## MPI Part 1

- References:

  **Using MPI.** Gropp, Lusk Skjellum

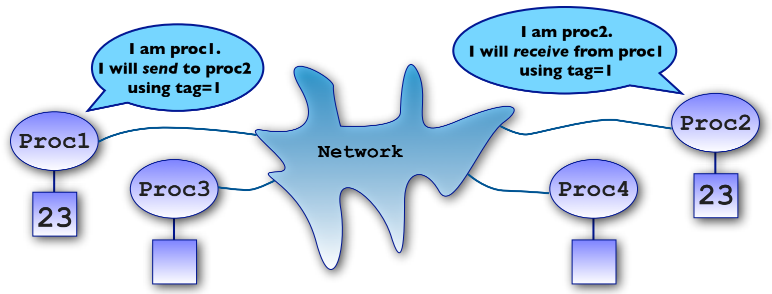
  http://www.mpi-forum.org/docs/mpi-11-html/node182.html

- What is MPI?

  1. MPI allows a collection of processes to communicate with messages.
  2. MPI is a **library** of subroutines called from Fortran, C and C++. Programs are compiler with ordinary compilers and linked with the MPI library.
  3. MPI is a **specification** which is independent from particular implementations. An MPI program should be portable to any vendors hardware that supports MPI.

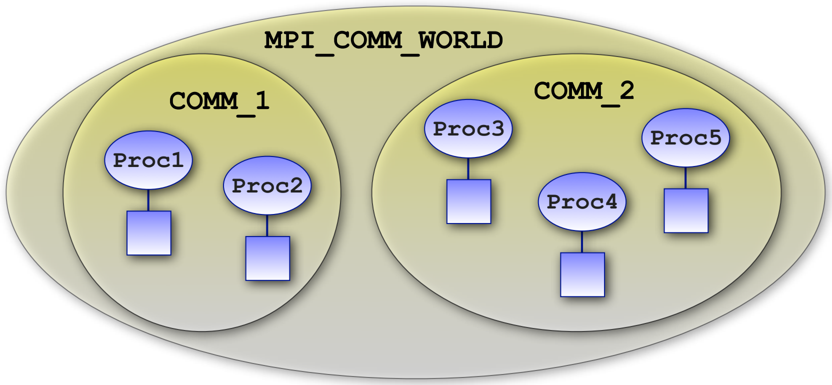
## A minimal message-passing model

- The processes execute in parallel and can have **separate address spaces**.
- **Communication is cooperative.** A message requires one process to execute a send command, and one process to execute a receive command.
- Information from one process's address space (memory) is transfered to another address space (memory) using a **message**.
- The two processes involved in the communication must **agree upon a message tag** to distinguish a message from other messages.


I am proc1.
I will *send* to proc2 using tag=1
I am proc2.
I will *receive* from proc1 using tag=1
Proc1
23
Proc3
Network
Proc4
Proc2
23

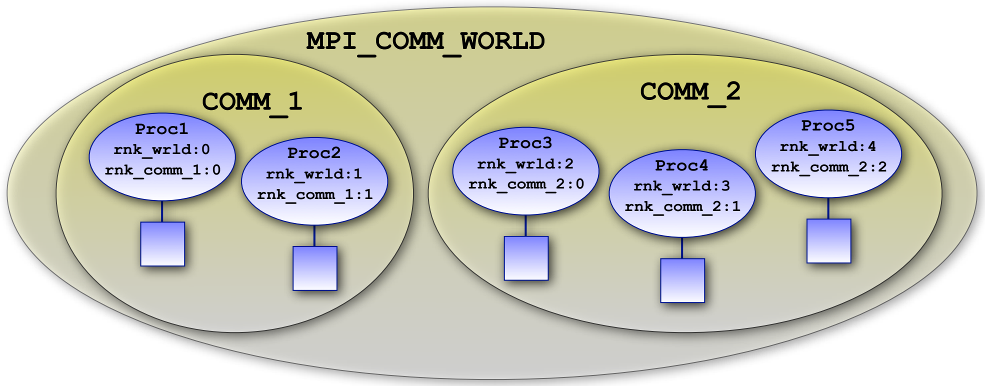
## Communicators

- ✦ Groups of processes are called communicators.
  - The default communicator is called **MPI_COMM_WORLD**. This communicator contains all the processes in the current MPI universe.
  - MPI allows for the formation of communicators within the global communicator.
  - Message tags are defined within the context of a communicator.



## Rank

- ✦ Processes are identified within a communicator by their rank
  - Rank is an integer
  - Rank defined within the context of a communicator.
  - If a communicator contains n processes, then the ranks are integers from 0 to n-1.



## The “hello world” Program

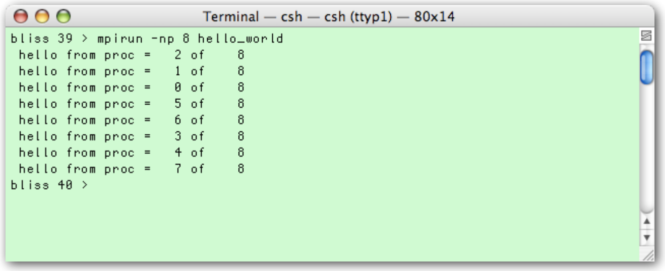
- ✦ Important features of the hello_world program
  1. Use the **`mpi`** module, or include the include file called **`mpif.h`**
  2. Initialize the MPI environment.
  3. Determine how many processes are in the current MPI environment.
  4. Determine rank within the **`MPI_COMM_WORLD`** communicator
  5. Terminate the MPI environment

```
PROGRAM hello_world
USE mpi

IMPLICIT NONE
INTEGER :: npe_wrld, &! number of processes within the world communicator
           rnk_wrld, &! rank of process within the world communicator
           ierr

CALL MPI_INIT (ierr)                                ! initialize MPI environment
CALL MPI_COMM_SIZE (MPI_COMM_WORLD,npe_wrld,ierr) ! determine world size
CALL MPI_COMM_RANK (MPI_COMM_WORLD,rnk_wrld,ierr) ! determine rank within world

PRINT "(A19,I3,A4,I4)"," hello from proc = ",rnk_wrld," of ",npe_wrld

CALL MPI_FINALIZE (ierr)                            ! terminate MPI environment

END PROGRAM hello_world
```

## Running the hello_world Program

```
PROGRAM hello_world
USE mpi

IMPLICIT NONE
INTEGER :: npe_wrld, &! number of processes within the world communicator
           rnk_wrld, &! rank of process within the world communicator
           ierr

CALL MPI_INIT (ierr)                                ! initialize MPI environment
CALL MPI_COMM_SIZE (MPI_COMM_WORLD,npe_wrld,ierr) ! determine world size
CALL MPI_COMM_RANK (MPI_COMM_WORLD,rnk_wrld,ierr) ! determine rank within world

PRINT "(A19,I3,A4,I4)"," hello from proc = ",rnk_wrld," of ",npe_wrld

CALL MPI_FINALIZE (ierr)                            ! terminate MPI environment

END PROGRAM hello_world
```

- To **run** the code use **`mpirun`**. The **`-np`** option determines the number of processes



```
bliss 39 > mpirun -np 8 hello_world
 hello from proc =   2 of    8
 hello from proc =   1 of    8
 hello from proc =   0 of    8
 hello from proc =   5 of    8
 hello from proc =   6 of    8
 hello from proc =   3 of    8
 hello from proc =   4 of    8
 hello from proc =   7 of    8
bliss 40 >
```

## The slightly modified "hello world" Program

- Important features of the slightly modified hello_world program
  1. Use the mpi commands **`MPI_GET_PROCESSOR_NAME`** to determine where a processes is actually running.
  2. Use the mpi commands **`MPI_WTICK`** and **`MPI_WTIME`** to time code
  3. Use the mpi commands **`MPI_BARRIER`** write output in order.

```
PROGRAM hello_world_2
USE mpi
IMPLICIT NONE

INTEGER :: npe_wrld, &! number of processes within the world communicator
           rnk_wrld, &! rank of process within the world communicator
           i,j,n,name_len,ierr
REAL (KIND=SELECTED_REAL_KIND (12)) :: wall_tick,time_start,time_end,x

CHARACTER (LEN=128) :: proc_name

CALL MPI_INIT (ierr)
CALL MPI_COMM_SIZE (MPI_COMM_WORLD,npe_wrld,ierr)
CALL MPI_COMM_RANK (MPI_COMM_WORLD,rnk_wrld,ierr)
```

## The slightly modified "hello world" Program

✦ Code (continued) for the slightly modified hello_world program

```
   CALL MPI_GET_PROCESSOR_NAME (proc_name,name_len,ierr)

   wall_tick  = MPI_WTICK () ! wall clock timer increment in seconds
   IF (rnk_wrld == 0)  PRINT "(A13,F12.8)"," wall_tick = ",wall_tick

! do some useless work
   time_start = MPI_WTIME () ! wall clock timer start
   x = 0.0_8
   DO j = 1,5000
      DO i = 1,5000
         x = x + SIN (x+FLOAT (rnk_wrld))
      ENDDO
   ENDDO
   time_end   = MPI_WTIME () ! wall clock timer stop

! write the results
   DO n = 0,npe_wrld-1
      IF (rnk_wrld == n) THEN
         PRINT "(A19,I3,A4,I4,A12,A16,A10,F8.5,A10,F12.8)", &
                             " hello from proc = ",rnk_wrld," of ",npe_wrld, &
                             " running on ",TRIM (proc_name), &
                             " time = ",time_end-time_start," answer = ",x
      ENDIF
      CALL MPI_BARRIER (MPI_COMM_WORLD,ierr)
   ENDDO

   CALL MPI_FINALIZE (ierr)

   END PROGRAM hello_world_2
```

## Collective Communication

- ✦ Transfer information for one process to many (scatter) or collect information from many processes to one (gather)

## Collective Communication: Scatter

- **MPI_BCAST** broadcasts a message from the process with rank **ROOT** to all processes of the communicator group **COMM**, itself included. It is called by all members of group using the same arguments. On return, the contents of root's send buffer has been copied to the receive buffer on all processes.

```
MPI_BCAST (buffer,data_count,data_type,root,comm)
```

## Collective Communication: Gather

- **`MPI_GATHER`**: Each process (root process included) sends the contents of its send buffer to the **`root`** process. The root process receives the messages into the receive buffer and stores them in rank order.

```
MPI_GATHER(send_buffer,send_count,send_type,
  recv_buffer,recv_count,recv_type,root,comm,ierr)
```

- **`MPI_REDUCE`**: Combines the elements in the send buffer of each process in the communicator group **`comm`**, using the operation **`op`**, and returns the combined value in the receive buffer of the process with rank **`root`**.

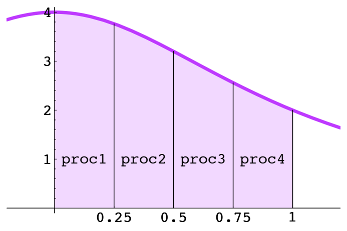
```
MPI_REDUCE(send_buffer,recv_buffer,send_count,
           send_type,op,root,comm,ierr)
```

where **op** can be several things including **MPI_MAX(MPI_MIN)** for maximum(minimum), **MPI_SUM** for summation.

## An example with **MPI_BCAST** and **MPI_REDUCE**

- Find an approximation for π using numerical integration

$$\int_0^1 \frac{4}{1 + x^2}\,dx = \pi$$



- The algorithm for the code:
  1. The root process will read the global number of intervals and broadcast the number to the other processes using **MPI_BCAST**.
  2. Each process will then determine its subinterval using its rank in the communicator and integrate  to find its subarea
  3. Using **MPI_REDUCE** with the option **MPI_SUM** the subareas are summed to find the total area

## Code the Pi example

```
   PROGRAM pi
   USE mpi
   IMPLICIT NONE

   INTEGER :: npe_wrld,rnk_wrld,n,i,ierr
   REAL (KIND=SELECTED_REAL_KIND (12)) :: &
      del_x,x_left,pi_piece,pi_approx,time_start,time_end,x
! setup MPI
   CALL MPI_INIT (ierr)
   CALL MPI_COMM_SIZE (MPI_COMM_WORLD,npe_wrld,ierr)
   CALL MPI_COMM_RANK (MPI_COMM_WORLD,rnk_wrld,ierr)
```

## Code the Pi example, continued

```
! read and broadcast total number of intervals
   IF (rnk_wrld==0) THEN
      PRINT *, 'Enter the total number of intervals '
      READ (*,*) n
   ENDIF
   CALL MPI_BCAST (n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)

   time_start = MPI_WTIME () ! wall clock timer start

! integrate subinterval
   del_x = 1.0_8/DBLE (n); x_left = DBLE (rnk_wrld)/DBLE (npe_wrld);
   pi_piece = 0.0_8
   DO i = 1,n/npe_wrld
      x = x_left + del_x*(DBLE(i)-0.5_8)
      pi_piece = pi_piece + del_x*(4.0_8/(1.0_8 + x**2))
   ENDDO

! gather the pieces of the pi
   CALL MPI_REDUCE(pi_piece,pi_approx,1,MPI_DOUBLE_PRECISION,MPI_SUM,0, &
                                                     MPI_COMM_WORLD,ierr)
   time_end = MPI_WTIME () ! wall clock timer stop
! print the approximate value
   IF (rnk_wrld==0) THEN
      PRINT "(A12,F22.20)","pi_approx = ",pi_approx
   ENDIF
   CALL MPI_BARRIER (MPI_COMM_WORLD,ierr)
   PRINT "(A12,F14.10)","time      = ",time_end-time_start

   CALL MPI_FINALIZE (ierr)

   END PROGRAM pi
```

## Point-to-point communication

- ✦ Here we send messages directly form one process to another.
- ✦ **MPI_SEND**:
  - This is a **blocking** send. Control does not return until the message data has been safely stored away so that the sender is free to overwrite the send buffer.
  - The syntax of the blocking send operation is given below:

```
MPI_SEND (BUFFER,DATA_COUNT,DATA_TYPE,DEST,TAG,
                    COMM,IERR)
```

where
`DEST` is the rank of destination (integer) within `COMM`
`TAG` is the message tag (integer)

## Point-to-point communication

✦ **MPI_RECV**:

- This is a blocking receive. Control returns only after the receive buffer contains the newly received message.
- The syntax of the blocking send operation is given below:

```
MPI_RECV (BUFFER,DATA_COUNT,DATA_TYPE,SOUR,TAG,COMM,
                    STATUS,IERR)
```

where

**SOUR** is the rank of source (integer) within **COMM**. The source can also be specified as **MPI_ANY_SOURCE**

**TAG** is the message tag (integer). The tag can also be specified as **MPI_ANY_TAG**

## Point-to-point communication. Matrix-vector multiplication

- ✦ This is a **“master-slave”** algorithm. One process (the master) is responsible for the coordinating the work of the others (the slaves).
- ✦ We wish to perform a matrix-vector multiply in parallel.

$$Ab=c$$

- ✦ The **master** algorithm for the code:
  1. The master will broadcast the vector b to all the slaves.
  2. The master will send one row of the matrix $A$ to each slave.
  3. The master then waits for the slave to perform the dot product and return the element of $c$. At this time the master sends that slave a new row of $A$. Continue until all rows are processed.
- ✦ The **slave** algorithm for the code:
  1. The slaves receive vector $b$ from master.
  2. Perform dot-products of $b$ and rows of $A$. Send result to master

## Point-to-point communication. Matrix-vector multiplication.

✦ The code is clearly partitioned into a **master** part and a **slave** part

```
PROGRAM mat_vec
USE mpi
IMPLICIT NONE

INTEGER,PARAMETER :: rows=100,cols=100
INTEGER :: npe_wrld,rnk_wrld,master,i,j,count_rows,sender,row_index,ierr
INTEGER :: status(MPI_STATUS_SIZE)
REAL (KIND=SELECTED_REAL_KIND (12)) :: &
   a(rows,cols),b(cols),c(rows),buffer(cols),ans,time_start,time_end

CALL MPI_INIT (ierr)
CALL MPI_COMM_SIZE (MPI_COMM_WORLD,npe_wrld,ierr)
CALL MPI_COMM_RANK (MPI_COMM_WORLD,rnk_wrld,ierr)

master = 0

IF (rnk_wrld==master) THEN ! THE MASTER DOES THIS BLOCK OF CODE
      .
       .
        .
ELSE ! THE SLAVES DO THIS BLOCK OF CODE
      .
       .
        .
ENDIF

CALL MPI_FINALIZE (ierr)

END PROGRAM mat_vec
```

## Point-to-point communication.  Matrix-vector multiplication.

✦ The **first** part of **master** code looks like this:

```
DO j = 1,cols ! make an arbitrary matrix a and vector b
   b(j) = 1.0_8
   DO i = 1,rows
      a(i,j) = DBLE (i+j)
   ENDDO
ENDDO
CALL MPI_BCAST (b,cols,MPI_DOUBLE_PRECISION,master,MPI_COMM_WORLD,ierr)

count_rows = 0
DO i = 1,npe_wrld-1
   DO j = 1,cols
      buffer(j) = a(i,j)
   ENDDO
   CALL MPI_SEND (buffer,cols,MPI_DOUBLE_PRECISION,i,i,MPI_COMM_WORLD,ierr)
   count_rows = count_rows+1
ENDDO
```

## Point-to-point communication.  Matrix-vector multiplication.

- The **second** part of **master** code looks like this:

```
DO i = 1,rows
   CALL MPI_RECV (ans,1,MPI_DOUBLE_PRECISION, &
                       MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,status,ierr)
   sender    = status(MPI_SOURCE)
   row_index = status(MPI_TAG) ! tag value in status is the row index
   c(row_index) = ans
   IF (count_rows < rows) THEN ! more work to be done.  send another row
      DO j = 1,cols
         buffer(j) = a(count_rows+1,j)
      ENDDO
      CALL MPI_SEND (buffer,cols,MPI_DOUBLE_PRECISION, &
                                     sender,count_rows+1,MPI_COMM_WORLD,ierr)
      count_rows = count_rows+1
   ELSE ! tell sender that there is no more work
      CALL MPI_SEND (MPI_BOTTOM,0,MPI_DOUBLE_PRECISION,sender,0,MPI_COMM_WORLD,ierr)
   ENDIF
ENDDO
```

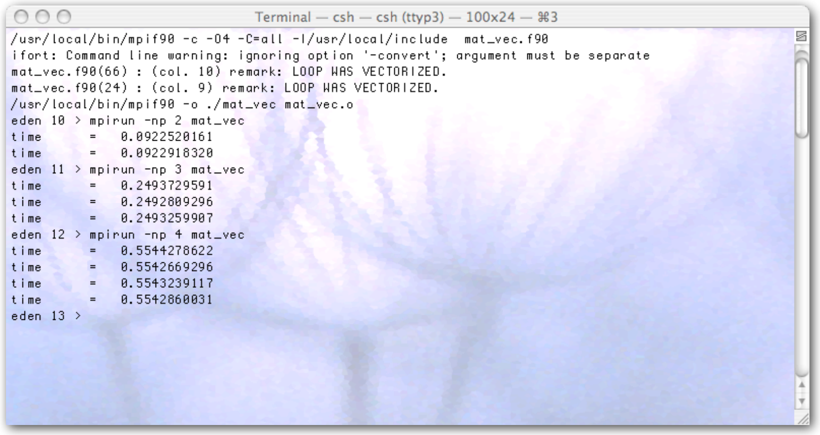
## Point-to-point communication. Matrix-vector multiplication.

✦ The **slave** code looks like this:

```
CALL MPI_BCAST (b,cols,MPI_DOUBLE_PRECISION,master,MPI_COMM_WORLD,ierr)
DO
  CALL MPI_RECV (buffer,cols,MPI_DOUBLE_PRECISION,master, &
                                     MPI_ANY_TAG,MPI_COMM_WORLD,status,ierr)
  IF (status(MPI_TAG)==0) EXIT ! there is no more work
  row_index = status(MPI_TAG)  ! tag value status is the row index
  ans = 0.0_8
   DO i = 1,cols
     ans = ans + buffer(i)*b(i)
   ENDDO
   CALL MPI_SEND (ans,1,MPI_DOUBLE_PRECISION, &
                                       master,row_index,MPI_COMM_WORLD,ierr)
ENDDO
```

## Point-to-point communication. Matrix-vector multiplication.

- ✦ Running the code
- ✦ Slower with more processes...



```
/usr/local/bin/mpif90 -c -O4 -C=all -I/usr/local/include  mat_vec.f90
ifort: Command line warning: ignoring option '-convert'; argument must be separate
mat_vec.f90(66) : (col. 10) remark: LOOP WAS VECTORIZED.
mat_vec.f90(24) : (col. 9) remark: LOOP WAS VECTORIZED.
/usr/local/bin/mpif90 -o ./mat_vec mat_vec.o
eden 10 > mpirun -np 2 mat_vec
time       =   0.0922520161
time       =   0.0922918320
eden 11 > mpirun -np 3 mat_vec
time       =   0.2493729591
time       =   0.2492809296
time       =   0.2493259907
eden 12 > mpirun -np 4 mat_vec
time       =   0.5544278622
time       =   0.5542669296
time       =   0.5543239117
time       =   0.5542860031
eden 13 >
```

## Nonblocking Send

- A **nonblocking** send call initiates the send operation, but does not complete it. The nonblocking send call will return before the message was copied out of the send buffer.
- A separate send complete call is needed to complete the communication, i.e., to verify that the data has been copied out of the send buffer.
- With suitable hardware, the transfer of data out of the sender memory may proceed concurrently with computations done by the sender after the send was initiated and before it completed.
- `MPI_ISEND` had the following syntax:

  ```
  MPI_ISEND (BUFFER,DATA_COUNT,DATA_TYPE,
             DEST,TAG,COMM,REQUEST)
  ```

  where the `REQUEST` argument determines if the operation has completed.

## Nonblocking Receive

- A **nonblocking** receive call initiates the receive operation, but does not complete it. The call will return before a message is stored into the receive buffer.
- A separate receive complete call is needed to complete the receive operation and verify that the data has been received into the receive buffer.
- With suitable hardware, the transfer of data into the receiver memory may proceed concurrently with computations done after the receive was initiated and before it completed.
- `MPI_IRECV` had the following syntax:

  ```
  MPI_IRECV (BUFFER,DATA_COUNT,DATA_TYPE,
             SOUR,TAG,COMM,REQUEST)
  ```

  where the `REQUEST` argument determines if the operation has completed.

## Completion of Nonblocking Send and Receive

- The call `MPI_WAITALL` blocks until all communication operations associated with active handles in the list are completed, and returns the status of all these operations.
- `MPI_WAITALL` had the following syntax:

  ```
  MPI_WAITALL(COUNT,ARRAY_OF_REQUESTS,
              ARRAY_OF_STATUSES,IERR)
  ```

  where the `REQUEST` argument determines if the operation has completed.

## Domain decomposition

- Here we will demonstrate the method of parallelization called domain decomposition. We will partition the physical domain into pieces and assign each piece to a process. Each process will communicate with it neighboring domain using message passing.
- We will numerically solve the Poisson equation.
- The continuous form of the problem:

$$\nabla^2\alpha = \beta(x,y) \quad \text{on the interior of the unit square } [0,1]\times[0,1]$$

$$\alpha(x,y) = \gamma(x,y) \quad \text{on the boundary}$$

- This simple PDE can be used as a template for more complicated problems. The communication patterns here are the same as more complex problems.

## Discrete Poisson problem: The grid

- The solution is approximated at discrete points. These points called a grid.
- The positions of the grid points $\left(x_i, y_j\right)$ are given by:

$$x_i = \frac{i}{n+1}, i = 0,\ldots,n+1 \qquad y_j = \frac{j}{n+1}, j = 0,\ldots,n+1$$

- The notation $\alpha_{i,j}$ refers to approximation of $\alpha$ at $\left(x_i, y_j\right)$
- The distance between grid points is given by

$$h = \frac{1}{n+1}$$

## Discrete Poisson problem: The discrete equation

- The continuous equation

$$\frac{\partial^2 \alpha}{\partial x^2} + \frac{\partial^2 \alpha}{\partial y^2} = \beta$$

- The discrete equation

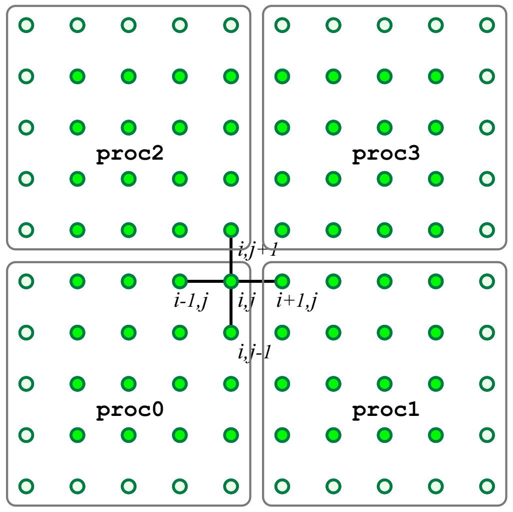
$$\frac{\alpha_{i-1,j} - 2\alpha_{i,j} + \alpha_{i+1,j}}{h^2} + \frac{\alpha_{i,j-1} - 2\alpha_{i,j} + \alpha_{i,j+1}}{h^2} = \beta_{i,j}$$

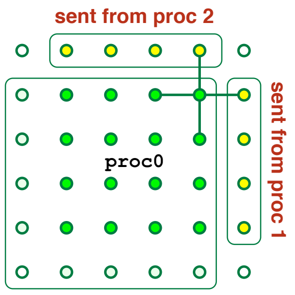
- Solve for $\alpha_{i,j}$ gives the Jacobi iteration

$$\alpha_{i,j}^{(k+1)} = \frac{1}{4}\left(\alpha_{i-1,j}^{(k)} + \alpha_{i+1,j}^{(k)} + \alpha_{i,j-1}^{(k)} + \alpha_{i,j+1}^{(k)} - h^2 \beta_{i,j}^{(k)}\right)$$

## Discrete Poisson problem: Domain decomposition

- ✦ For the case when $n = 8$, the 10 X 10 grid looks like this:
  - Solid green is a interior grid points
  - open circle is a boundary point
- ✦ Suppose we divide the grid to four processes.
- ✦ Then, for example, **`proc0`** is assigned an array 6X6 like this:





## Discrete Poisson problem: Algorithm

- The algorithm for the Jacobi iteration is given by:
  1. Communicate information to fill ghost cells
     a. Initiate nonblocking sends
     b. Initiate nonblocking receives
     c. Wait for message to be completed
  2. Perform one sweep of the Jacobi iteration
  3. GOTO 1.

## Where am I?  Who are my neighbors?

- It is useful to make a process map.  This can be used to determine position of the local process relative to other processes

```
INTEGER,PARAMETER :: &
   n = 256, &! global number of grid points along an edge
   iblk_max = 4, &! number domain decomposition blocks in the i-direction
   jblk_max = 4, &! number domain decomposition blocks in the j-direction
   i_max = n/iblk_max, &! local number of grid-points in the  i-direction
   j_max = n/jblk_max   ! local number of grid-points in the  j-direction

INTEGER :: i,j,ib,jb,proc,iblk,jblk,nghbr_count,req,edge,iter
INTEGER :: proc_map(0:iblk_max+1,0:jblk_max+1),nghbr_list(4)

      .
       .
        .

! set proc_map
   proc_map(:,:) = -1
   proc = 0
   DO jb = 1,jblk_max
      DO ib = 1,iblk_max
         proc_map(ib,jb) = proc; proc = proc + 1;
      ENDDO
   ENDDO

! determine position of the local process on the proc_map
   iblk = 1 +            MOD (rnk_wrld,iblk_max)
   jblk = 1 + (rnk_wrld-MOD (rnk_wrld,iblk_max))/iblk_max

! count the number of neighboring blocks
   nghbr_list(:) = (/ proc_map(iblk+1,jblk),proc_map(iblk,jblk+1), &
                      proc_map(iblk-1,jblk),proc_map(iblk,jblk-1) /)

   nghbr_count = COUNT (nghbr_list(:) /= -1)
```

## Initiate sends with MPI_ISEND

- Check each edge for a neighbor, load buffers and post sends

```
   TYPE buf_node
      REAL (KIND=SELECTED_REAL_KIND (12)),POINTER :: send(:),recv(:)
   END TYPE buf_node
   TYPE (buf_node) :: buf(4)

        .
         .
          .
! allocate memory for send and recv buffers
   ALLOCATE (buf(1)%send(j_max),buf(1)%recv(j_max)) ! east
   ALLOCATE (buf(2)%send(i_max),buf(2)%recv(i_max)) ! north
   ALLOCATE (buf(3)%send(j_max),buf(3)%recv(j_max)) ! west
   ALLOCATE (buf(4)%send(i_max),buf(4)%recv(i_max)) ! south
   ALLOCATE (send_req(nghbr_count))

      .
       .
        .
! post sends
   req = 0; send_req(:) = -999
   DO edge = 1,4
      IF (nghbr_list(edge) /= -1) THEN
         IF (edge == 1) buf(edge)%send(:) = alph(i_max,1:j_max) ! east
         IF (edge == 2) buf(edge)%send(:) = alph(1:i_max,j_max) ! north
         IF (edge == 3) buf(edge)%send(:) = alph(1,1:j_max)     ! west
         IF (edge == 4) buf(edge)%send(:) = alph(1:i_max,1)     ! south

         msg_tag = (npe_wrld+1)*rnk_wrld + nghbr_list(edge) + 1
         req = req + 1

         CALL MPI_ISEND (buf(edge)%send,SIZE (buf(edge)%send(:)), &
                            MPI_DOUBLE_PRECISION,nghbr_list(edge),msg_tag, &
                            MPI_COMM_WORLD,send_req(req),ierr)
      ENDIF
   ENDDO
```

## Initiate receives with MPI_IRECV

- Check each edge for a neighbor, clear buffers and post receives

```
! post receives
   req = 0; recv_req(:) = -999
   DO edge = 1,4
      IF (nghbr_list(edge) /= -1) THEN
         buf(edge)%recv(:) = 0.0

         msg_tag = (npe_wrld+1)*nghbr_list(edge) + rnk_wrld + 1
         req = req + 1

         CALL MPI_IRECV (buf(edge)%recv,SIZE (buf(edge)%recv(:)), &
                           MPI_DOUBLE_PRECISION,nghbr_list(edge),msg_tag, &
                           MPI_COMM_WORLD,recv_req(req),ierr)
      ENDIF
   ENDDO
```

## Wait for messages to be completed with MPI_WAITALL

- Check each edge for a neighbor, clear buffers and post receives

```
! allocate send_req, recv_req, send_status, recv_status
   ALLOCATE (send_req(nghbr_count))
   ALLOCATE (recv_req(nghbr_count))
   ALLOCATE (send_status(MPI_STATUS_SIZE,nghbr_count))
   ALLOCATE (recv_status(MPI_STATUS_SIZE,nghbr_count))

       .
        .
         .
!  wait for messages to complete
   send_status(:,:) = -999; recv_status(:,:) = -999;
   CALL MPI_WAITALL (nghbr_count,send_req,send_status,ierr)
   CALL MPI_WAITALL (nghbr_count,recv_req,recv_status,ierr)
```

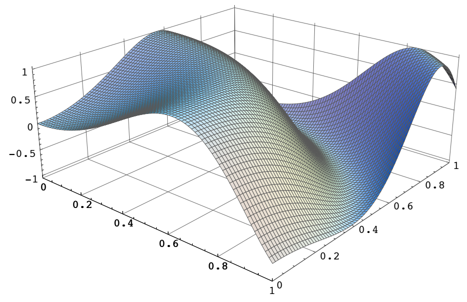
## Discrete Poisson problem: Set-up

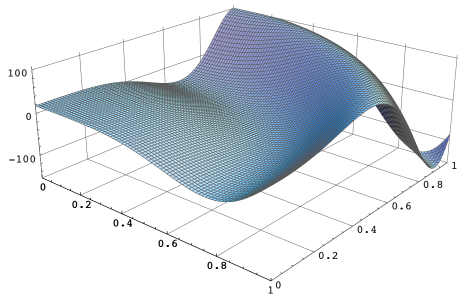
- Consider

$$\alpha(x,y) = \sin\left(4x^2 + 5y^2\right)$$

then

$$\beta(x,y) = 18\cos\left(4x^2 + 5y^2\right) - 64x^2\sin\left(4x^2 + 5y^2\right) - 100y^2\sin\left(4x^2 + 5y^2\right)$$


1
0.5
0
-0.5
-1
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1


100
0
-100
0
0.2
0.4
0.6
0.8
1
0
0.2
0.4
0.6
0.8
1