Solve Poisson Equations on a Grid

- Goal: explain communication patterns in grid-based problems
  - Domain decomposition
  - Variations on MPI send/receive operations
- Approach:
  - Finite-difference scheme on the Poisson problems
  - Topologies
    - Cartesian topology (decomposition in the x,y,z directions)
  - Use Fortran rather than C

The Poisson Problem

Equations:

$d^2u / d\,x^2 \;+\; d^2u / d\,y^2 \;=\; f\,(x,y)$ (1)   in the interior

$u(x,y) \;=\; g(x,y)$ (2)   on the physical boundary

Approximation:

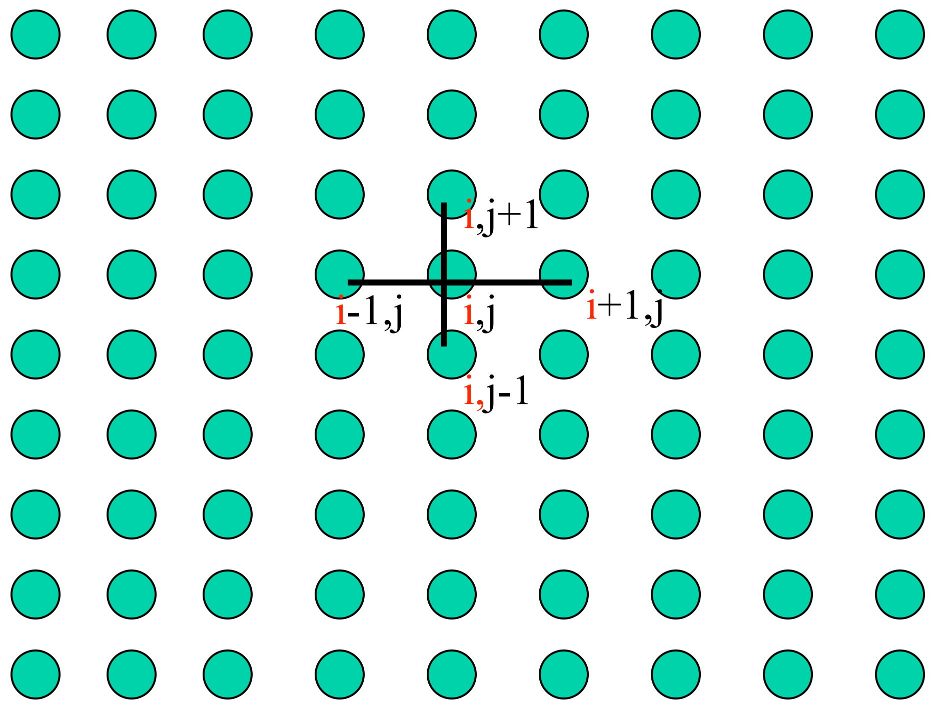
solve equations on a square mesh (grid) *not* everywhere

$x_i = i/(n+1), \quad i = 0, \ldots, n+1,$

$y_i = j/(n+1), \quad j = 0, \ldots, n+1$

$h = 1/(n+1)$

Five-Point Stencil Approximation



Five-point stencil approximation for 2-D Poisson Problem (n=7)

The Poisson Problem (continue)

Finite-difference solution for Eq. (1):

$$u_{i-1,j} + u_{i,j+1} + u_{i,j-1} + u_{i+1,j} - 4u_{i,j} / h^2 = f_{i,j} \qquad (3)$$

Use Jacobi iteration to solve for u everywhere on the mesh

Rewrite (3) as

$$u_{i,j} = (u_{i-1,j} + u_{i,j+1} + u_{i,j-1} + u_{i+1,j} - h^2 f_{i,j})/4$$

Iterate and update with

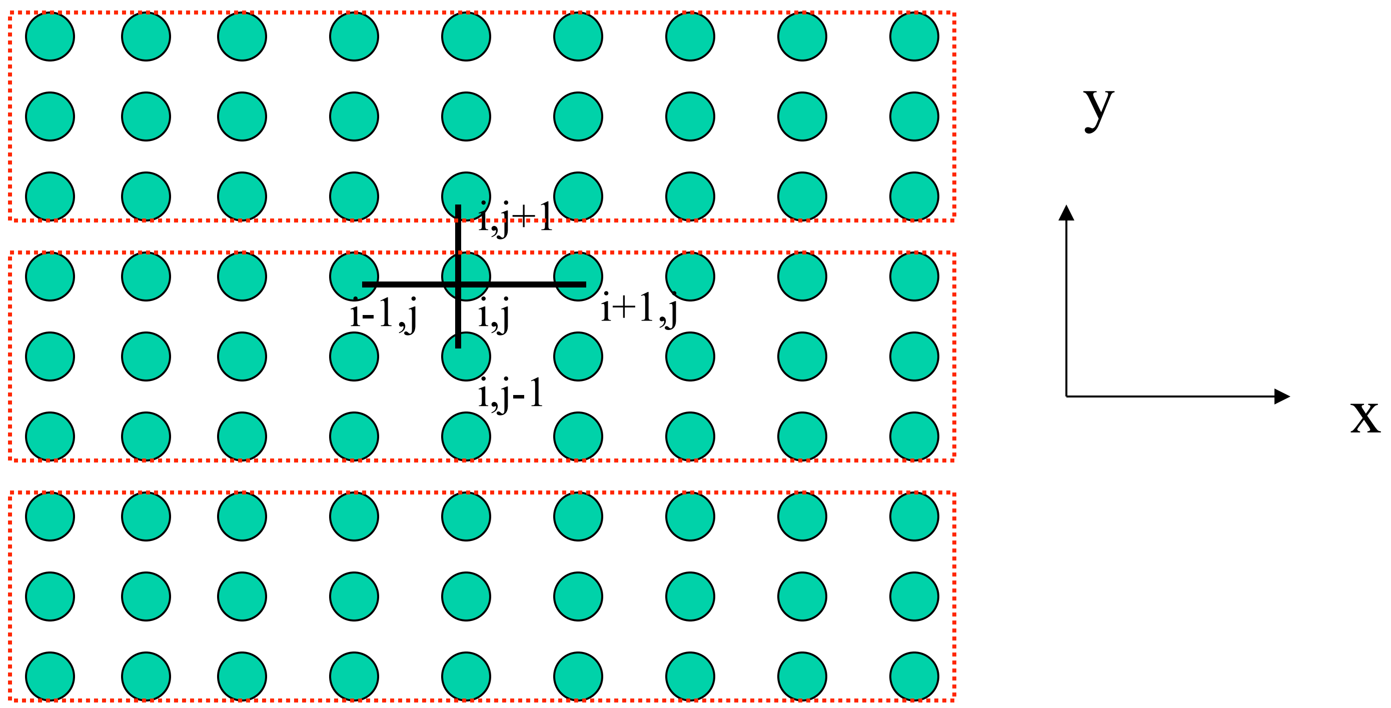
$$u^{k+1}_{i,j} = (u^{k}_{i-1,j} + u^{k}_{i,j+1} + u^{k}_{i,j-1} + u^{k}_{i+1,j} - h^2 f_{i,j})/4$$

The Poisson Problem (continue)

The sequential (*single* process) code:

```
integer i, j, nx, ny
double precision u(0:nx+1,0:ny+1), unew(0:nx+1,0:ny+1)

do 10 j=1, ny
   do 10 i=1, nx

      unew(i,j) = 0.25 * (u(i-1,j)+u(i,j+1)+u(i,j-1)+u(i+1,j)) - h * h * f(i,j)

10 continue
```

1D Domain Decomposition



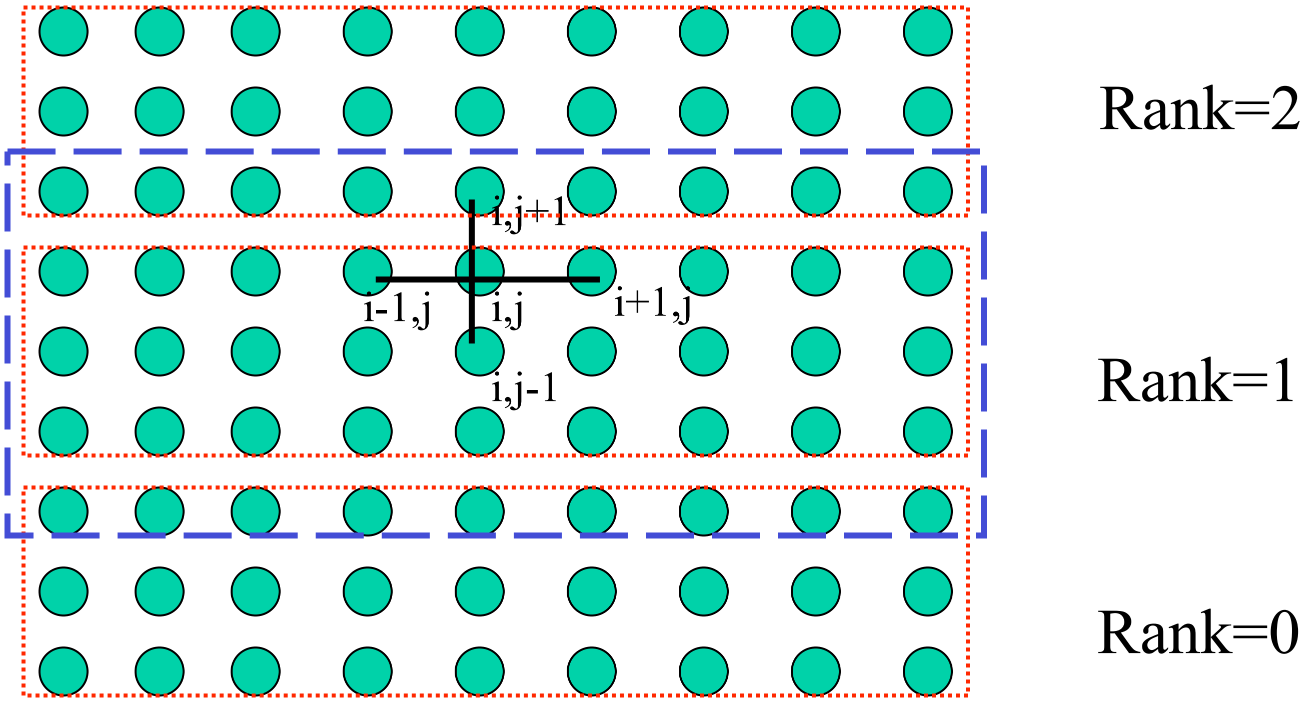
1D domain decomposition along the y direction

The Poisson Problem (continue)

The code on *each* process with *1D domain decomposition* along y/j:

```
  integer i, j, nx, ny
C  s:e indicates the value of j that this process is responsible for.
  double precision u(0:nx+1, s:e), unew(0:nx+1, s:e)

  do 10 j=s,e
    do 10 i=1, nx

      unew(i,j) = 0.25 * (u(i-1,j)+u(i,j+1)+u(i,j-1)+u(i+1,j)) - h * h * f(i,j)

 10 continue
```

1D Domain Decomposition



The computational domain, with 1-point ghost zone, for one of the processes

The Poisson Problem (continue)

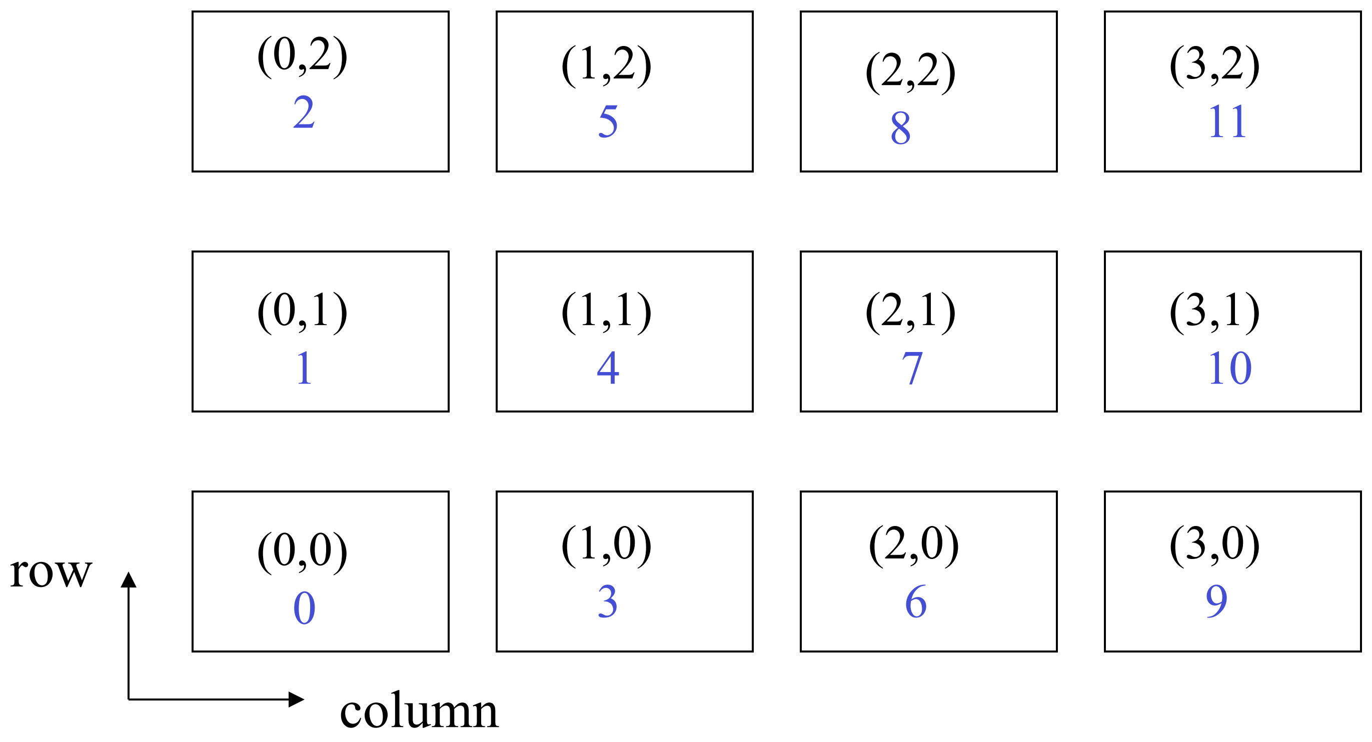
The code on each process with 1D domain decomposition along y/j with *1-point ghost zone*:

```
  integer i, j, nx, ny
C need grid points from top and bottom processes.
    double precision u(0:nx+1, s-1:e+1), unew(0:nx+1, s-1:e+1)

C the do-loop bound is still from s to e along the y direction
    do 10 j=s,e
     do 10 i=1, nx
       unew(i,j) = 0.25 * (u(i-1,j)+u(i,j+1)+u(i,j-1)+u(i+1,j)) - h * h * f(i,j)
   10 continue
```

Topologies

- Topologies is the description of how the processes in a parallel computer are connected to one another (or more precisely, of the interconnection network).
- How a particular application map to the physical topology of the parallel computer depends on the vendor
- One MPI application topology is the Cartesian topology which is simply a decomposition in the x,y,z directions

2D Cartesian Decomposition



Note: The coordinate is (column, row) which would be returned by MPI_Get_coords. The rank is returned by MPI_Cart_Rank.

2D Cartesian Decomposition (code)

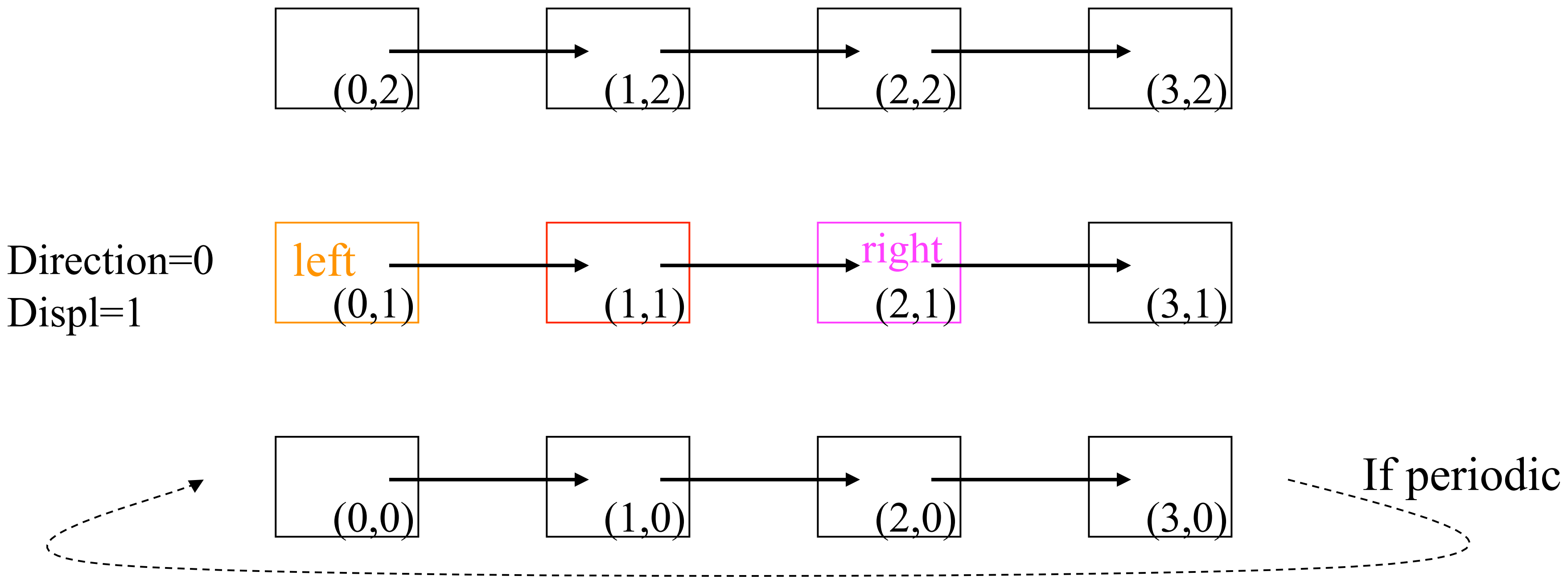
```
dims(1)      =    4
dims(2)      =    3
periods(1)   =    .false.
periods(2)   =    .false.
reorder      =    .true.
ndim         =    2
call MPI_CART_CREATE( MPI_COMM_WORLD, ndim,
                      dims, periods, reorder, comm2d, ierr)
```

Note: *reorder = .true.* allows MPI to map the processes to the elements of the decomposition optimally

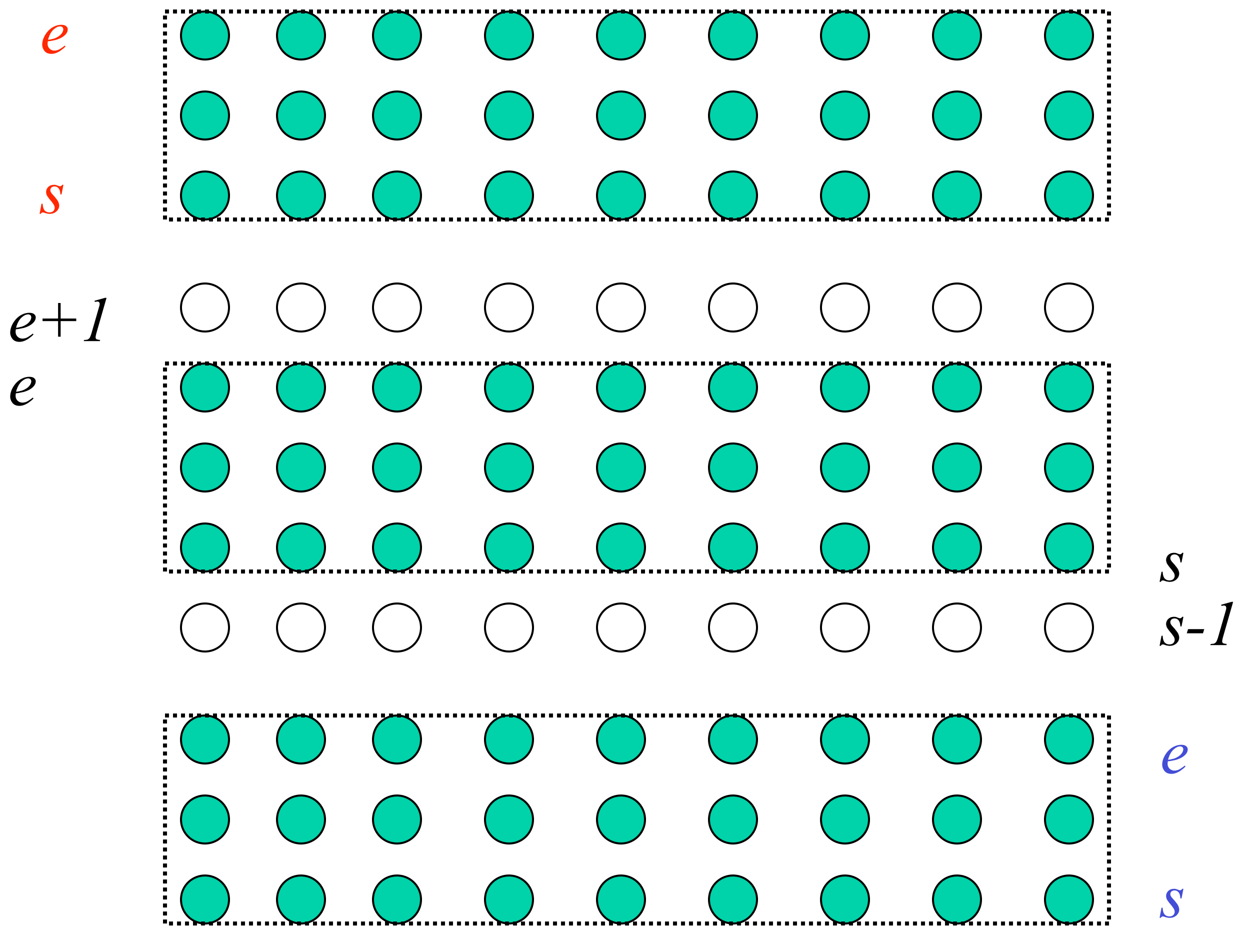
MPI_Cart_Shift

*Int MPI_Cart_shift(MPI_Comm comm, int direction, int displ, int *src, int*dest)*

MPI_Cart_Shift() is used to find the neighbors. For example, shift by one in the 1st dimension. (1,1)'s destination and source are (2,1) and (0,1), respectively



1D Domain Decomposition (Transfer Data)



Main Program

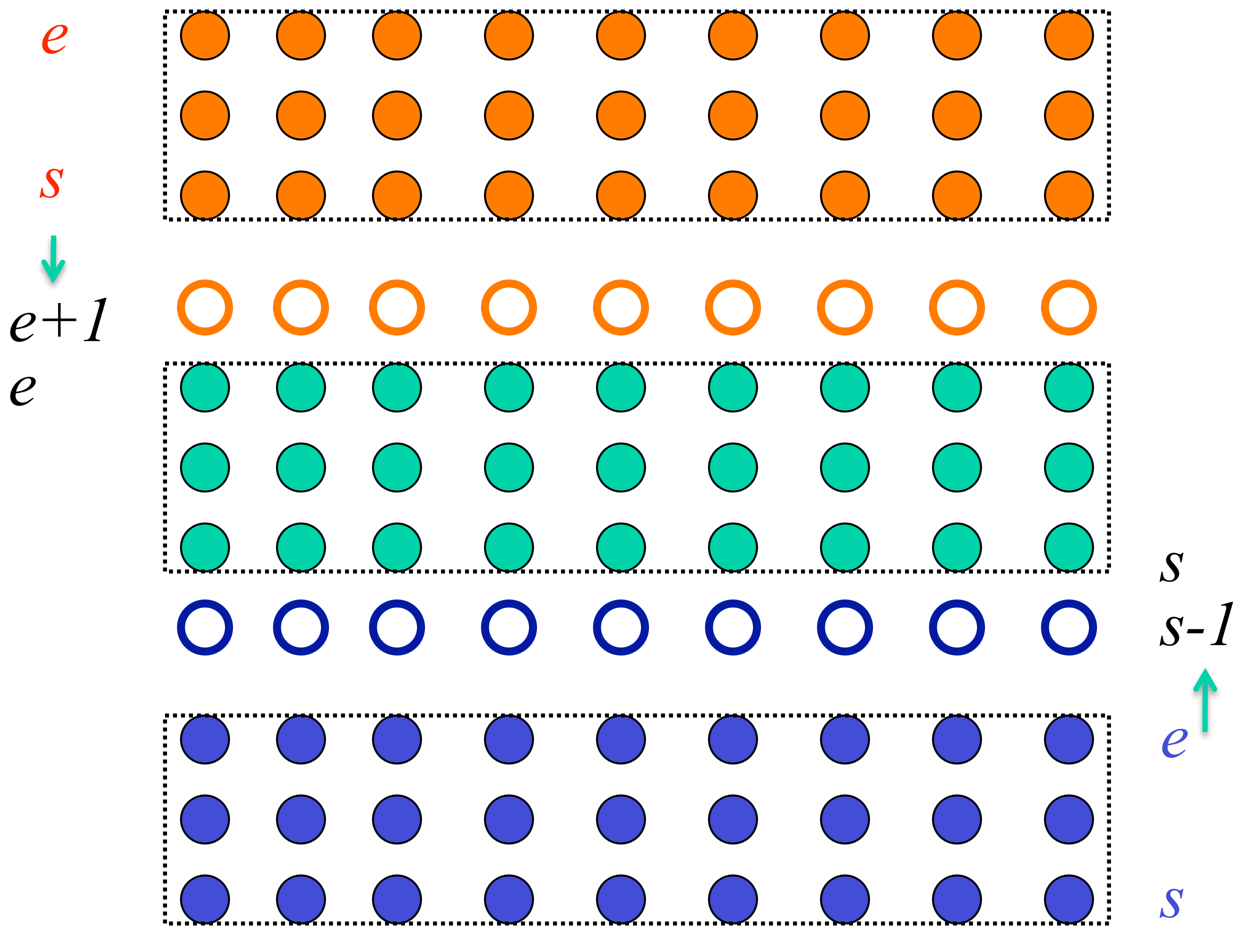
```
c*******************************************************************
c oned.f - a solution to the Poisson problem using Jacobi
c interation on a 1-d decomposition
c
C The size of the domain is read by processor 0 and broadcast to
c all other processors. The Jacobi iteration is run until the
c change in successive elements is small or a maximum number of
c iterations is reached. The difference is printed out at each step.
c*******************************************************************
program main

include "mpif.h"
integer maxn parameter (maxn = 128)
double precision a(maxn,maxn), b(maxn,maxn), f(maxn,maxn)
integer nx, ny
```

```
      integer myid, numprocs, ierr
      integer comm1d, nbrbottom, nbrtop, s, e, it, maxit
      double precision diff, diffnorm, dwork
      double precision t1, t2
      double precision MPI_WTIME
      external MPI_WTIME
      external diff

      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )

      if (myid .eq. 0) then
c
c Get the size of the problem
c
      print *, 'Enter nx'
      read *, nx
      endif
```

```
 call MPI_BCAST(nx,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
 ny = nx
c
c Get a new communicator, comm1d, for a 1D decomposition of the domain
c
 call MPI_CART_CREATE( MPI_COMM_WORLD, 1, numprocs,
 $                                  .false., .true., comm1d, ierr )
c
c Get my position, myid, in this communicator, comm1d, and my neighbors
c at the top, nbrtop and the bottom, nbrbottom
c
 call MPI_COMM_RANK( comm1d, myid, ierr )
 call MPI_Cart_shift( comm1d, 0, 1, nbrbottom, nbrtop, ierr )
c
c Compute the actual decomposition: s and e
 call MPE_DECOMP1D( ny, numprocs, myid, s, e )
c
```

```
c Initialize the right-hand-side (f) and the initial solution guess (a)
c
      call onedinit( a, b, f, nx, s, e )
c
c Actually do the computation. Note the use of a collective operation to
c check for convergence, and a do-loop to bound the number of iterations.
c
      call MPI_BARRIER( MPI_COMM_WORLD, ierr )
      t1 = MPI_WTIME()
      do 10 it=1, maxit
C     get ghost points
        call exchng1( a, nx, s, e, comm1d, nbrbottom, nbrtop )
      c perform one jacobi “sweep”
        call sweep1d( a, f, nx, s, e, b )
        call exchng1( b, nx, s, e, comm1d, nbrbottom, nbrtop )
        call sweep1d( b, f, nx, s, e, a )
        dwork = diff( a, b, nx, s, e )

        call MPI_Allreduce( dwork, diffnorm, 1, MPI_DOUBLE_PRECISION,
        $                              MPI_SUM, comm1d, ierr )
      if (diffnorm .lt. 1.0e-5) goto 20
c
      if (myid .eq. 0) print *, 2*it, ' Difference is ', diffnorm
10     continue

      if (myid .eq. 0)
      print *, 'Failed to converge'

20     continue
      t2 = MPI_WTIME()

      if (myid .eq. 0) then
      print *, 'Converged after ', 2*it, ' Iterations in ', t2 - t1,
      $ ' secs ' endif
c
      call MPI_FINALIZE(ierr)
      end
```

1D Domain Decomposition (Transfer Data)



Exchange Data for Ghost Points (I) Using blocking sends and receives

```
      subroutine exchng1( a, nx, s, e, comm1d, nbrbottom, nbrtop )
      include 'mpif.h'
      integer nx, s, e
      double precision a(0:nx+1,s-1:e+1)
      integer comm1d, nbrbottom, nbrtop
      integer status(MPI_STATUS_SIZE), ierr
c
      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, status, 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, status, ierr )
      return
      end
```

Exchange Data for Ghost Points (II) Using paired blocking sends and receives

```
 …
 integer comm1d, nbrbottom, nbrtop, rank, coord
…
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, status, 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, status, ierr )
 else
   call MPI_RECV(a(1,s-1), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0,
   &                        comm1d, status, 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, status, ierr )
   call MPI_SEND(a(1,s), nx, MPI_DOUBLE_PRECISION, nbrbottom, 1, comm1d,ierr)
 …
```

Exchange Data for Ghost Points (III) Using send-recv

```
      subroutine exchng1( a, nx, s, e, comm1d, nbrbottom, nbrtop )
      include 'mpif.h'
      integer nx, s, e
      double precision a(0:nx+1,s-1:e+1)
      integer comm1d, nbrbottom, nbrtop
      integer status(MPI_STATUS_SIZE), ierr
c
      call MPI_SENDRECV(
      &                   a(1,e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0,
      &                   a(1,s-1), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0,
      &                   comm1d, status, ierr )

      call MPI_SENDRECV( & a(1,s), nx, MPI_DOUBLE_PRECISION, nbrbottom, 1,
      &                   a(1,e+1), nx, MPI_DOUBLE_PRECISION, nbrtop, 1,
      &                   comm1d, status, ierr )
      return
      end
```

Exchange Data for Ghost Points (IV) Using nonblocking operations

```
      subroutine exchng1( a, nx, s, e, comm1d, nbrbottom, nbrtop )
      include 'mpif.h'
      integer nx, s, e
      double precision a(0:nx+1,s-1:e+1)
      integer comm1d, nbrbottom, nbrtop
      integer status_array(MPI_STATUS_SIZE,4), ierr, req(4)
c
      call MPI_IRECV(a(1,s-1), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0,
     &                   comm1d, req(1), ierr )
      call MPI_IRECV(a(1,e+1), nx, MPI_DOUBLE_PRECISION, nbrtop, 1,
     &                   comm1d, req(2), ierr )

      call MPI_ISEND( a(1,e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0,
     &                   comm1d, req(3), ierr)
      call MPI_ISEND(a(1,s), nx, MPI_DOUBLE_PRECISION, nbrbottom, 1,
     &                   comm1d, req(4), ierr)
      return
      end
```

Compute Decomposition

```
c This file contains a routine for producing a decomposition of a 1-d array
c when given a number of processors. It may be used in "direct" product
c decomposition. The values returned assume a "global" domain in [1:n]
c  n, numprocs, myid are input while s and e are output
      subroutine MPE_DECOMP1D( n, numprocs, myid, s, e )
      integer n, numprocs, myid, s, e, nlocal, deficit
c
      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
```

Calculate the Difference of Two Successive Solutions

```
      double precision function diff( a, b, nx, s, e )
      integer nx, s, e
      double precision a(0:nx+1, s-1:e+1), b(0:nx+1, s-1:e+1)
c
      double precision sum integer i, j
c
      sum = 0.0d0
      do 10 j=s,e
      do 10 i=1,nx
      sum = sum + (a(i,j) - b(i,j)) ** 2
10 continue
c
      diff = sum
      return
      end
```

Initialization Routine

```
    subroutine onedinit( a, b, f, nx, s, e )
    integer nx, s, e, i,j
    double precision a(0:nx+1, s-1:e+1), b(0:nx+1, s-1:e+1),
    &                f(0:nx+1, s-1:e+1)
c
    do 10 j=s-1,e+1
      do 10 i=0,nx+1
       a(i,j) = 0.0d0
       b(i,j) = 0.0d0
       f(i,j) = 0.0d0
10 continue
```

```
c Handle boundary conditions
        do 20 j=s,e
           a(0,j) = 1.0d0
           b(0,j) = 1.0d0
           a(nx+1,j) = 0.0d0
           b(nx+1,j) = 0.0d0
20 continue
           if (s .eq. 1) then
              do 30 i=1,nx
                a(i,0) = 1.0d0
                b(i,0) = 1.0d0
30 continue
        endif
        return
        end
```

Jacobi Sweep

```
c Perform a Jacobi sweep for a 1-d decomposition.
c Sweep from a into b
c
      subroutine sweep1d( a, f, nx, s, e, b )
      integer nx, s, e, i,j
      double precision a(0:nx+1,s-1:e+1), f(0:nx+1,s-1:e+1),
      + b(0:nx+1,s-1:e+1), h
```

```
       h = 1.0d0 / dble(nx+1)
      do 10 j=s, e
        do 10 i=1, nx
           b(i,j) = 0.25 * (a(i-1,j)+a(i,j+1)+a(i,j-1)+a(i+1,j)) -
      $             h * h * f(i,j)
 10   continue
      return
      end
```

#3 Homework for MPI

- In the header, list your name, email address, general description of the code. Inside code, add sufficient comments on each nontrivial section

# 3 Homework for MPI

- Solve a 2D wave equation with a finite-difference scheme
  - Wave equation: $-\phi_{,tt} / c^2 + \phi_{,xx} + \phi_{,yy} + \phi_{,zz} = 0$
  - Explicit *finite differencing* (centered, second order)

$$\phi^{n+1}_{i,j} = 2\phi^{n}_{i,j} - \phi^{n-1}_{i,j}$$

$$+ \Delta t^2/\Delta x^2(\phi^{n}_{i+1,j} -2\,\phi^{n}_{i,j} + \phi^{n}_{i-1,j})$$

$$+ \Delta t^2/\Delta y^2(\phi^{n}_{i,j+1} -2\,\phi^{n}_{i,j} + \phi^{n}_{i,j-1})$$

  - Initial condition: ϕ (i,j) = sin(i*dx)*cos(j*dy), where dx = 2 π/GridSizeX, dy = 2 π/GridSizeY.
  - Free boundary condition along the x direction and periodic along the y direction
  - Parameters: GridSizeX=512, GridSizeY=1024, c=1; use 2D domain decomposition with 2 and 4 processes in x and y directions, respectively; run 10 time steps; i is from 0 to 511 and j is from 0 to 1023.
  - Compare the parallel code with the serial code in performance and accuracy. You are encouraged to vary the number of grid points along x and y to analyze the performance scalability.