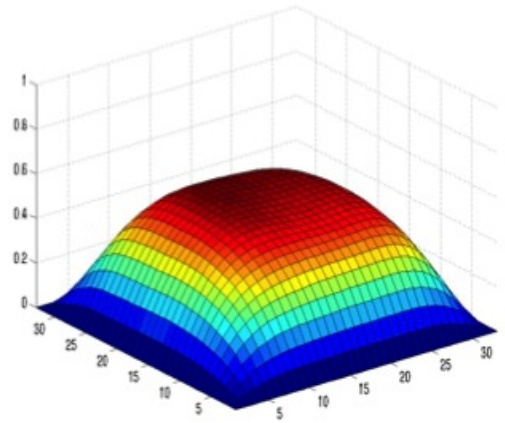
COMP 705: Advanced Parallel Computing
Topic Notes: 2D Jacobian Iterative Solver, in C

Mary Thomas

Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

Topic Update: 09/10/17



Partial Differential Equations

- For solving systems of equations, including nonlinear systems of the form:

$$A\mathbf{x} = \mathbf{b}.$$

- Heat/diffusion equation : Heat transfer, particle diffusion, approximation of nuclear transport
- Poisson/Laplace equation : Electromagnetics
- Wave equation : wave propagation, vibration
- Fluid dynamics

PDE Solver methods: Direct

- Mathematical procedure that generates a sequence of improving approximate solutions
- Gaussian elimination: sequence of elementary row operations modify matrix until lower left-hand corner of the matrix is filled with zeros
- LU decomposition: product of a lower triangular matrix and an upper triangular matrix; can be viewed as the matrix form of Gaussian elimination.
- Can require a lot of memory/computation for required resolution

PDE Solver methods: Iterative/Relaxation Methods

- Approach the solution gradually – converge to a residual
- Jacobi - diagonally dominant system of linear equations.
  - Solves for each diagonal element → approximate value.
  - The process is iterated until it converges
- Gauss-Seidel: method of successive displacement
  - Decompose A into lower and upper triangles.
  - convergence only guaranteed if matrix is diagonally dominant, or symmetric+positive definite
- Successive over-relaxation (SOR)
  - Solves for x^{k+1} using forward substituion.
  - calculates $x_i^{k+1} = F\{x_{i-1}^{k+1}, x_{i+1}^{k}\}$
- Multigrid
  - using a recursive number of discretizations: coarse → fine grids
  - stops when error reaches tolerance and cost is minimized.
  - typical application: elliptic partial differential equations
- Krylov subspace methods
  - Generalized minimum residual (GMRES)
  - Conjugate gradient

2D Laplacian - Heat Equation

2D Laplacian:

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

Boundary Conditions:

u(x,0) = $sin(\pi x)$ $0 <= x <= 1$

u(x,1) = $sin(\pi x)\, e^{-x}$ $0 \leq x \leq 1$

u(1,y) = 0 $0 <= y <= 1$

Analytical solution: $sin(\pi x)\, e^{-xy}$ $(0 \leq x \leq 1); (0 \leq y \leq 1)$.

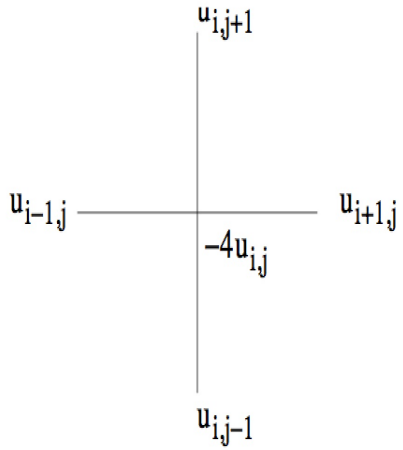
Jacobi Iterative Scheme - Gropp MPI Examples

Jacobi Iteration - Finite Difference Approximation

Use Taylor Series expansion on uniform grid to yield linear system of equations

$$\nabla^2 u_{i,j} = \frac{1}{h^2}[u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}] = 0$$

```
do it=1, 100
call exchng2( b, sx, ex, sy, ey, comm2d, stride, &
                      nbrleft, nbrright, nbrtop, nbrbottom )
call sweep2d( b, f, nx, sx, ex, sy, ey, a )
call exchng2( a, sx, ex, sy, ey, comm2d, stride, &
                      nbrleft, nbrright, nbrtop, nbrbottom )
call sweep2d( a, f, nx, sx, ex, sy, ey, b )
dwork = diff2d( a, b, nx, sx, ex, sy, ey )
call MPI_Allreduce( dwork, diffnorm, 1, MPI_DOUBLE_PRECISION, &
                           MPI_SUM, comm2d, ierr )
        if (diffnorm .lt. 1.0e-5) exit
        if (myid .eq. 0) print *, 2*it, ' Difference is ', diffnorm
enddo
```



Source: Gropp: http://www.mcs.anl.gov/research/projects/mpi/usingmpi/examples-usingmpi/intermediate/index.html

Gropp: MPI Jacobi Iterative Scheme - Main Routine

```
!**************************************************************************
!   twod.f90 - a solution to the Poisson problem by using Jacobi
!   interation on a 2-d decomposition
!
!   .... the rest of this is from pi3.f to show the style ...
!
!   Each node:
!    1) receives the number of rectangles used in the approximation.
!    2) calculates the areas of it's rectangles.
!    3) Synchronizes for a global summation.
!   Node 0 prints the result.
!
!  Variables:
!
!    pi  the calculated result
!    n   number of points of integration.
!    x           midpoint of each rectangle's interval
!    f           function to integrate
!    sum,pi      area of rectangles
!    tmp         temporary scratch space for global summation
!    i           do loop index
!
!     This code is included (without the prints) because one version of
!     MPICH SEGV'ed (probably because of errors in handling send/recv of
!     MPI_PROC_NULL source/destination).
!
!**************************************************************************
```

Source: Gropp: http://www.mcs.anl.gov/research/projects/mpi/usingmpi/examples-usingmpi/intermediate/index.html

Jacobi Iterative Scheme - Main Routine

```
      program main
      use mpi
      integer maxn
      parameter (maxn = 128)
      double precision  a(maxn,maxn), b(maxn,maxn), f(maxn,maxn)
      integer nx, ny
      integer myid, numprocs, it, rc, comm2d, ierr, stride
      integer nbrleft, nbrright, nbrtop, nbrbottom
      integer sx, ex, sy, ey
      integer dims(2)
      logical periods(2)
      double precision diff2d, diffnorm, dwork
      double precision t1, t2
      external diff2d
      data periods/2*.false./

      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
!      print *, "Process ", myid, " of ", numprocs, " is alive"
      if (myid .eq. 0) then
!
!         Get the size of the problem
!
!          print *, 'Enter nx'
!          read *, nx
           nx = 10
      endif
!      print *, 'About to do bcast on ', myid
      call MPI_BCAST(nx,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
      ny   = nx
```

```
! Get a new communicator for a decomposition of the domain.
! Let MPI find a "good" decomposition
!
      dims(1) = 0
      dims(2) = 0
      call MPI_DIMS_CREATE( numprocs, 2, dims, ierr )
      call MPI_CART_CREATE( MPI_COMM_WORLD, 2, dims, periods, .true.,
                            comm2d, ierr )
!
! Get my position in this communicator
!
      call MPI_COMM_RANK( comm2d, myid, ierr )
!      print *, "Process ", myid, " of ", numprocs, " is alive"
!
! My neighbors are now +/- 1 with my rank.  Handle the case of the
! boundaries by using MPI_PROCNULL.
      call fnd2dnbrs( comm2d, nbrleft, nbrright, nbrtop, nbrbottom )
!      print *, "Process ", myid, ":",
!     *     nbrleft, nbrright, nbrtop, nbrbottom
!
! Compute the decomposition
!
      call fnd2ddecomp( comm2d, nx, sx, ex, sy, ey )
!      print *, "Process ", myid, ":", sx, ex, sy, ey
!
! Create a new, "strided" datatype for the exchange in the
! "non-contiguous"  direction
!
      call mpi_Type_vector( ey-sy+1, 1, ex-sx+3, &
                            MPI_DOUBLE_PRECISION, stride, ierr )
      call mpi_Type_commit( stride, ierr )
```

Source: Gropp: http://www.mcs.anl.gov/research/projects/mpi/usingmpi/examples-usingmpi/intermediate/index.html

Jacobi Iterative Scheme - Main Routine

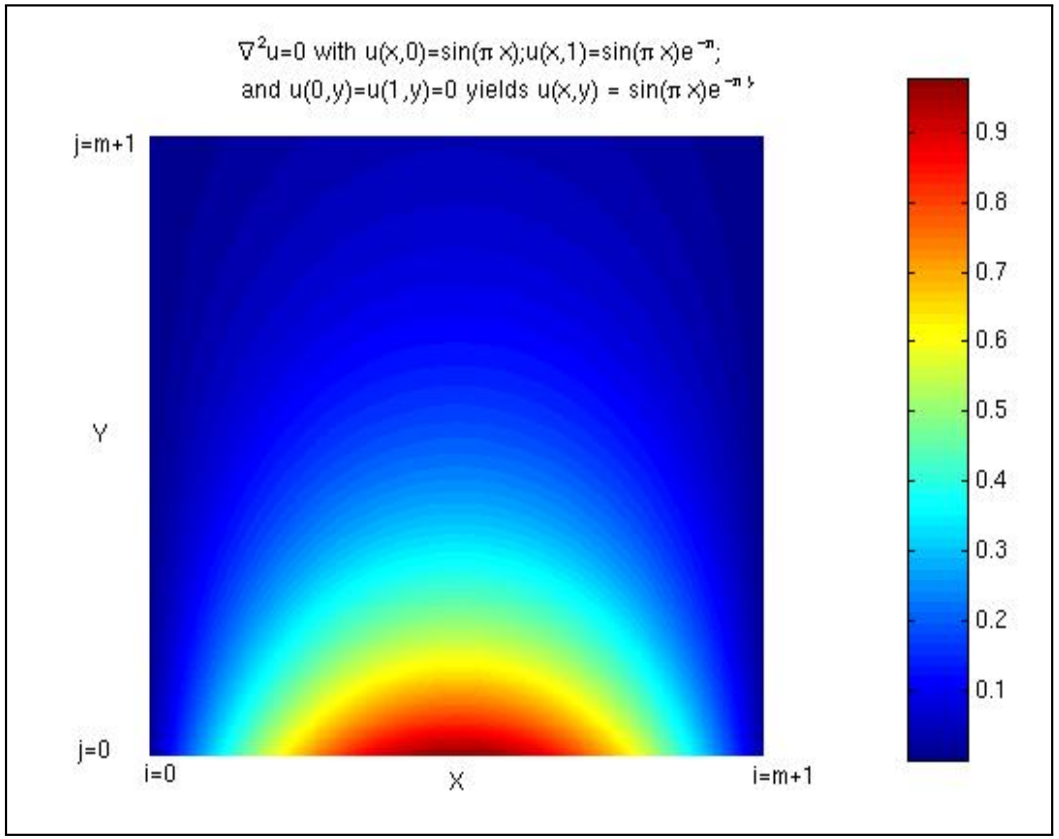
```
!
!
! Initialize the right-hand-side (f) and the initial solution guess (a)
!
      call twodinit( a, b, f, nx, sx, ex, sy, ey )
!
! Actually do the computation.  Note the use of a collective operation to
! check for convergence, and a do-loop to bound the number of iterations.
!
      call MPI_BARRIER( MPI_COMM_WORLD, ierr )
      t1 = MPI_WTIME()
      do it=1, 100
call exchng2( b, sx, ex, sy, ey, comm2d, stride, &
                     nbrleft, nbrright, nbrtop, nbrbottom )
call sweep2d( b, f, nx, sx, ex, sy, ey, a )
call exchng2( a, sx, ex, sy, ey, comm2d, stride, &
                     nbrleft, nbrright, nbrtop, nbrbottom )
call sweep2d( a, f, nx, sx, ex, sy, ey, b )
dwork = diff2d( a, b, nx, sx, ex, sy, ey )
call MPI_Allreduce( dwork, diffnorm, 1, MPI_DOUBLE_PRECISION, &
                          MPI_SUM, comm2d, ierr )
        if (diffnorm .lt. 1.0e-5) exit
        if (myid .eq. 0) print *, 2*it, ' Difference is ', diffnorm
      enddo
      t2 = MPI_WTIME()
      if (myid .eq. 0 .and. it .gt. 100) print *, 'Failed to converge'
!      if (myid .eq. 0) then
!         print *, 'Converged after ', 2*it, ' Iterations in ', t2 - t1,
!     $        ' secs '
!      endif
!
```

```
call MPI_Type_free( stride, ierr )
 call MPI_Comm_free( comm2d, ierr )
 call MPI_FINALIZE(rc)
 end
```

Source: Gropp: http://www.mcs.anl.gov/research/projects/mpi/usingmpi/examples-usingmpi/intermediate/index.html

Jacobi Iterative Scheme - Boundary Conditions

```
      subroutine twodinit( a, b, f, nx, sx, ex, sy, ey )
         integer nx, sx, ex, sy, ey
         double precision a(sx-1:ex+1, sy-1:ey+1), b(sx-1:ex+1, sy-1:ey+1), &
                          f(sx-1:ex+1, sy-1:ey+1)
         integer i, j
!
         do j=sy-1,ey+1
            do i=sx-1,ex+1
               a(i,j) = 0.0d0
               b(i,j) = 0.0d0
               f(i,j) = 0.0d0
            enddo
         enddo
!
!     Handle boundary conditions
         if (sx .eq. 1) then
            do j=sy,ey
               a(0,j) = 1.0d0
               b(0,j) = 1.0d0
            enddo
         endif
         if (ex .eq. nx) then
            do j=sy,ey
               a(nx+1,j) = 0.0d0
               b(nx+1,j) = 0.0d0
            enddo
         endif
         if (sy .eq. 1) then
            do i=sx,ex
               a(i,0) = 1.0d0
               b(i,0) = 1.0d0
            enddo
         endif
         return
         end
```



Source: Code: Gropp: http://www.mcs.anl.gov/research/projects/mpi/usingmpi/examples-usingmpi/intermediate/index.html

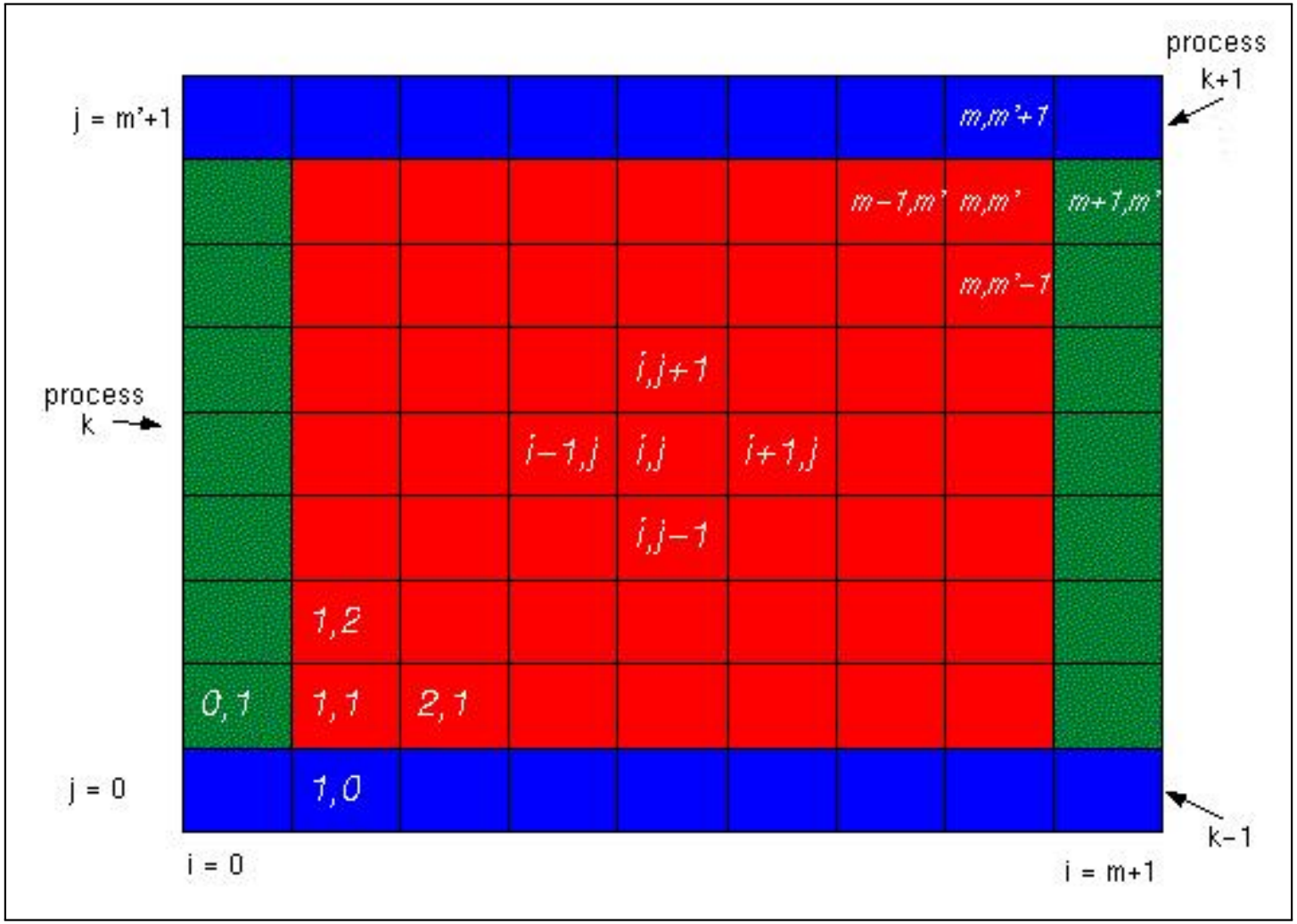
Parallel Jacobi Approach

- Divide work evenly among processors (mxm/p),
- Divide work into P (number of PEs) horizontal strips
- Rewrite FD equation for solving u on PE k:

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

- n is the iteration number
- **Red** cells hold solution at iteration $(n+1)$
- **Blue** cells on top/bottom are the neighbor cells –¿ need to get them from other processor
- Green cells hold boundary conditions

Ghost Cell Layout



Source: Kaden Notes: `http://scv.bu.edu/~kadin/alliance/apply/solvers/`

Parallel Jacobi Code: Exchange routines

```
subroutine exchng1(a, nx, s, e, comm1d, nbrbottom, nbrtop)
  use mpi
  integer nx, s, e, comm1d, nbrbottom, nbrtop
  double precision a(0:nx+1,s-1:e+1)
  integer rank, coord, ierr
!
  call MPI_COMM_RANK(comm1d, rank, ierr)
  call MPI_CART_COORDS(comm1d, rank, 1, coord, ierr)
  if (mod(coord, 2) .eq. 0) then
     call MPI_SEND(a(1,e), nx, MPI_DOUBLE_PRECISION, &
                   nbrtop, 0, comm1d, ierr)
     call MPI_RECV(a(1,s-1), nx, MPI_DOUBLE_PRECISION, &
                   nbrbottom, 0, comm1d, MPI_STATUS_IGNORE, ierr)
     call MPI_SEND(a(1,s), nx, MPI_DOUBLE_PRECISION, &
                   nbrbottom, 1, comm1d, ierr)
     call MPI_RECV(a(1,e+1), nx, MPI_DOUBLE_PRECISION, &
                   nbrtop, 1, comm1d, MPI_STATUS_IGNORE, ierr)
  else
     call MPI_RECV(a(1,s-1), nx, MPI_DOUBLE_PRECISION, &
                   nbrbottom, 0, comm1d, MPI_STATUS_IGNORE, ierr)
     call MPI_SEND(a(1,e), nx, MPI_DOUBLE_PRECISION, &
                   nbrtop, 0, comm1d, ierr)
     call MPI_RECV(a(1,e+1), nx, MPI_DOUBLE_PRECISION, &
                   nbrtop, 1, comm1d, MPI_STATUS_IGNORE, ierr)
     call MPI_SEND(a(1,s), nx, MPI_DOUBLE_PRECISION, &
                   nbrbottom, 1, comm1d, ierr)
  endif
  return
  end
```

```
 subroutine exchng2(a, sx, ex, sy, ey,  &
                    comm2d, stridetype, &
                    nbrleft, nbrright, nbrtop, nbrbottom )
  use mpi
  integer sx, ex, sy, ey, stridetype
  double precision a(sx-1:ex+1, sy-1:ey+1)
  integer nbrleft, nbrright, nbrtop, nbrbottom, comm2d
  integer ierr, nx
!
  nx = ex - sx + 1
!  These are just like the 1-d versions, except for less data
  call MPI_SENDRECV(a(sx,ey),  nx, MPI_DOUBLE_PRECISION,    &
                    nbrtop, 0,                              &
                    a(sx,sy-1), nx, MPI_DOUBLE_PRECISION,   &
                    nbrbottom, 0, comm2d, MPI_STATUS_IGNORE, ierr)
  call MPI_SENDRECV(a(sx,sy),  nx, MPI_DOUBLE_PRECISION,   &
                    nbrbottom, 1,                           &
                    a(sx,ey+1), nx, MPI_DOUBLE_PRECISION,  &
                    nbrtop, 1, comm2d, MPI_STATUS_IGNORE, ierr)
!
! This uses the vector datatype stridetype
  call MPI_SENDRECV(a(ex,sy),  1, stridetype, nbrright, 0, &
                    a(sx-1,sy), 1, stridetype, nbrleft, 0, &
                    comm2d, MPI_STATUS_IGNORE, ierr)
  call MPI_SENDRECV(a(sx,sy),  1, stridetype, nbrleft,   1,&
                    a(ex+1,sy), 1, stridetype, nbrright, 1,&
                    comm2d, MPI_STATUS_IGNORE, ierr)
  return
  end
```

Parallel Jacobi - Update Routines

```
  SUBROUTINE update_bc_1(v, m, mp, k, below, above)
    IMPLICIT NONE
    INCLUDE 'mpif.h'
    INTEGER :: m, mp, k, ierr, below, above
    REAL(real8), DIMENSION(0:m+1,0:mp+1) :: v
    INTEGER status(MPI_STATUS_SIZE)
! Select 2nd index for domain decomposition to have stride 1
! Use odd/even scheme to reduce contention in message passing
    IF(mod(k,2) == 0) THEN     ! even numbered processes
      CALL MPI_Send( v(1,mp  ), m, MPI_DOUBLE_PRECISION, above, 0,  &
                     MPI_COMM_WORLD, ierr)
      CALL MPI_Recv( v(1,0   ), m, MPI_DOUBLE_PRECISION, below, 0,  &
                     MPI_COMM_WORLD, status, ierr)
      CALL MPI_Send( v(1,1   ), m, MPI_DOUBLE_PRECISION, below, 1,  &
                     MPI_COMM_WORLD, ierr)
      CALL MPI_Recv( v(1,mp+1), m, MPI_DOUBLE_PRECISION, above, 1,  &
                     MPI_COMM_WORLD, status, ierr)
    ELSE                          ! odd numbered processes
      CALL MPI_Recv( v(1,0   ), m, MPI_DOUBLE_PRECISION, below, 0,  &
                     MPI_COMM_WORLD, status, ierr)
      CALL MPI_Send( v(1,mp  ), m, MPI_DOUBLE_PRECISION, above, 0,  &
                     MPI_COMM_WORLD, ierr)
      CALL MPI_Recv( v(1,mp+1), m, MPI_DOUBLE_PRECISION, above, 1,  &
                     MPI_COMM_WORLD, status, ierr)
      CALL MPI_Send( v(1,1   ), m, MPI_DOUBLE_PRECISION, below, 1,  &
                     MPI_COMM_WORLD, ierr)
    ENDIF
    RETURN
  END SUBROUTINE update_bc_1
```

Parallel Jacobi - Update Routines

```
SUBROUTINE update_bc_2( v, m, mp, k, below, above )
  INCLUDE "mpif.h"
  INTEGER :: m, mp, k, below, above, ierr
  REAL(real8), dimension(0:m+1,0:mp+1) :: v
  INTEGER status(MPI_STATUS_SIZE)

  CALL MPI_SENDRECV(                                        &
            v(1,mp  ), m, MPI_DOUBLE_PRECISION, above, 0,  &
            v(1,   0), m, MPI_DOUBLE_PRECISION, below, 0,  &
            MPI_COMM_WORLD, status, ierr )
  CALL MPI_SENDRECV(                                        &
            v(1,   1), m, MPI_DOUBLE_PRECISION, below, 1,  &
            v(1,mp+1), m, MPI_DOUBLE_PRECISION, above, 1,  &
            MPI_COMM_WORLD, status, ierr )
  RETURN
END SUBROUTINE update_bc_2
```