# *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:***

- ***F90 is used.***
- ***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***
- ***Subroutine printmesh may be used to print local solution for tiny cases (like 4x4)***
- ***Pointer arrays c, n, e, w, and s point to various parts of 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.***

```
! Solve Laplace equation using Jacobi iteration method
! Kadin Tseng, Boston University, November 1999

MODULE jacobi_module
  IMPLICIT NONE
  INTEGER, PARAMETER :: real4 = selected_real_kind(6,37)
  INTEGER, PARAMETER :: real8 = selected_real_kind(15,307)
  REAL(real8), DIMENSION(:,:), ALLOCATABLE         :: vnew
  REAL(real8), DIMENSION(:,:), ALLOCATABLE, TARGET :: v  ! solution array
  REAL(real8) :: tol=1.d-4, del, gdel=1.0d0
  REAL(real4) :: start_time, end_time
  INCLUDE "mpif.h"  !! This brings in pre-defined MPI constants, ...
  INTEGER :: p, ierr, below, above, k, m, mp, iter=0
  PUBLIC

CONTAINS
  SUBROUTINE bc(v, m, mp, k, p)
```

```
! 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, mp, k, p, j
    REAL(real8), DIMENSION(0:m+1,0:mp+1) :: v
    REAL(real8), DIMENSION(:,:), POINTER :: c
    REAL(real8), DIMENSION(0:m+1) :: y0

    y0 = sin(3.141593*(/(j,j=0,m+1)/)/(m+1))

    IF(p > 1) THEN
      u = 0.0d0
      IF (k == 0  ) v(:,   0) = y0
      IF (k == p-1) v(:,mp+1) = y0*exp(-3.141593)
    ELSE
      v = 0.0d0
      v(:,0) = y0
      v(:,m+1) = y0*exp(-3.141593)
    END IF
    RETURN
  END SUBROUTINE bc

  SUBROUTINE neighbors(k, below, above, p)
    IMPLICIT NONE
    INTEGER :: k, below, above, p

    IF(k == 0) THEN
      below = -1        ! tells MPI not to perform send/recv
      above = k+1
    ELSE IF(k == p-1) THEN
      below = k-1
      above = -1        ! tells MPI not to perform send/recv
    ELSE
      below = k-1
      above = k+1
    ENDIF

    RETURN
  END SUBROUTINE neighbors

  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

  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

  SUBROUTINE print_mesh(v,m,mp,k,iter)
  IMPLICIT NONE
  INTEGER :: m, mp, k, iter, i, out
  REAL(real8), DIMENSION(0:m+1,0:mp+1) :: v

    out = 20 + k
    do i=0,m+1
      write(out,"(2i3,i5,' => ',4f10.3)")k,i,iter,v(i,:)
    enddo
    write(out,*)'++++++++++++++++++++++++++++++++++++++++++++++++++++++++'

    RETURN
  END SUBROUTINE print_mesh
END MODULE jacobi_module

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

CALL MPI_Init(ierr)                              ! starts MPI
CALL MPI_Comm_rank(MPI_COMM_WORLD, k, ierr)    ! get current process id
CALL MPI_Comm_size(MPI_COMM_WORLD, p, ierr)    ! get # procs from env

if( k == 0) then
  write(*,*)'Enter matrix size, m:'
  read(*,*)m
endif
CALL MPI_Bcast(m, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
CALL cpu_time(start_time)        ! start timer, measured in seconds

mp = m/p                         ! columns for each proc
ALLOCATE ( vnew(m,mp), v(0:m+1,0:mp+1) )  ! mem for vnew, v

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

CALL neighbors(k, below, above, p) ! determines neighbors of current process

DO WHILE (gdel > tol)            ! iterate until error below threshold
  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
  del = MAXVAL(DABS(vnew-c))     ! find local max error
  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

IF(k == 0) THEN
  CALL CPU_TIME(end_time)            ! stop timer
  PRINT *,'Total cpu time =',end_time - start_time,' x',p
  PRINT *,'Stopped at iteration =',iter
  PRINT *,'The maximum error =',gdel
ENDIF

DEALLOCATE (vnew, v)

CALL MPI_Finalize(ierr)

END PROGRAM Jacobi
```