COMP/CS 605: Introduction to Parallel Computing
Lecture 17: MPI Communicators & Topologies: Data Distribution & PDE's

Mary Thomas

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

Posted: 03/09/17
Updated: 03/09/17

Table of Contents

MPI SPMD Data Distribution

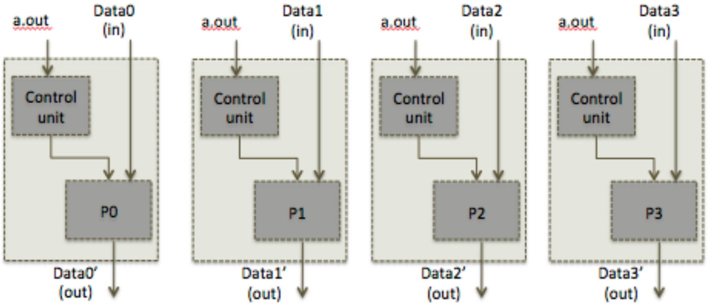
Desgin Considerations for Distribution of Computational Problem

- Depends on problem type:
    - Dense, sparse, banded matrix?
    - Nature of scientific problem being solved: tightly coupled (gas chemistry); natural decamp (2D heat flow, ocean flow); loosely coupled/EP tasks
    - Is there any symmetry in problem being solved that leads to 1,2, or 3D cartesian mapping?
    - Tradeoffs between problem size, computation and communication
    -
- Decomposition Approach for 2D Mat-Mat-Mult
    - Use 1,2, or 3D cartesian mapping
    - Choose Row/Col/Block-Block/Tree
    - Allocate space on each processor P_{ij} for subarrays of A, B, and C.
    - Distribute A,B,C subarrays
    - Calculate local data points for C
    - Exchange A, B data as needed with neighbors
    - Scales for larger array sizes

Distributing the Work (Problem Size)

Using Virtual Topologies to Distribute Data

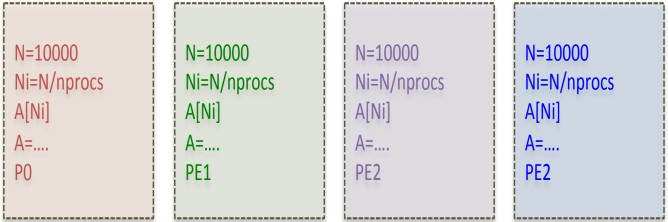
Single Program Multiple Data (SPMD)



Single Program Multiple Data

Each processor gets a copy of the executable, its own set of data (which may or may not be the same), and produces its own results.

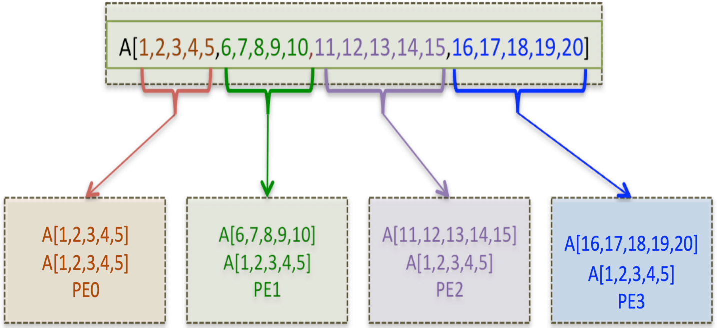
Distributing the Work (Problem Size)



1D Problem Size Distribution

- Example of problem size distribution across 4 PEs.
- Each node loads the max number of elements, computes its own local problem size, allocates an array, and performs some computation.
- MPI_Send/MPI_Recv only needed if collecting results.

Distributing Data (1D Vector)



- 1D vector being distributed to 1D PE (logical) geometry.
- Distributing the ProbSize and Data
- Must be concerned about how the global problem maps onto local PE's
- MPI_Send/MPI_Recv required for data distribution, updates, collection.

Distributing Data (1D Vector)

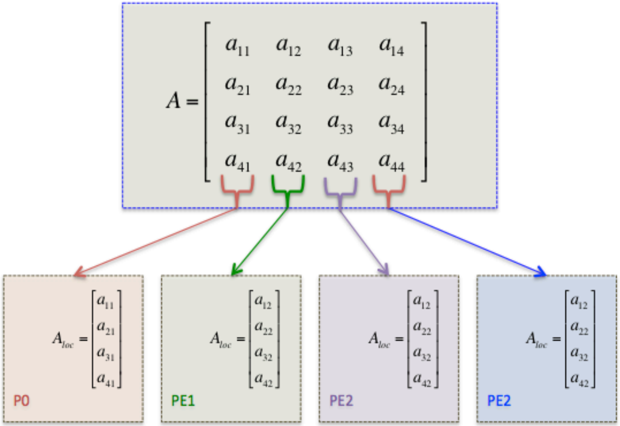
```
%,backgroundcolor=\color{yellow}]
!
!  This file contains a routine for producing a decomposition of a 1-d array
!  when given a number of processors.  It may be used in "direct" product
!  decomposition.  The values returned assume a "global" domain in [1:n]
!
      subroutine MPE_DECOMP1D( n, numprocs, myid, s, e )
      integer n, numprocs, myid, s, e
      integer nlocal
      integer deficit
!
      nlocal  = n / numprocs
      s              = myid * nlocal + 1
      deficit = mod(n,numprocs)
      s              = s + min(myid,deficit)
      if (myid .lt. deficit) then
          nlocal = nlocal + 1
      endif
      e = s + nlocal - 1
      if (e .gt. n .or. myid .eq. numprocs-1) e = n
      return
      end
```

Source: http://www.mcs.anl.gov/research/projects/mpi/usingmpi/examples-usingmpi/intermediate/decomp_f90.html

What about 2D (or 3D) data sets and processor geometries?

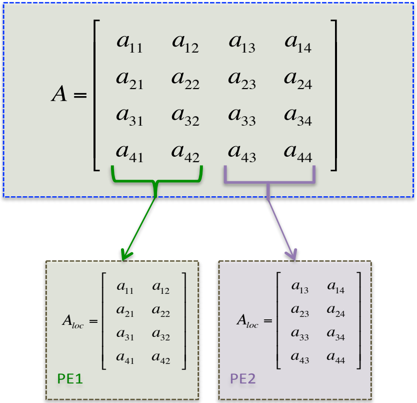
- How will global problem map onto local PE's
- Many possible ways to decompose data and problem: e.g. 1D slabs or 2D blocks?
- Must be concerned about how the global problem maps onto local PE's
- MPI_Send/MPI_Recv required for data distribution, updates, collection.
- Must some understanding of Matrices and Matrix operations.

Data Distribution: 2D Matrix onto 4 PEs in 1D config


$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$
$A_{loc} = \begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \\ a_{41} \end{bmatrix}$
P0
$A_{loc} = \begin{bmatrix} a_{12} \\ a_{22} \\ a_{32} \\ a_{42} \end{bmatrix}$
PE1
$A_{loc} = \begin{bmatrix} a_{12} \\ a_{22} \\ a_{32} \\ a_{42} \end{bmatrix}$
PE2
$A_{loc} = \begin{bmatrix} a_{12} \\ a_{22} \\ a_{32} \\ a_{42} \end{bmatrix}$
PE2

2D (4x4) matrix horizontal data Distribution onto a 1D processor arrangement using vertical slabs and 4 PE's.

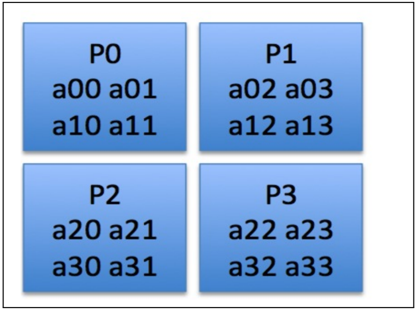
Data Distribution: 2D Matrix onto 2 PEs in 1D config


$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$
$A_{loc} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix}$
PE1
$A_{loc} = \begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \\ a_{33} & a_{34} \\ a_{43} & a_{44} \end{bmatrix}$
PE2

2D (4x4) matrix horizontal data Distribution onto a 1D processor arrangement using vertical slabs and 2 PE’s.

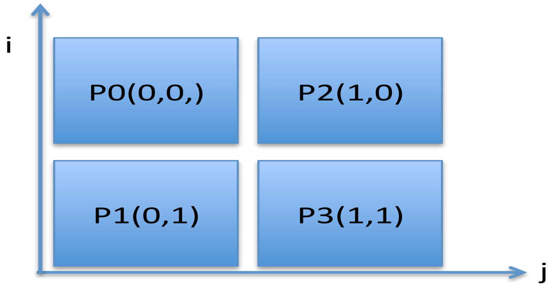
2D "Checkerboard" Decomposition

- Use 2D cartesian mapping for Processors
- Use 2D cartesian mapping of the data
- Allocate space on each processor P_{ij} for subarrays of A, B, and C.
- Distribute A,B,C subarrays
- Calculate local data points for C
- Exchange A, B data as needed with neighbors



REFS: Foster Ch4 [?] and Anghelescu [?]

MPI PE Distr: Cartesian Coordinate System (Block-Block)



NPEs = 4 PE Dimsv= (2x2)
MPI Scheme: 0:NPE-1

- MPI creates the cartesian topology based on 3D mapping
- call to MPI_DIMS_CREATE(nprocs, NDIMS, dims)

Distributing Data (2D Matrix)

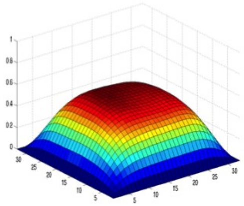
```
! Compute the decomposition
     call fnd2ddecomp( comm2d, nx, sx, ex, sy, ey )
!     print *, "Process ", myid, ":", sx, ex, sy, ey
```

```
! This routine show how to determine the neighbors in a 2-d decomposition of
! the domain. This assumes that MPI_Cart_create has already been called
!
      subroutine fnd2dnbrs( comm2d, & nbrleft, nbrright, nbrtop, nbrbottom )
      integer comm2d, nbrleft, nbrright, nbrtop, nbrbottom
      integer ierr
!
      call MPI_Cart_shift( comm2d, 0,  1, nbrleft,   nbrright, ierr )
      call MPI_Cart_shift( comm2d, 1,  1, nbrbottom, nbrtop,   ierr )
!
      return
      end
!
      subroutine fnd2ddecomp( comm2d, n, sx, ex, sy, ey )
      integer comm2d, n, sx, ex, sy, ey
      integer dims(2), coords(2), ierr
      logical periods(2)
!
      call MPI_Cart_get( comm2d, 2, dims, periods, coords, ierr )

      call MPE_DECOMP1D( n, dims(1), coords(1), sx, ex )
      call MPE_DECOMP1D( n, dims(2), coords(2), sy, ey )
!
      return
      end
```

Source: http://www.mcs.anl.gov/research/projects/mpi/usingmpi/examples-usingmpi/intermediate/\http://www.mcs.anl.gov/research/projects/mpi/usingmpi/examples-usingmpi/intermediate/fndnbr2d_f90.html

MPI 2D Jacobian Iterative Solver



Partial Differential Equations

- 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 solvers
  - Gauss elimination
  - LU decomposition
- Iterative solvers
  - Basic iterative solvers
    - Jacobi
    - Gauss-Seidel
    - Successive over-relaxation
    - Relaxation methods are iterative methods for solving systems of equations, including nonlinear systems.
    - Relaxation methods were developed for solving large sparse linear systems using finite-difference
  - 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) $= sin\,(\pi x)$ $\quad 0 <= x <= 1$

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

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

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

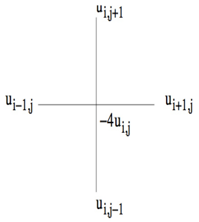
Jacobi Iterative Scheme

Jacobi Iteration - Finite Difference Approximation

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

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

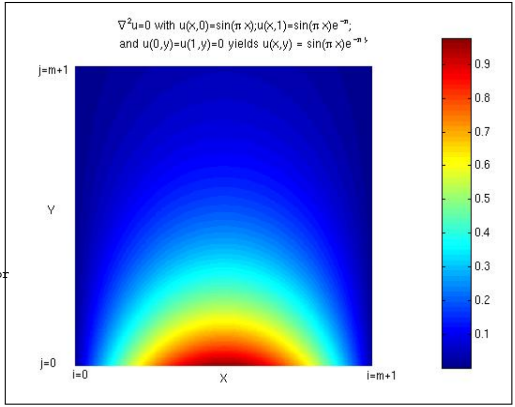
```
c => u(1:m  ,1:m  )    ! i  ,j  Current/Central
                              ! for 1<=i<=m; 1<=j<=m
n => u(1:m  ,2:m+1) ! i  ,j+1  North (of Current)
e => u(2:m+1,1:m )  ! i+1,j    East  (of Current)
w => u(0:m-1,1:m  ) ! i-1,j    West  (of Current)
s => u(1:m  ,0:m-1)  ! i  ,j-1  South (of Current)
```



Source: http://www.eng.utah.edu/~cs3200/notes/cs3200-Finite-Differences.pdf

Serial Jacobi Iterative Scheme - Boundary Conditions

```
  SUBROUTINE bc(u, m)
! 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)
    IMPLICIT NONE
    INTEGER m, j
    REAL(real8), DIMENSION(0:m+1,0:m+1) :: u
    REAL(real8), DIMENSION(:,:), POINTER :: c
    REAL(real8), DIMENSION(0:m+1) :: y0
    y0 = sin(3.141593*(/(j,j=0,m+1)/)/(m+1))
    u = 0.0d0     ! at x=0,1; all y plus initialize interior
    u(:,  0) = y0               ! at y = 0; all x
    u(:,m+1) = y0*exp(-3.141593) ! at y = 1; all x
    RETURN
  END SUBROUTINE bc
```



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

Serial Jacobi Iterative Scheme - Boundary Conditions

```
PROGRAM Jacobi
USE serial_jacobi_module
REAL(real8), DIMENSION(:,:), POINTER :: c, n, e, w, s

write(*,*)'Enter matrix size, m:'
read(*,*)m
! start timer, measured in seconds
CALL cpu_time(start_time)
 ! mem for unew, u
ALLOCATE ( unew(m,m), u(0:m+1,0:m+1) )

c => u(1:m  ,1:m  )    ! i  ,j  Current/Central
                               ! for 1<=i<=m; 1<=j<=m
n => u(1:m  ,2:m+1) ! i  ,j+1  North (of Current)
e => u(2:m+1,1:m )  ! i+1,j    East  (of Current)
w => u(0:m-1,1:m  ) ! i-1,j    West  (of Current)
s => u(1:m  ,0:m-1)  ! i  ,j-1  South (of Current)

CALL bc(u, m)         ! set up boundary values
```

```
 ! iterate until error below threshold
DO WHILE (gdel > tol)
  ! increment iteration counter
  iter = iter + 1
  IF(iter > 5000) THEN
    WRITE(*,*)'Iteration terminated (exceeds 5000)'
    STOP                         ! nonconvergent solution
  ENDIF
  unew = ( n + e + w + s )*0.25 ! new solution, Eq. 3
  gdel = MAXVAL(DABS(unew-c))    ! find local max error
  IF(MOD(iter,10)==0) WRITE(*,"('iter,gdel:',i6,e12.4)")iter,gdel
  c = unew                       ! update interior u
ENDDO

CALL CPU_TIME(end_time)      ! stop timer
PRINT *,'Total cpu time =',end_time - start_time,' x 1'
PRINT *,'Stopped at iteration =',iter
PRINT *,'The maximum error =',gdel

write(40,"(3i5)")m,m,1
write(41,"(6e13.4)")u
DEALLOCATE (unew, u)

END PROGRAM Jacobi
```

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

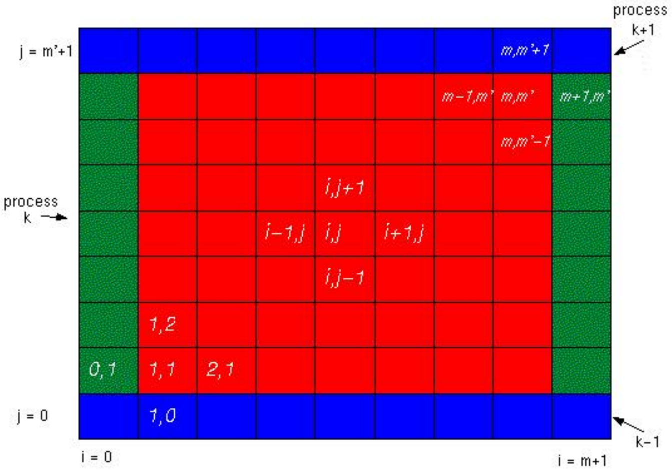
Parallel Jacobi Approach

- 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:

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

- 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/`

Ghost Cell Layout

Distributed-memory parallel programming w

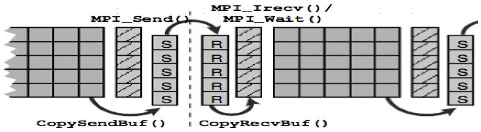


Figure 9.9: Halo communication for the Jacobi solver (illustrat
along one of the coordinate directions. Hatched cells are ghost la
("S") belong to the intermediate receive (send) buffer. The latter
directions. Note that halos are always provided for the grid that g
upcoming sweep. Fixed boundary cells are omitted for clarity.

Source: F. B. Kjolstad and M. Snir, Ghost Cell Pattern, in Proceedings of the 2010 Workshop on Parallel Programming Patterns
`http://people.csail.mit.edu/fred/ghost_cell.pdf`

Ghost Cell Deadlock-Free Exchange

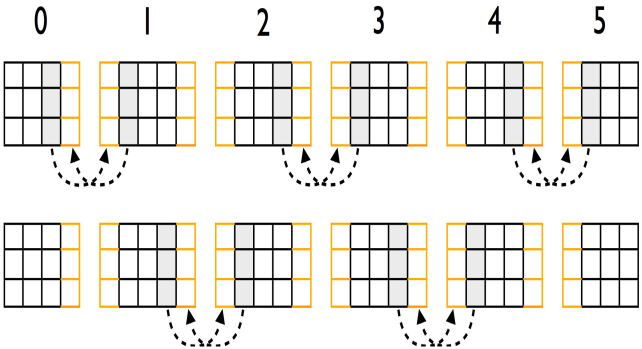


Figure 5: Deadlock-free border exchanges

Source: F. B. Kjolstad and M. Snir, Ghost Cell Pattern, in Proceedings of the 2010 Workshop on Parallel Programming Patterns
`http://people.csail.mit.edu/fred/ghost_cell.pdf`

Parallel Jacobi Code

```
PROGRAM Jacobi
USE types_module;   USE jacobi_module;
USE mpi_module
REAL, DIMENSION(:,:), POINTER :: c, n, e, w, s
CALL MPI_Init(ierr)  ! starts MPI
! get current process id
CALL MPI_Comm_rank(MPI_COMM_WORLD, k, ierr)
! get # procs from env
CALL MPI_Comm_size(MPI_COMM_WORLD, p, ierr)
if( k == 0) then
  write(*,*)'Enter matrix size, m:'   ;
  read(*,*)m
endif
CALL MPI_Bcast(m, 1, MPI_INTEGER, 0, &
              MPI_COMM_WORLD, ierr)
 ! start timer, measured in seconds
CALL cpu_time(start_time)
mp = m/p      ! columns for each proc
! mem for vnew, v
ALLOCATE ( vnew(m,mp), v(0:m+1,0:mp+1) )
c => v(1:m  ,1:mp  )      ! i  ,j
   ! for 1<=i<=m; 1<=j<=mp
n => v(1:m  ,2:mp+1)   ! i  ,j+1
e => v(2:m+1,1:mp  )   ! i+1,j
w => v(1:m  ,0:mp-1)   ! i-1,j
s => v(0:m-1,1:mp  )    ! i  ,j-1
```

```
CALL bc(v, m, mp, k, p)         ! set up boundary values
! determines domain border flags
CALL neighbors(k, below, above, p)
! iterate until error below threshold
DO WHILE (gdel > tol)
  iter = iter + 1      ! increment iteration counter
  IF(iter > 5000) THEN
    WRITE(*,*)'Iteration terminated (exceeds 5000)'
    STOP       ! nonconvergent solution
  ENDIF
  vnew = ( n + e + w + s )*0.25   ! new solution
  ! find local max error
  del = MAXVAL(DABS(vnew-c))
  IF(MOD(iter,10)==0)   &
       WRITE(*,"('k,iter,del:',i4,i6,e12.4)")k,iter,del
  IF(m==4 .and. MOD(iter,10) == 0)    &
     CALL print_mesh(v,m,mp,k,iter)
  c = vnew                       ! update interior v
  CALL MPI_Allreduce( del, gdel, 1,    &
        MPI_DOUBLE_PRECISION, MPI_MAX,    &
        MPI_COMM_WORLD, ierr )  ! find global max error

  CALL update_bc_2( v, m, mp, k, below, above)
! CALL update_bc_1( v, m, mp, k, below, above)
ENDDO
```

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
```