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:

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

$\alpha(x,y) = \gamma(x,y)$ 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 α 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 continous equation

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

- The discrete equation

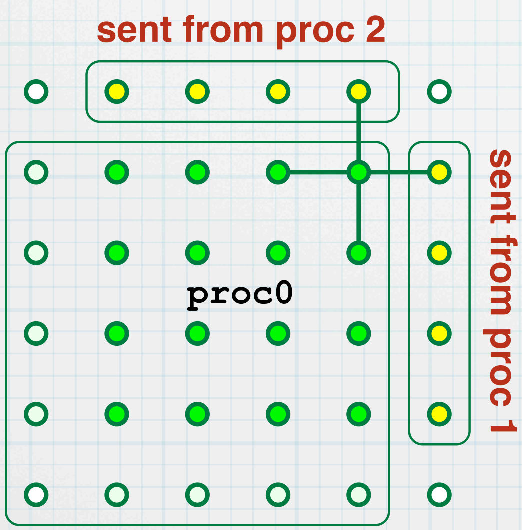
$$\frac{\alpha_{i-1,j} - 2\alpha_{i,j} + \alpha_{i+1,j}}{h^2} + \frac{\alpha_{i,j-1} - 2\alpha_{i,j} + \alpha_{i,j+1}}{h^2} = \beta_{i,j}$$

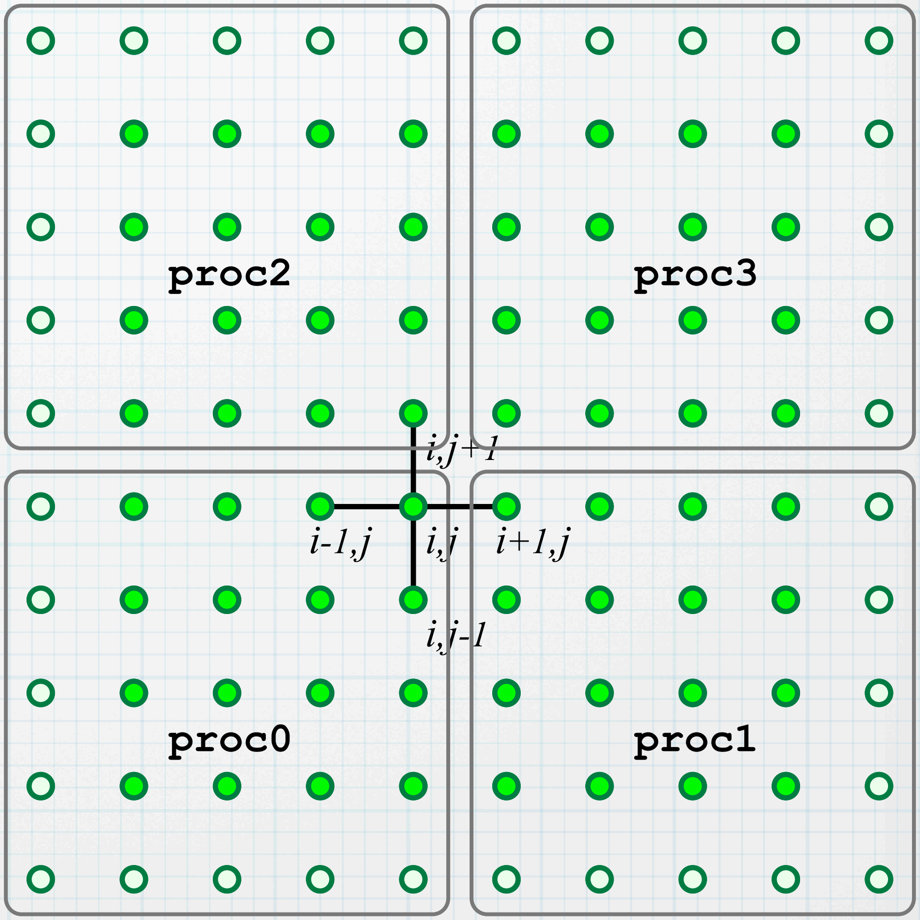
- Solve for gives the Jacobi iteration

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

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.

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.

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)
```

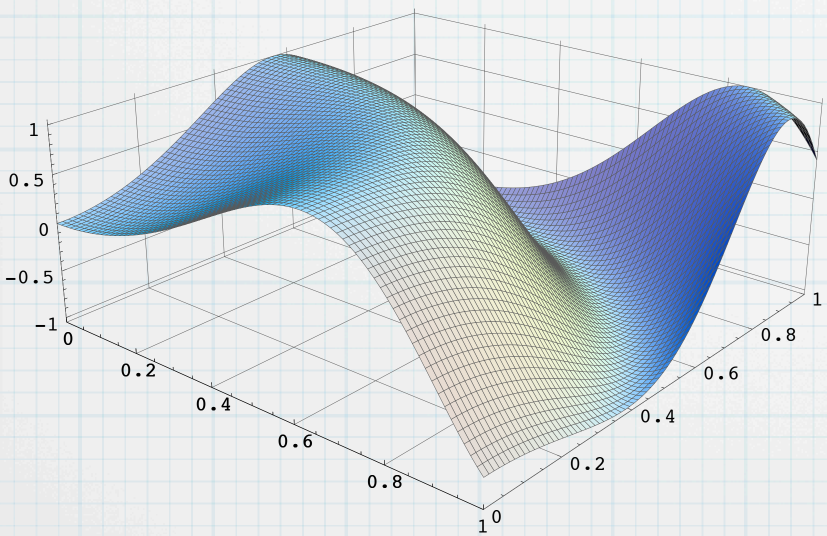
Discrete Poisson problem: Set-up

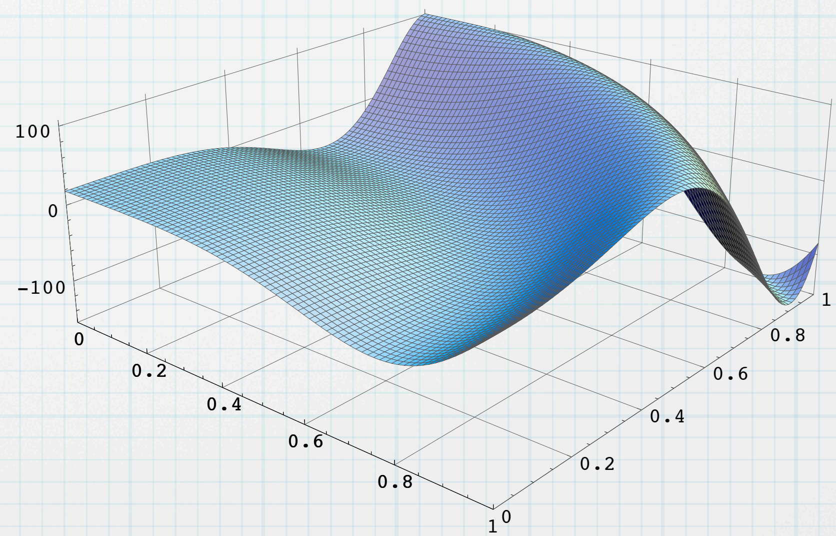
* Consider

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

then

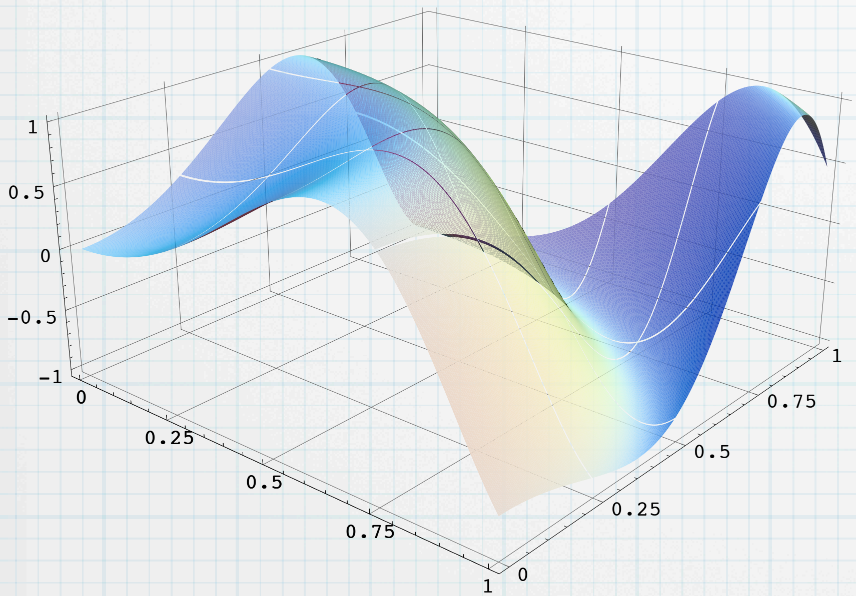
$$\beta(x,y) = 18\cos\left(4x^2 + 5y^2\right) - 64x^2\sin\left(4x^2 + 5y^2\right) - 100y^2\sin\left(4x^2 + 5y^2\right)$$

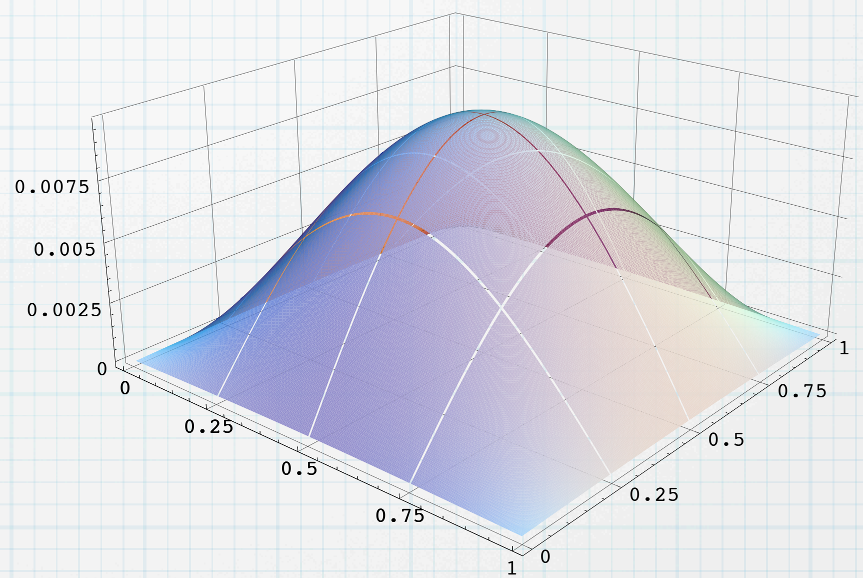




Discrete Poisson problem: Results

* The results look like this:





Next time...

1. Non-blocking sends and receives
    - Overlapping communications and computations
2. Topologies
3. MPI datatypes