and define B.C. for u */

replicate(m, mp, u, un);         /* u = un */
```

sjacobi.c

```
    while (gdel > TOL) {   /* iterate until error below threshold */
        iter++;                        /* increment iteration counter */

        if(iter > MAXSTEPS) {
                fprintf(OUTPUT,"Iteration terminated (exceeds %6d", MAXSTEPS);
                fprintf(OUTPUT," )\n");
                return (0);    /* nonconvergent solution */
        }
/* compute new solution according to the Jacobi scheme */
        update_jacobi(m, mp, u, un, &gdel);

        if(iter%INCREMENT == 0) {
                fprintf(OUTPUT,"iter,gdel: %6d, %lf\n",iter,gdel);
        }
    }

    fprintf(OUTPUT,"Stopped at iteration %d\n",iter);
    fprintf(OUTPUT,"The maximum error = %f\n",gdel);

/* write u to file for use in MATLAB plots */
    write_file( m, mp, u, K, P );

    return (0);
}
```

Jacobi and SOR Iterative Scheme Utility Functions

- The following includes solvers.h, utils.h and utils.c.

```
#ifndef _SOLVERS_H_INCLUDED_
#define _SOLVERS_H_INCLUDED_

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#define CHAR char
#define REAL double
#define INT  int

#define OUTPUT stdout       /* output to standard out                   */
#define PLOT_FILE "plots"   /* output files base name                   */
#define INCREMENT 100       /* number of steps between convergence check  */

#define P 1                 /* define processor count for serial codes    */
#define K 0                 /* current thread number for serial code is 0 */
#define MAX_M 512           /* maximum size of indices of Array u         */
#define MAXSTEPS 50000      /* Maximum number of iterations               */
#define TOL 0.000001        /* Numerical Tolerance */
#define PI 3.14159265       /* pi */

#include "utils.h"          /* header file of function prototype in utils.c */
#endif
```

Jacobi and SOR Iterative Scheme Utility Functions

```
#ifndef _UTILS_H_INCLUDED_
#define _UTILS_H_INCLUDED_

/* begin function prototyping  */

REAL **allocate_2D(int m, int n);
REAL my_max(REAL a, REAL b);
void init_array( INT m, INT n, REAL **a);
void bc( INT m, INT n, REAL **a, INT k, INT p );
void prtarray( INT nx, INT ny, REAL **a, FILE *fd);
INT write_file( INT m, INT n, REAL **u, INT k, INT p );
INT update_jacobi( INT m, INT n, REAL **u, REAL **unew, REAL *gdel);
INT update_sor( INT m, INT n, REAL **u, REAL omega, REAL *del, CHAR redblack);
INT replicate( INT m, INT n, REAL **u, REAL **ut );
INT transpose( INT m, INT n, REAL **u, REAL **ut );
void neighbors(INT k, INT p, INT UNDEFINED, INT *below, INT *above);

/* end function prototyping */

#endif
```

Jacobi and SOR Iterative Scheme Utility Functions

```
/********** U T I L I T Y **************************************
 * Utility functions for use with the Jacobi and SOR solvers          *
 * Kadin Tseng, Boston University, November 1999                      *
 ****************************************************************/
#include "solvers.h"
#include <malloc.h>

REAL **allocate_2D(INT m, INT n) {
    INT i;
    REAL **a;

    a = (REAL **) malloc((unsigned) m*sizeof(REAL*));

/* Each pointer array element points to beginning of a row with n entries*/
    for (i = 0; i < m; i++) {
          a[i] = (REAL *) malloc((unsigned) n*sizeof(REAL));
    }

    return a;
}
```

Jacobi and SOR Iterative Scheme Utility Functions

```
INT write_file( INT m, INT n, REAL **u, INT k, INT p ) {
/**************************************************
 * Writes 2D array ut columnwise (i.e. C convention)    *
 * m- size of rows                                      *
 * n - size of columns                                  *
 * u - scratch array                                    *
 * k - 0 <= k < p; = 0 for single thread code           *
 * p - p >= 0; =1 for single thread code                *
 **************************************************/
    INT ij, i, j, per_line;
    CHAR filename[50], file[53];
    FILE *fd;
/*
    prints u, 6 per line; used for matlab plots;
    PLOT_FILE contains the array size and number of procs;
    PLOT_FILE.(k+1) contains u pertaining to proc k;
    for serial job, PLOT_FILE.1 contains full u array.
*/

    (void) sprintf(filename, "%s", PLOT_FILE);
```

Jacobi and SOR Iterative Scheme Utility Functions

```
    if ( k == 0 ) {
        fd = fopen(filename, "w");
        fprintf(fd, "%5d %5d %5d\n", m, n, p);
        fclose(fd);
    }
    per_line = 6;       /* to print 6 per line */
    (void) sprintf(file, "%s.%d", filename, k); /* create output file */
    fd = fopen(file, "w");
    ij = 0;
    for (j = 0; j < n; j++) {
        for (i = 0; i < m; i++) {
         fprintf(fd, "%11.4f ", u[i][j]);
         if ((ij+1)%per_line == 0) fprintf(fd, "\n");
         ij++;
        }
    }
    fprintf(fd, "\n");
    fclose(fd);
    return (0);
}
```

Jacobi and SOR Iterative Scheme Utility Functions

```
void init_array(INT m, INT n, REAL **a) {
/********* Initialize Array **********************
 * Initialize array with nx rows and ny columns  *
 *************************************************/
    INT i, j;


    for (i = 0; i < m; i++) {
          for (j = 0; j < n; j++) {
                    a[i][j] = 0.0;                  /* initialize all entries to zero */
          }
    }
}
```

Jacobi and SOR Iterative Scheme Utility Functions

```
void bc(INT m, INT n, REAL **u, INT k, INT p) {
/********** Boundary Conditions ****************************************
 *   PDE: Laplacian u = 0;                        0<=x<=1;  0<=y<=1
                             *
 *   B.C.: u(x,0)=sin(pi*x); u(x,1)=sin(pi*x)*exp(-pi); u(0,y)=u(1,y)=0          *
 *   SOLUTION: u(x,y)=sin(pi*x)*exp(-pi*y)
                             *
 ***********************************************************************/
     INT i;

     init_array( m, n, u);
            /* initialize u to 0 */

     if (p > 1) {
           if (k == 0) {
                     for (i = 0; i < m; i++) {
                               u[i][0] = sin(PI*i/(m-1));
           /* at y = 0; all x */
                     }
           }
```

Jacobi and SOR Iterative Scheme Utility Functions

```
        if (k == p-1) {
                for (i = 0; i < m; i++) {
                        u[i][n-1] = sin(PI*i/(m-1))*exp(-PI); /* at y = 1; all x */
                }
        }
    } else if (p == 1) {
        for (i = 0; i < m; i++) {
                u[i][       0] = sin(PI*i/(m-1));          /* at y = 0; all x */
                u[i][n-1] = u[i][0]*exp(-PI); /* at y = 1; all x */
        }
    } else {
        printf("p is invalid\n");
    }
}
```

Jacobi and SOR Iterative Scheme Utility Functions

```
void prtarray( INT m, INT n, REAL **a, FILE *fd) {
/************ Print Array *************************
 * Prints array "a" with m rows and n columns   *
 * tda is the Trailing Dimension of Array a          *
 ***************************************************/
    INT i, j;
    for (i = 0; i < m; i++) {
     for (j = 0; j < n; j++) {
          fprintf(fd, "%8.2f", a[i][j]);
     }
     fprintf(fd, "\n");
    }
}
```

Jacobi and SOR Iterative Scheme Utility Functions

```
INT update_jacobi( INT m, INT n, REAL **u, REAL **unew, REAL *del) {
/***********************************************************************
 * Updates u according to Jacobi method                                      *
 * m            - (INPUT)     size of interior rows            *
 * n            - (INPUT)     size of interior columns         *
 * u            - (INPUT)     solution array                                 *
 * unew         - (INPUT)     next solution array              *
 * del          - (OUTPUT) error norm between 2 solution steps               *
 ***********************************************************************/
        INT i, j;
        *del = 0.0;
        for (i = 1; i < m-1; i++) {
                for (j = 1; j < n-1; j++) {
                                unew[i][j] = ( u[i              ][j+1] + u[i+1][j] +
                                                u[i-1][j] + u[i][j-1] )*0.25;
                                *del += fabs(unew[i][j] - u[i][j]);            /* find local max error */
                }
        }
        for (i = 1; i < m-1; i++) {
                for (j = 1; j < n-1; j++) {
                                u[i][j] = unew[i][j];
                }
        }
        return (0);
}
```

Jacobi and SOR Iterative Scheme Utility Functions

```
INT update_sor( INT m, INT n, REAL **u, REAL omega, REAL *del, CHAR redblack) {
/*********************************************************************
 * Updates u according to successive over relaxation method           *
 * m            - (INPUT)      size of interior rows              *
 * n            - (INPUT)      size of interior columns           *
 * u            - (INPUT)      array                              *
 * omega        - (INPUT)      adjustable constant used to speed up convergence of SOR        *
 * del          - (OUTPUT) error norm between 2 solution steps                                 *
 * redblack - (INPUT)          either 'r' for red and 'b' for black                            *
 *********************************************************************/
      INT i, ib, ie, j, jb, je;
      REAL up;

      *del = 0.0;
      if (redblack == 'r') {
/* process RED odd points ... */
            jb = 1; je = n-2; ib = 1; ie = m-2;
            for ( j = jb; j <= je; j+=2 ) {
                        for ( i = ib; i <=ie; i+=2 ) {
                                    up = ( u[i][j+1] + u[i+1][j] +
                                                u[i-1][j] + u[i][j-1] )*0.25;
                                    u[i][j] = (1.0 - omega)*u[i][j] + omega*up;
                                    *del += fabs(up-u[i][j]);
                        }
            }
```

Jacobi and SOR Iterative Scheme Utility Functions

```
/* process RED even points ... */
        jb = 2; je = n-2; ib = 2; ie = m-2;
        for ( j = jb; j <= je; j+=2 ) {
                for ( i = ib; i <= ie; i+=2 ) {
                        up = ( u[i][j+1] + u[i+1][j] +
                                u[i-1][j] + u[i][j-1] )*0.25;
                        u[i][j] = (1.0 - omega)*u[i][j] + omega*up;
                        *del += fabs(up-u[i][j]);
                }
        }
        return (0);
   } else {
        if (redblack == 'b') {
/* process BLACK odd points ... */
                jb = 2; je = n-2; ib = 1; ie = m-2;
                for ( j = jb; j <= je; j+=2 ) {
                        for ( i = ib; i <= ie; i+=2 ) {
                                up = ( u[i][j+1] + u[i+1][j] +
                                        u[i-1][j] + u[i][j-1] )*0.25;
                                u[i][j] = (1.0 - omega)*u[i][j] + omega*up;
                                *del += fabs(up-u[i][j]);
                        }
                }
```

Jacobi and SOR Iterative Scheme Utility Functions

```
/* process BLACK even points ... */
                jb = 1; je = n-2; ib = 2; ie = m-2;
                for ( j = jb; j <= je; j+=2 ) {
                        for ( i = ib; i <= ie; i+=2 ) {
                                up = ( u[i][j+1] + u[i+1][j] +
                                        u[i-1][j] + u[i][j-1] )*0.25;
                                u[i][j] = (1.0 - omega)*u[i][j] + omega*up;
                                *del += fabs(up-u[i][j]);
                        }
                }
                return (0);
        } else {
                return (1);
        }
   }
}
```

Jacobi and SOR Iterative Scheme Utility Functions

```
INT replicate( INT m, INT n, REAL **a, REAL **b ) {
/****************************************************
 * Replicates array a into array b          *
 * m - (INPUT)        size of interior points in 1st index *
 * n - (INPUT)        size of interior points in 2st index *
 * a - (INPUT)        solution at time N                *
 * b - (OUTPUT) solution at time N + 1                *
 ****************************************************/
    INT i, j;

    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
                b[i][j] = a[i][j];
        }
    }
    return (0);
}
```

Jacobi and SOR Iterative Scheme Utility Functions

```
INT transpose( INT m, INT n, REAL **a, REAL **at ) {
/*********************************************************
 * Transpose a(0:m+1,0:n+1) into at(0:n+1,0:m+1)                              *
 * m        - (INPUT)            size of interior points in 1st index         *
 * n        - (INPUT)            size of interior points in 2st index         *
 * a        - (INPUT)            a = a(0:m+1,0:n+1)                           *
 * at       - (OUTPUT) at = at(0:n+1,0:m+1)                                   *
 *********************************************************/
    INT i, j;

    for (i = 0; i < m; i++) {
          for (j = 0; j < n; j++) {
                    at[j][i] = a[i][j];
          }
    }
    return (0);
}
```

Jacobi and SOR Iterative Scheme Utility Functions

```
void neighbors(INT k, INT p, INT UNDEFINED, INT *below, INT *above) {
/*************************************************************
 * determines two adjacent threads                          *
 * k             - (INPUT)  current thread                  *
 * p             - (INPUT)  number of processes (threads)              *
 * UNDEFINED     - (INPUT)  code to assign to out-of-bound neighbor    *
 * below         - (OUTPUT) neighbor thread below k (usually k-1)      *
 * above         - (OUTPUT) neighbor thread above k (usually k+1)      *
 *************************************************************/
    if(k == 0) {
            *below = UNDEFINED;                 /* tells MPI not to perform send/recv */
            *above = k+1;
    } else if(k == p-1) {
            *below = k-1;
            *above = UNDEFINED;                 /* tells MPI not to perform send/recv */
    } else {
            *below = k-1;
            *above = k+1;
    }
}
```

Parallel Algorithm for the Jacobi Scheme

- First, to enable parallelism, the work must be divided among the individual processes; this is known commonly as domain decomposition.
- Because the governing equation is two-dimensional, typically the choice is to use a 1D or 2D decomposition.
- This section will focus on a 1D decomposition, deferring the discussion of a 2D decomposition for later.
- Assuming that ***p*** processes will be used, the computational domain is split into ***p*** horizontal strips, each assigned to one process, along the north-south or y-direction. This choice is made primarily to facilitate simpler boundary condition (code) implementations.

Parallel Algorithm for the Jacobi Scheme

- For the obvious reason of better load-balancing, we will divide the amount of work, in this case proportional to the grid size, evenly among the processes (***m x m / p***). For convenience, ***m' = m/p*** is defined as the number of cells in the y-direction for each process. Next, Equation 3 is restated for a process ***k*** as follows:

  Equation 6:

$$u_{i,j}^{n+1,k} \cong \frac{u_{i+1,j}^{n,k} + u_{i-1,j}^{n,k} + u_{i,j+1}^{n,k} + u_{i,j-1}^{n,k}}{4};$$

$$i = 1,\ldots,m;\ j = 1,\ldots,m';k = 0,\ldots,p-1$$

  where ***v*** denotes the local solution corresponding to the process ***k*** with ***m'=m/p***.

Parallel Algorithm for the Jacobi Scheme

- The figure below depicts the grid of a typical process ***k***, as well as part of the adjoining grids of ***k-1***, ***k+1***.

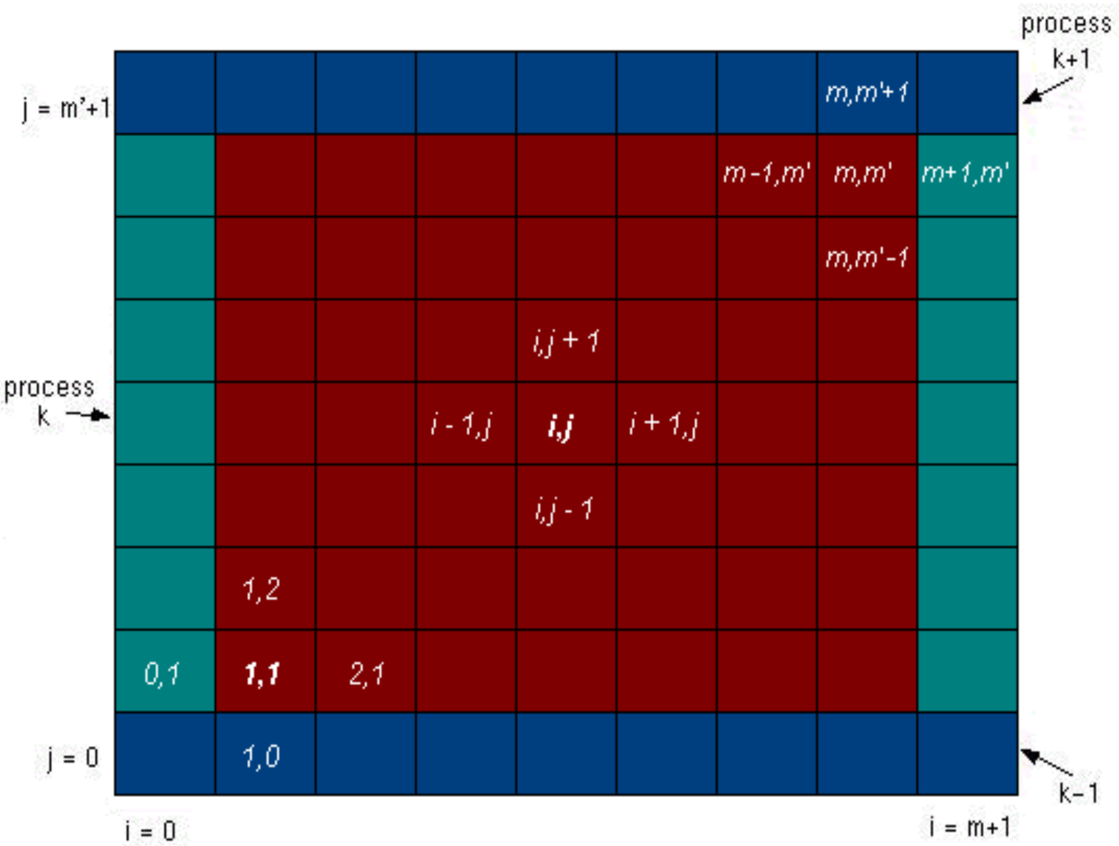


Figure 8.10. The grid of a typical process ***k*** as well as part of adjoining grids of ***k-1, k+1***

Parallel Algorithm for the Jacobi Scheme

- The **red** cells represent process ***k***'s grid cells for which the solution ***u*** is sought through Equation 6.
- The **blue** cells on the bottom row represent cells belonging to the first row (***j* = *0***) of cells of process ***k-1***.
- The **blue** cells on the top row represent the last row (***j* = *m'***) of cells of process ***k+1***.
- It is important to note that the ***u*** at the blue cells of ***k*** belong to adjoining processes (***k-1*** and ***k+1***) and hence must be "imported" via MPI message passing routines. Similarly, process ***k***'s first and last rows of cells must be "exported" to adjoining processes for the same reason.

Parallel Algorithm for the Jacobi Scheme

- For ***i* = 1** and ***i* = *m***, Equation 6 again requires an extra cell beyond these two locations. These cells contain the prescribed boundary conditions (***u(0,y) = u(1,y) = 0***) and are colored **green** to distinguish them from the **red** and **blue** cells.
- Note that no message passing operations are needed for these green cells as they are fixed boundary conditions and are known a priori.
- From the standpoint of process ***k***, the blue and green cells may be considered as additional "boundary" cells around it. As a result, the range of the strip becomes (***0:m+1,0:m'+1***).
- Physical boundary conditions are imposed on its green cells, while ***u*** is imported to its blue "boundary" cells from two adjoining processes. With the boundary conditions in place, Equation 6 can be applied to all of its interior points.
- Concurrently, all other processes proceed following the same procedure. It is interesting to note that the grid layout for a typical process ***k*** is completely analogous to that of the original undivided grid. Whereas the original problem has fixed boundary conditions, the problem for a process ***k*** is subjected to variable boundary conditions.

Parallel Algorithm for the Jacobi Scheme

- These boundary conditions can be stated mathematically as

  Equation 7:

$$v_{i,0}^{n,k} = u(x_i,0) = \sin(\pi x_i); \qquad \mathrm{i} = 0,\ldots,m+1; k = 0$$

$$v_{i,m'+1}^{n,k} = v_{i,1}^{n,k+1}; \qquad \mathrm{i} = 0,\ldots,m+1; k = 0$$

$$v_{i,0}^{n,k} = v_{i,m'}^{n,k-1}; \qquad \mathrm{i} = 0,\ldots,m+1; 0 < k < p-1$$

$$v_{i,m'+1}^{n,k} = v_{i,1}^{n,k+1}; \qquad \mathrm{i} = 0,\ldots,m+1; 0 < k < p-1$$

$$v_{i,0}^{n,k} = v_{i,m'}^{n,k-1}; \qquad \mathrm{i} = 0,\ldots,m+1; 0 < k < p-1$$

$$v_{i,0}^{n,k} = v_{i,m'}^{n,k-1}; \qquad \mathrm{i} = 0,\ldots,m+1; k = p-1$$

$$v_{i,m'+1}^{n,k} = u(x_i,1) = \sin(\pi x_i)e^{-x}; \qquad \mathrm{i} = 0,\ldots,m+1; k = p-1$$

$$v_{0,j}^{n,k} = u(0,y_j) = 0; \qquad \mathrm{j} = 1,\ldots,m'; 0 \le k \le p-1$$

$$v_{m+1,j}^{n,k} = u(1,y_j) = 0; \qquad \mathrm{j} = 1,\ldots,m'; 0 \le k \le p-1$$

Parallel Algorithm for the Jacobi Scheme

- Note that the interior points of ***u*** and ***v*** are related by the relationship

  Equation 8:

  $$u^{n}_{i,j+k\times m/p} = v^{n,k}_{i,j}; \qquad \mathrm{i} = 1,\ldots,m; j = 1,\ldots,m';0 < k < p-1$$

- Note that Cartesian topology is not employed in this implementation but will be used later in the parallel SOR example with the purpose of showing alternative ways to solve this type of problems.

Parallel Jacobi Iterative Scheme

- A parallel implementation of the Jacobi Scheme (based on a serial implementation) as applied to the Laplace equation is included below. Note that:
  - System size, m, is determined at run time.
  - Boundary conditions are handled by subroutine bc.
  - Subroutine neighbors provides the process number ABOVE and BELOW the current process. These numbers are needed for message passing (subroutine update_bc_2). If ABOVE or BELOW is "-1", its at process 0 or p-1. No message passing will be needed in that case.
  - Subroutine update_bc_2 updates the blue cells of current and adjoining processes simultaneously by MPI routine that pairs send and receive, MPI_Sendrecv, for subsequent iteration.
  - Subroutine update_bc_1 can be used in place of update_bc_2 as an alternative message passing method

Parallel Jacobi Iterative Scheme

- Subroutine printmesh may be used to print local solution for tiny cases (like 4x4)
- Pointer arrays c, n, e, w, and s point to the solution space, u. They are used to avoid unnecessary memory usage as well as to improve readability.
- MPI_Allreduce is used to collect global error from all participating processes to determine whether further interation is required. This is somewhat costly to do in every iteration. Can improve performance by calling this routine only once in a while. There is a small price to pay; the solution may have converged between MPI_Allreduce calls. See parallel SOR implementation on how to reduce MPI_Allreduce calls.
- This scheme is very slow to converge and is not used in practice. However, it serves to demonstrate parallel concepts.

Parallel Jacobi Iterative Scheme

- A parallel implementation of the Jacobi Scheme (based on a serial implementation) as applied to the Laplace equation is included below.

pjacobi.c

```
#include "solvers.h"
#include "mpi.h"

INT main(INT argc, CHAR *argv[]) {
/********** MAIN PROGRAM **********************************
 * Solve Laplace equation using Jacobi iteration method  *
 * Kadin Tseng, Boston University, August, 2000          *
 ********************************************************/
    INT iter, m, mi, mp, k, p, below, above;
    REAL del, gdel;
    CHAR line[80];
    REAL **v, **vt, **vnew;

    MPI_Init(&argc, &argv);                          /* starts MPI */
    MPI_Comm_rank(MPI_COMM_WORLD, &k); /* get current process id */
    MPI_Comm_size(MPI_COMM_WORLD, &p); /* get # procs from env or */
```

pjacobi.c

```
if(k == 0) {
        fprintf(OUTPUT,"Enter size of interior points, mi :\n");
        (void) fgets(line, sizeof(line), stdin);
        (void) sscanf(line, "%d", &mi);
        fprintf(OUTPUT,"mi = %d\n",mi);
}
MPI_Bcast(&mi, 1, MPI_INT, 0, MPI_COMM_WORLD);
m = mi + 2;   /* total is interior points plus 2 b.c. points */
mp = mi/p+2;

v  = allocate_2D(m, mp);  /* allocate mem for 2D array */
vt = allocate_2D(mp, m);
vnew = allocate_2D(mp, m);

gdel = 1.0;
iter = 0;
```

pjacobi.c

```
    bc(m, mp, v, k, p); /* initialize and define B.C. for v */
    transpose(m, mp, v, vt);           /* solve for vt */
                                       /* driven by need of update_bc_2 */
    replicate(mp, m, vt, vnew);        /* vnew = vt */
    neighbors(k, p, -1, &below, &above);   /* domain borders */

    while (gdel > TOL) {  /* iterate until error below threshold */
            iter++;                         /* increment iteration counter */

            if(iter > MAXSTEPS) {
                       fprintf(OUTPUT,"Iteration terminated (exceeds %6d", MAXSTEPS);
                       fprintf(OUTPUT," )\n");
                       return (0);      /* nonconvergent solution */
            }
/* compute new solution according to the Jacobi scheme */
            update_jacobi(mp, m, vt, vnew, &del);     /* compute new vt */
            if(iter%INCREMENT == 0) {
                       MPI_Allreduce( &del, &gdel, 1, MPI_DOUBLE,
                                   MPI_MAX, MPI_COMM_WORLD );       /* find global max error */
                       if( k == 0) {
                                   fprintf(OUTPUT,"iter,del,gdel: %6d, %lf %lf\n",iter,del,gdel);
                       }
            }
            update_bc_2( mp, m, vt, k, below, above); /* update b.c. */
    }
```

pjacobi.c

```
    if (k == 0) {
        fprintf(OUTPUT,"Stopped at iteration %d\n",iter);
        fprintf(OUTPUT,"The maximum error = %f\n",gdel);
    }

/* write v to file for use in MATLAB plots */
    transpose(mp, m, vt, v);
    write_file( m, mp, v, k, p );

    MPI_Barrier(MPI_COMM_WORLD);

    free(v); free(vt); free(vnew); /* release allocated arrays  */

    return (0);
}
```

Parallel Jacobi Iterative Scheme

- In addition, there are two modules needed in connection with the above:
- Some utilities – please refer to the slides before
- MPI-related utilities

```
/* begin MODULE mpi_module */
#include "solvers.h"
#include "mpi.h"

INT update_bc_2( INT mp, INT m, REAL **vt, INT k, INT below, INT above ) {
    MPI_Status status[6];  /* SGI doesn't define MPI_STATUS_SIZE */

    MPI_Sendrecv( vt[mp-2]+1, m-2, MPI_DOUBLE, above, 0,
        vt[0]+1, m-2, MPI_DOUBLE, below, 0,
        MPI_COMM_WORLD, status );

    MPI_Sendrecv( vt[1]+1, m-2, MPI_DOUBLE, below, 1,
        vt[mp-1]+1, m-2, MPI_DOUBLE, above, 1,
        MPI_COMM_WORLD, status );
    return (0);
}

/* end MODULE mpi_module */
```

Successive Over Relaxation (SOR)

- While the Jacobi iteration scheme is very simple and easily parallelizable, its slow convergent rate renders it impractical for any "real world" applications. One way to speed up the convergent rate would be to "over predict" the new solution by linear extrapolation. This leads to the Successive Over Relaxation (SOR) scheme shown below:

1. Make initial guess for $u_{i,j}$ at all interior points *(i,j)*.
2. Define a scalar w_n $(0 < w_n < 2)$.
3. Apply Equation 3 to all interior points *(i,j)* and call it $u'_{i,j}$.
4. $u^{n+1}_{i,j} = w_n u'_{i,j} + (1 - w_n) u^n_{i,j}$.
5. Stop if the prescribed convergence threshold is reached, otherwise continue to the next step.
6. $u^n_{i,j} = u^{n+1}_{i,j}$.
7. Go to Step 2.

Successive Over Relaxation (SOR)

- Note that in the above setting ***w$_n$* = *1*** recovers the Jacobi scheme while ***w$_n$*< *1*** underrelaxes the solution. Ideally, the choice of ***w$_n$*** should provide the optimal rate of convergence and is not restricted to a fixed constant. As a matter of fact, an effective choice of ***w$_n$***, known as the Chebyshev acceleration, is defined as

$$\omega_n = \begin{cases} 0 & for\ \mathrm{n} = 0 \\ \dfrac{1}{1 - p^2/2} & for\ \mathrm{n} = 1 \\ \dfrac{1}{1 - p^2\omega_1/4} & for\ \mathrm{n} = 2 \\ \dfrac{1}{1 - p^2\omega_{q-1}/4} & for\ \mathrm{n} = \mathrm{q} > 2 \end{cases}$$

$$\text{where } \rho = 1 - \left(\frac{\pi}{2(m+1)}\right)^2 \text{ is the spectral radius}$$

Successive Over Relaxation (SOR)

- We can further speed up the rate of convergence by using ***u*** at time level ***n +1*** for any or all terms on the right hand side of Equation 6 as soon as they become available. This is the essence of the Gauss-Seidel scheme. A conceptually similar red-black scheme will be used here. This scheme is best understood visually by painting the interior cells alternately in red and black to yield a checkerboard-like pattern as shown in Figure 8.11.

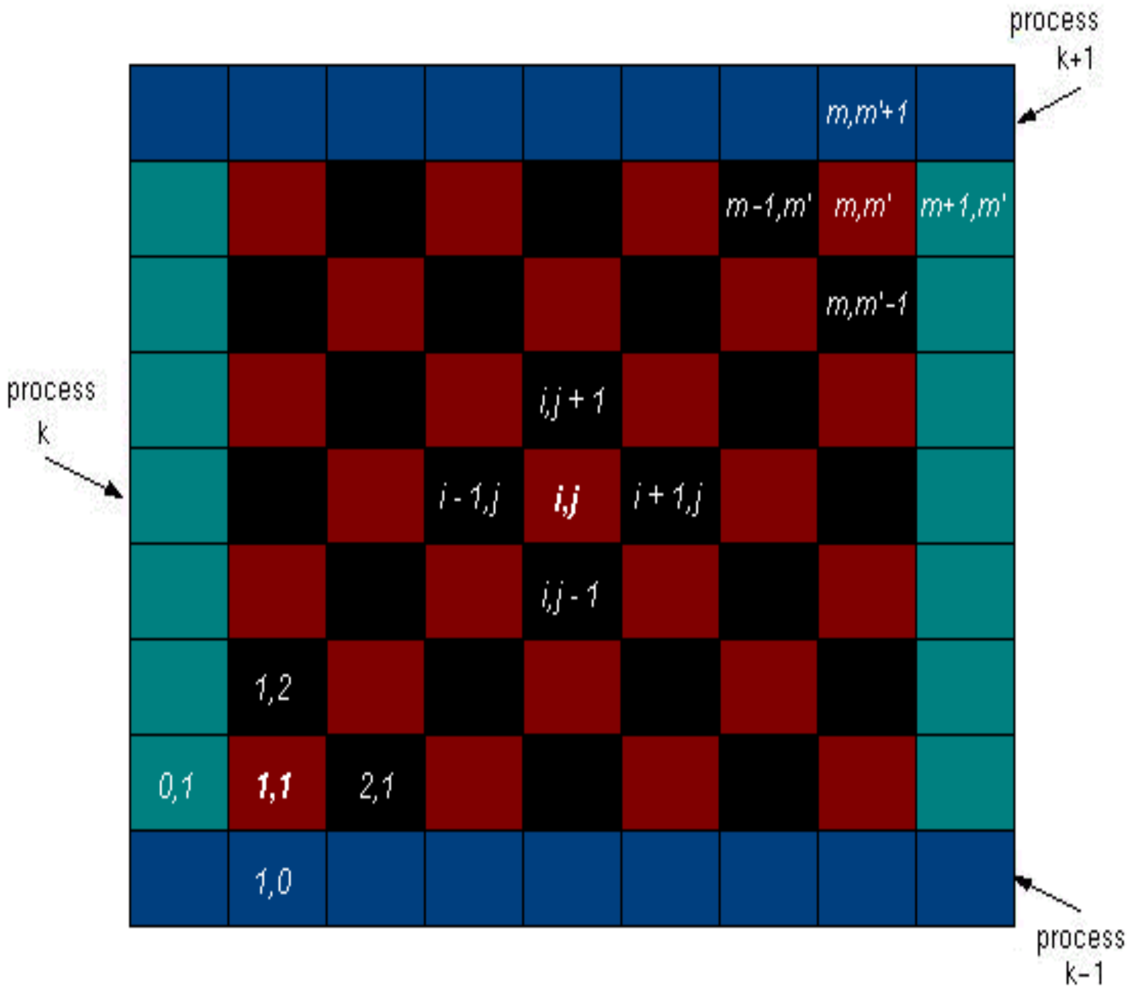


Figure 8.11. Checkerboard-like pattern depicting a parallel SOR red-black scheme.

Successive Over Relaxation (SOR)

- By using this ***red*-*black*** group identification strategy and applying the five-point finite-difference stencil to a point ***(i,j)*** located at a red cell, it is immediately apparent that the solution at the red cell depends only on its four immediate black neighbors to the north, east, west, and south (by virtue of Equation 6). On the contrary, a point ***(i,j)*** located at a black cell depends only on its north, east, west, and south red neighbors.
- In other words, the finite-difference stencil in Equation 6 effects an uncoupling of the solution at interior cells such that the solution at the red cells depends only on the solution at the black cells and vice versa.
- In a typical iteration, if we first perform an update on all red ***(i,j)*** cells, then when we perform the remaining update on black ***(i,j)*** cells, the red cells that have just been updated could be used. Otherwise, everything that we described about the Jacobi scheme applies equally well here; i.e., the ***green*** cells represent the physical boundary conditions while the solutions from the first and last rows of the grid of each process are deposited into the ***blue*** cells of respective process grids to be used as the remaining boundary conditions.

Serial SOR Iterative Scheme

- A single-process implementation of the SOR Scheme as applied to the Laplace equation is given below. Note that
  - Program is written in C.
  - System size, m, is determined at run time.
  - Boundary conditions are handled by subroutine bc.
  - This scheme converges much more rapidly than the Jacobi Scheme, especially when coupled with a Checbyshev acceleration.

ssor.c

```
#include "solvers.h"
INT main() {
/***************MAIN PROGRAM **************************************
 * Solve Laplace equation using Successive Over Relaxation        *
 * and Chebyshev Acceleration (see Numerical Recipe for detail) *
 * Kadin Tseng, Boston University, August, 2000                   *
 *******************************************************************/
    INT m, mi, mp, iter;  CHAR line[10];
    REAL omega, rhoj, rhojsq, delr, delb, gdel;
    REAL **u;

    fprintf(OUTPUT,"Enter size of interior points, mi :");
    (void) fgets(line, sizeof(line), stdin);
    (void) sscanf(line, "%d", &mi);
    fprintf(OUTPUT,"mi = %d\n",mi);

    m = mi + 2;
    gdel = 1.0; iter = 0; mp = m/P;
    rhoj = 1.0 - PI*PI*0.5/m/m;
    rhojsq = rhoj*rhoj;
```

ssor.c

```
u = allocate_2D(m, mp);   /* allocate space for 2D array u */

bc( m, mp, u, K, P);   /* initialize and define B.C. for u */

omega = 1.0;
update_sor( m, mp, u, omega, &delr, 'r');
omega = 1.0/(1.0 - 0.50*rhojsq);
update_sor( m, mp, u, omega, &delb, 'b');

while (gdel > TOL) {     /* iterate until error below threshold */
        iter++;                          /* increment iteration counter */
        omega = 1.0/(1.0 - 0.25*rhojsq*omega);
        update_sor( m, mp, u, omega, &delr, 'r');
        omega = 1.0/(1.0 - 0.25*rhojsq*omega);
        update_sor( m, mp, u, omega, &delb, 'b');
        gdel = (delr + delb)*4.0;
```

ssor.c

```
        if(iter%INCREMENT == 0) {
                fprintf(OUTPUT,"iter gdel omega: %5d %13.5f %13.5f\n",iter,gdel,omega);
        }
        if(iter > MAXSTEPS) {
                fprintf(OUTPUT,"Iteration terminated (exceeds %6d", MAXSTEPS);
                fprintf(OUTPUT," )\n");
                return (0);                 /* nonconvergent solution */
        }
    }

    fprintf(OUTPUT,"Stopped at iteration %d\n",iter);
    fprintf(OUTPUT,"The maximum error = %f\n",gdel);

/* write u to file for use in MATLAB plots */
    write_file( m, mp, u, K, P);

    return (0);
}
```

Parallel SOR Red-black Scheme

- The parallel aspect of the Jacobi scheme can be usec verbatim for the SOR scheme. Figure 8.11, as introduced in the previous section on the single-thread SOR scheme, may be used to represent the layout for a typical thread "k" of the SOR scheme.
- As before, the green boxes denote boundary cells that are prescribed while the blue boxes represent boundary cells whose values are updated at each iteration by way of message passing.

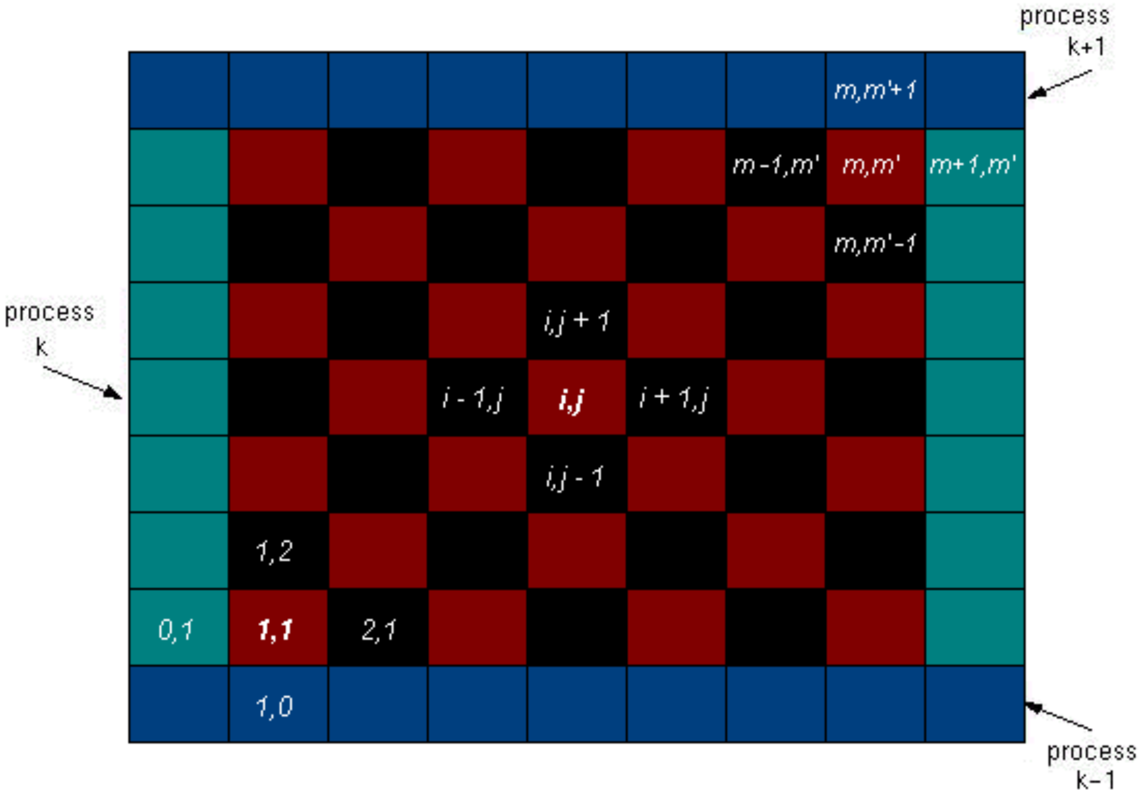


Figure 8.11. Checkerboard-like pattern depicting a parallel SOR red-black scheme.

Parallel SOR Red-black Scheme

- A multi-threaded implementation of the SOR Scheme as applied to the Laplace equation is given below. Note that
  - Program is written in C.
  - System size, m, is determined at run time.
  - Boundary conditions are handled by subroutine bc.
  - This scheme converges much more rapidly than the Jacobi Scheme, especially when coupled with a Checbyshev acceleration.

psor.c

```
#include "solvers.h"
#include "mpi.h"

INT main(INT argc, CHAR *argv[]) {
/**************MAIN PROGRAM ************************************
 * Solve Laplace equation using Successive Over Relaxation        *
 * and Chebyshev Acceleration (see Numerical Recipe for detail)  *
 * Kadin Tseng, Boston University, August, 2000                                *
 *************************************************************/
    INT iter, m, mi, mp, p, k, below, above;
    REAL omega, rhoj, rhojsq, del, delr, delb, gdel;
    CHAR line[80], red, black;
    MPI_Comm grid_comm;
    INT me, iv, coord[1], dims, periods, ndim, reorder;
    REAL **v, **vt;

    MPI_Init(&argc, &argv);                            /* starts MPI */
    MPI_Comm_rank(MPI_COMM_WORLD, &k);  /* get current process id */
    MPI_Comm_size(MPI_COMM_WORLD, &p);  /* get # procs from env or */
```

psor.c

```
periods = 0; ndim = 1; reorder = 0; red = 'r'; black = 'b';

if(k == 0) {
        fprintf(OUTPUT,"Enter size of interior points, mi :\n");
        (void) fgets(line, sizeof(line), stdin);
        (void) sscanf(line, "%d", &mi);
        fprintf(OUTPUT,"mi = %d\n",mi);
        m = mi + 2;     /* total is mi plus 2 b.c. points */
}
MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
mp = (m-2)/p+2;

v  = allocate_2D(m, mp);  /* allocate mem for 2D array */
vt = allocate_2D(mp, m);

gdel = 1.0;
iter = 0;
rhoj = 1.0 - PI*PI*0.5/m/m;
rhojsq = rhoj*rhoj;
```

psor.c

```
/* create cartesian topology for matrix */
   dims = p;
   MPI_Cart_create(MPI_COMM_WORLD, ndim, &dims,
                                  &periods, reorder, &grid_comm);
   MPI_Comm_rank(grid_comm, &me);
   MPI_Cart_coords(grid_comm, me, ndim, coord);
   iv = coord[0];
   bc( m, mp, v, iv, p);      /* set up boundary conditions */
   transpose(m, mp, v, vt);  /* transpose v into vt */

   replicate(mp, m, vt, v);
   MPI_Cart_shift(grid_comm, 0, 1, &below, &above);

   omega = 1.0;
   update_sor( mp, m, vt, omega, &delr, red);
   update_bc_2( mp, m, vt, iv, below, above);
   omega = 1.0/(1.0 - 0.50*rhojsq);
   update_sor( mp, m, vt, omega, &delb, black);
   update_bc_2( mp, m, vt, iv, below, above);
```

psor.c

```
while (gdel > TOL) {
        iter++;   /* increment iteration counter */
        omega = 1.0/(1.0 - 0.25*rhojsq*omega);
        update_sor( mp, m, vt, omega, &delr, red);
        update_bc_2( mp, m, vt, iv, below, above);
        omega = 1.0/(1.0 - 0.25*rhojsq*omega);
        update_sor( mp, m, vt, omega, &delb, black);
        update_bc_2( mp, m, vt, iv, below, above);
        if(iter%INCREMENT == 0) {
                del = (delr + delb)*4.0;
                MPI_Allreduce( &del, &gdel, 1, MPI_DOUBLE,
                MPI_MAX, MPI_COMM_WORLD);        /*  find global max error */
                if (k == 0) {
                        fprintf(OUTPUT,"iter gdel omega: %5d %13.5f %13.5f\n",iter,gdel,omega);
                }
        }
        if(iter > MAXSTEPS) {
                fprintf(OUTPUT,"Iteration terminated (exceeds %6d", MAXSTEPS);
                fprintf(OUTPUT," )\n");
                return (1);                              /* nonconvergent solution */
        }
}
```

psor.c

```
    if (k == 0) {
        fprintf(OUTPUT,"Stopped at iteration %d\n",iter);
        fprintf(OUTPUT,"The maximum error = %f\n",gdel);
    }

/* write v to file for use in MATLAB plots */
    transpose(mp, m, vt, v);  /* transpose v into vt */
    write_file( m, mp, v, k, p);

    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Finalize();

    return (0);
}
```

Scalability Plot of SOR

- The plot in Figure 8.12 below shows the scalability of the MPI implementation of the Laplace equation using SOR on an SGI Origin 2000 shared-memory multiprocessor.

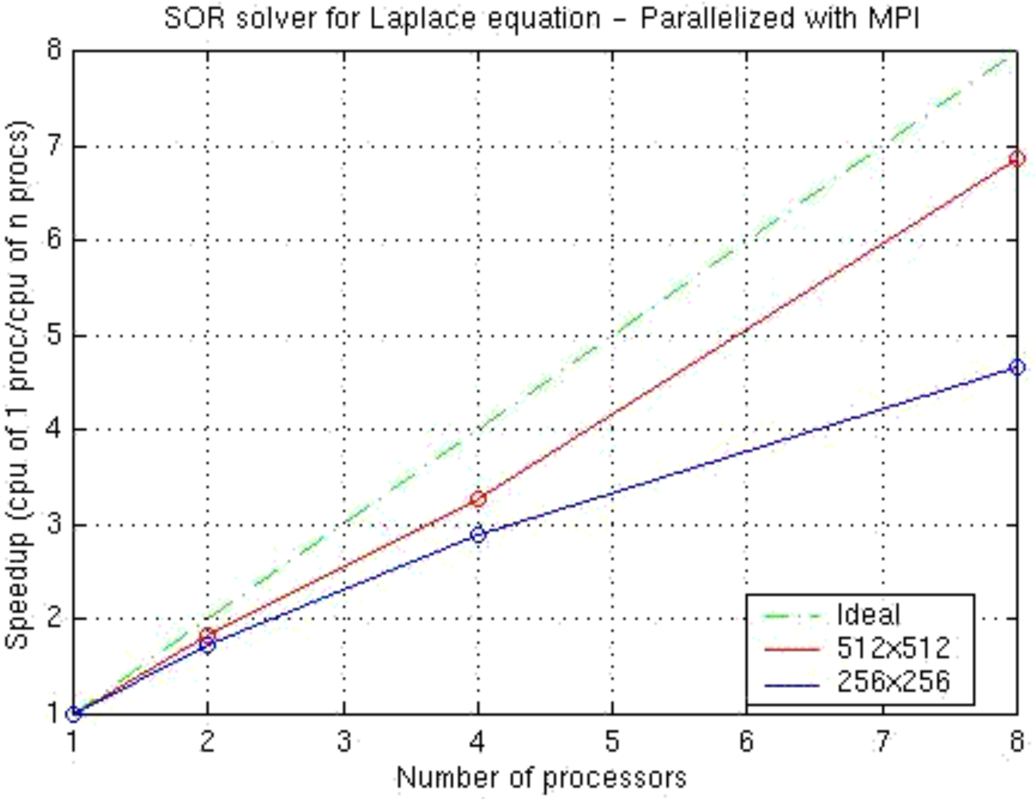


Figure 8.12. Scalability plot using SOR on an SGI Origin 2000.

END