Virtual Topologies: Iterative Solvers

Slides taken from full lecture slides at:

http://site.sci.hkbu.edu.hk/tdgc/tutorial.php

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)

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

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

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

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 $\Delta x = \Delta y = \frac{1}{m+1}$
- 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

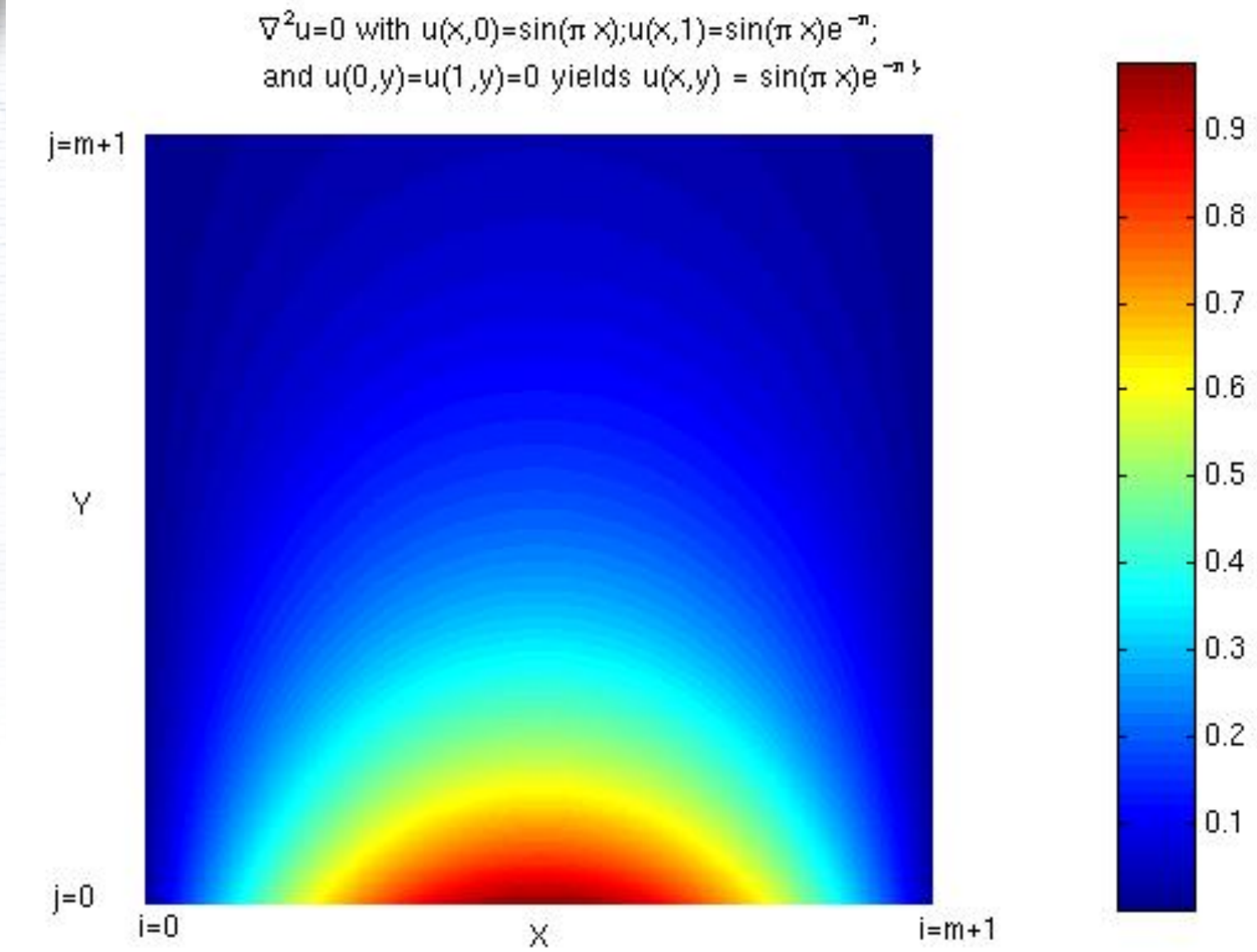

Figure 8.9. Contour plot showing the analytical solution for the boundary value problem.

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***.

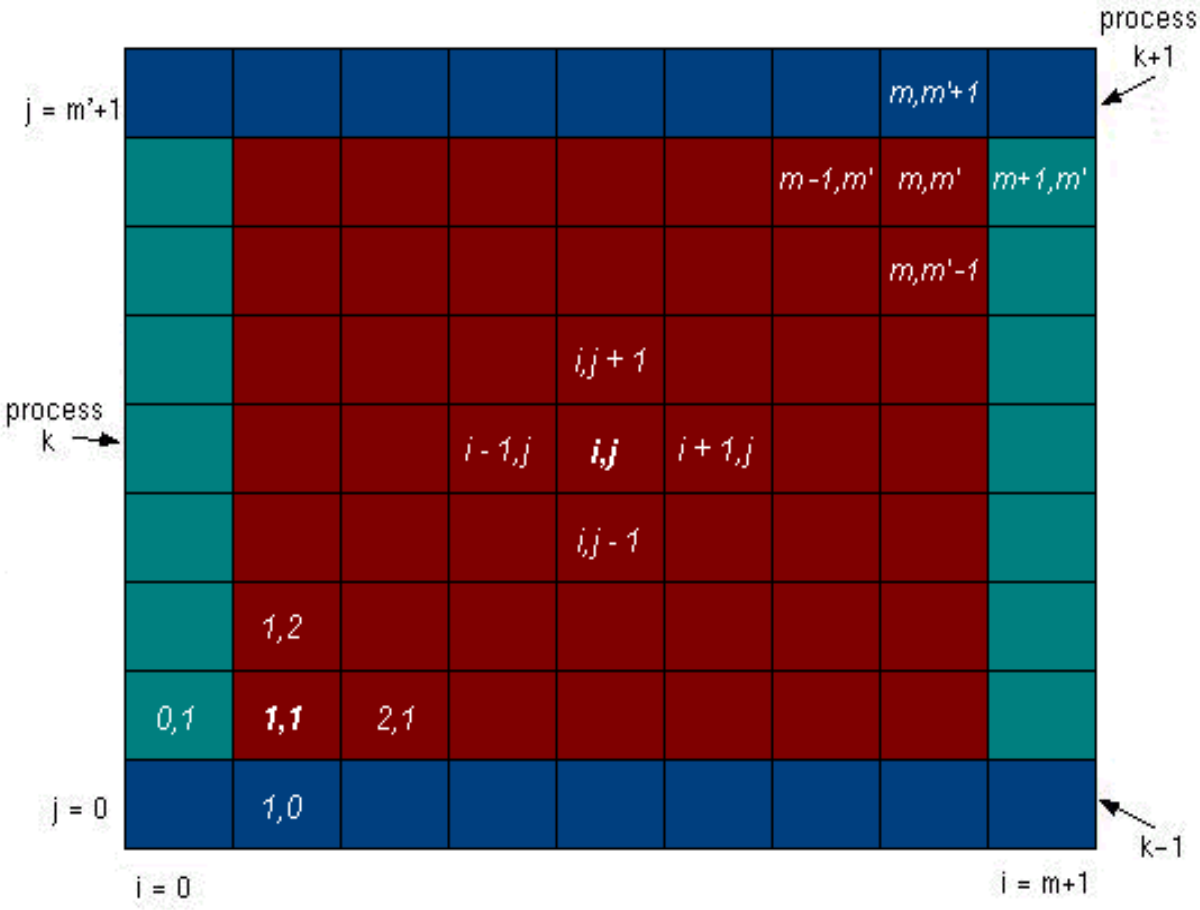

Figure 8.10. The grid of a typical process ***k*** as well as part of adjoining grids of ***k-1***, ***k+1***

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 $\mathbf{w_n}$ ***(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

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

where $\rho = 1 - \left(\dfrac{\pi}{2(m+1)}\right)^2$ 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.

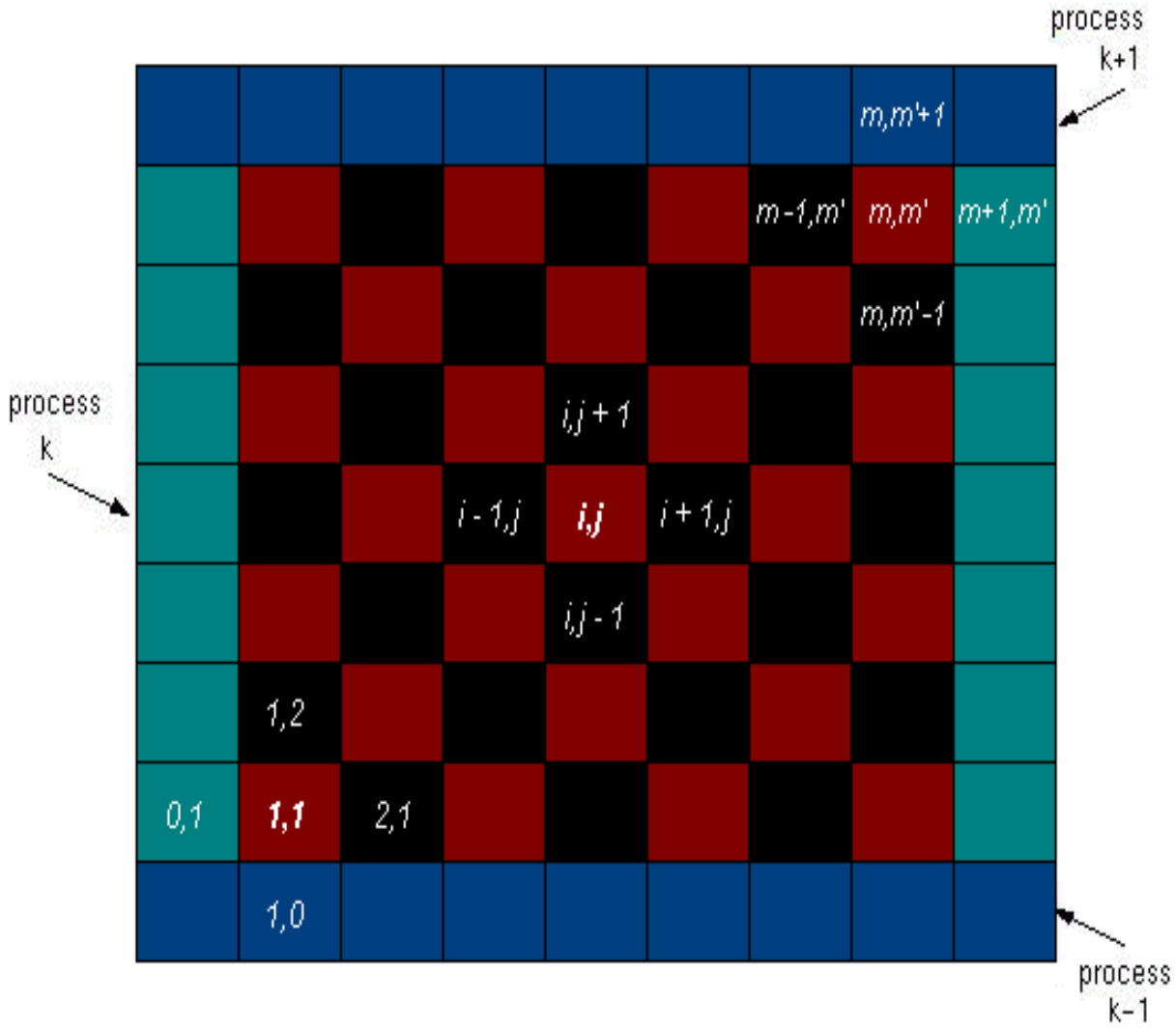

Figure 8.11. Checkerboard-like pattern depicting a parallel SOR red-black scheme.

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.

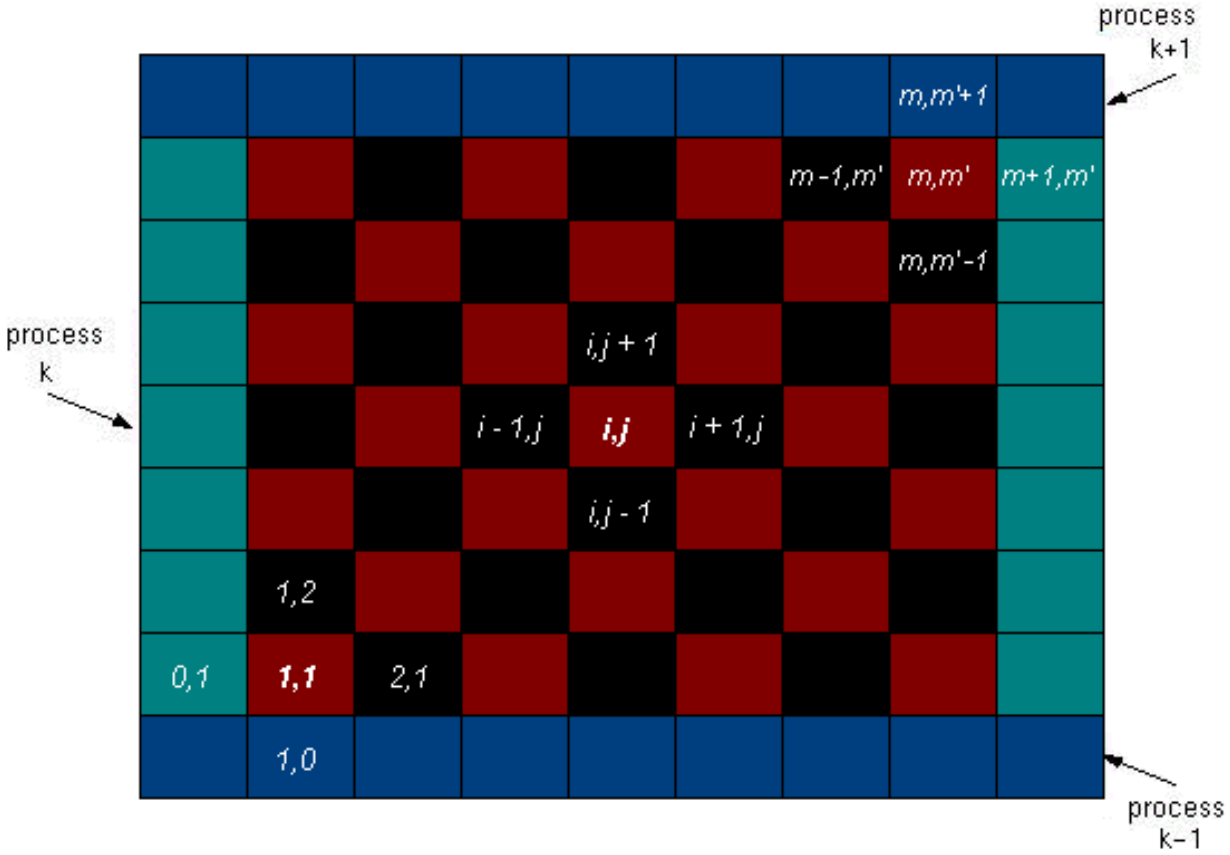

Figure 8.11. Checkerboard-like pattern depicting a parallel SOR red-black scheme.

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.

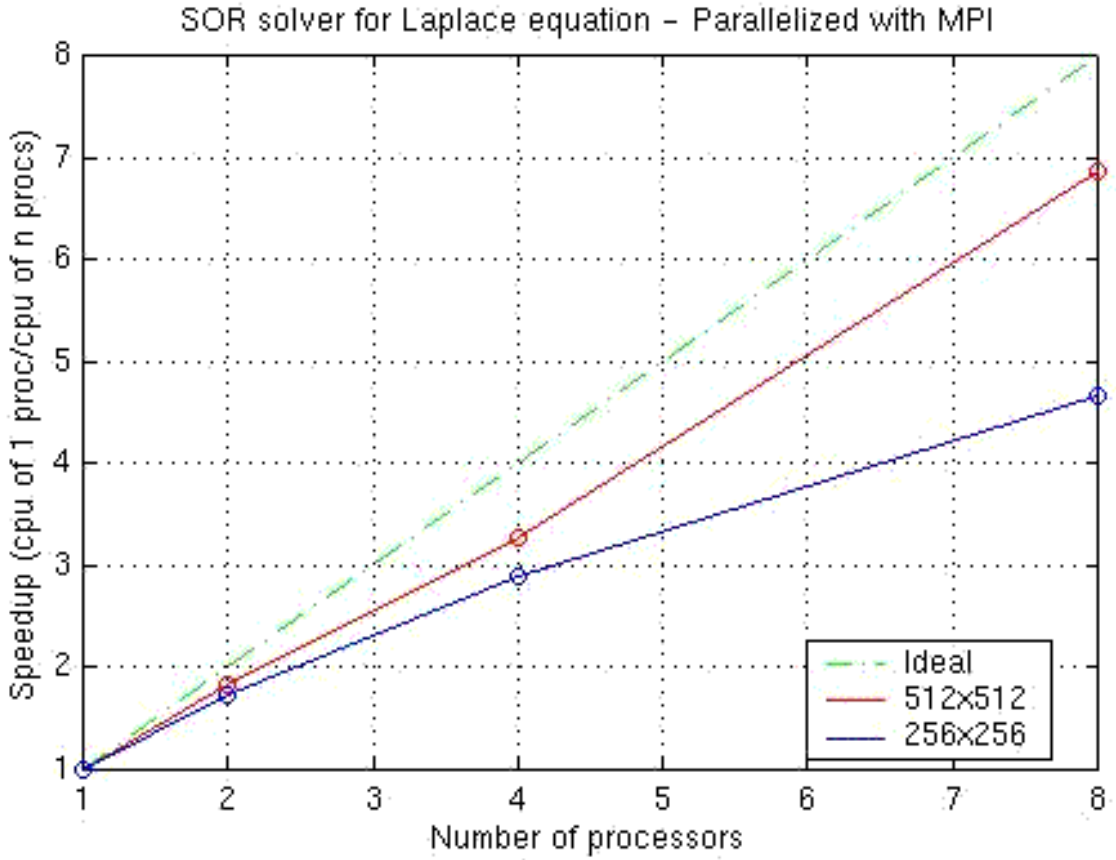

Figure 8.12. Scalability plot using SOR on an SGI Origin 2000.

END