Intermediate MPI:Domain Decomposition

A Tutorial with Exercises

by: Rosalinda de Fainchtein, Ph.D
CSC/NASA GSFC, Code 931

This tutorial was adapted from the class with exercises of the same name taught to NCCS computer users on May 2001. It constitutes a second course on MPI after "INTRODUCTION TO MPI: A Tutorial with Exercises". (You might want to start with the introductory tutorial if you are new to MPI programming, or if you would like a quick review of the proper use of basic MPI routines).

As you follow this tutorial, you will explore the basic steps and concepts involved in using domain decomposition to run your model or simulation in parallel. You will also get acquainted with and use MPI_CART routines, the set of MPI routines designed to manage domain decomposition in your MPI parallel program.

Exercises (and their solutions) are provided for you to practice each new concept. While you can review the pages of the tutorial relatively quickly, you will derive the most benefit from it if you solve the exercises on your own before reviewing their solutions.

Finally, although the examples and exercises are presented using Fortran programming, C versions are also available, courtesy of Dr. Hamid Oloso. They are stored both in jsimpson and an anonymous ftp site on UniTree under the sub-directory "C".

CONTENTS

Domain Decomposition

Question:
How do I code my model in parallel using MPI?

One Answer:
If your code lends itself to it,
divide the work among the processes through domain decomposition.

What is Domain Decomposition?

- A parallelization method
- Domain portions are assigned to individual processes

A Typical Example

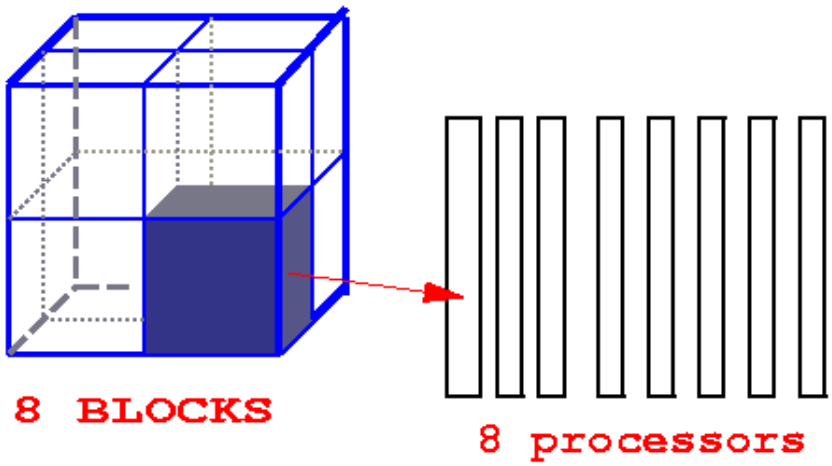
I have a finite difference model on a 64x64x64 cell domain.

It updates the arrays:

- **rho(1:64,1:64,1:64)**
- **vel(1:64,1:64,1:64)**

I want to run it on a parallel machine on 8 processors.

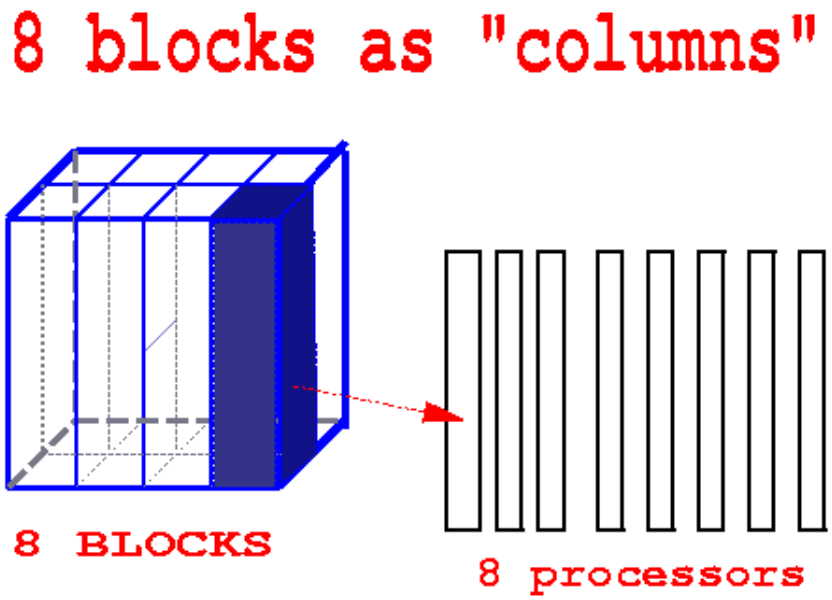
====> Assign a portion of the domain to each process.
.....(8 blocks, 1 block to each processor)



A Typical Example -- cont.

There are many ways to break up the 3D domain into blocks, not just "cubes". You should choose the way that best suits your code.

e.g. 8 columns of 3D data. One column to each processor*...

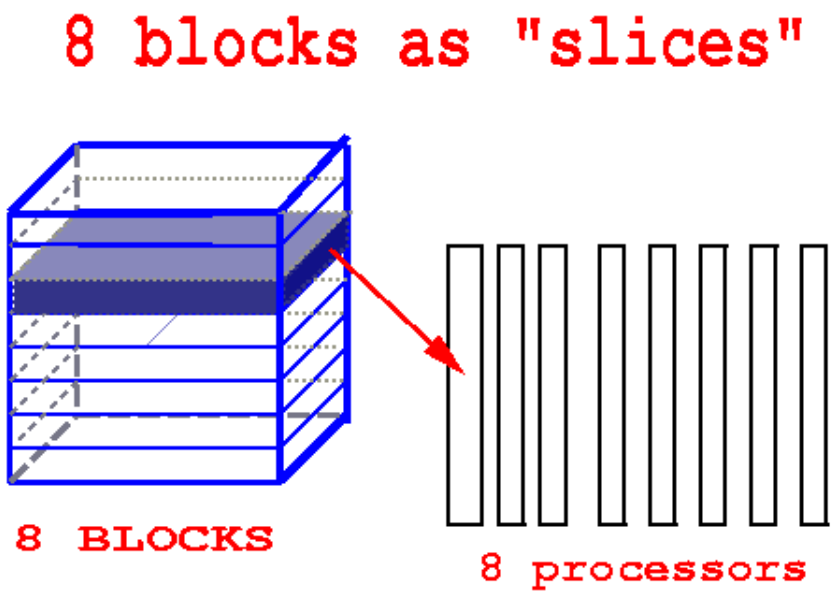


****This partitioning works well with many codes that have a lot of 1D computation along the z axis -- thus requiring no communication with neighboring columns.***

A Typical Example -- cont.

Another way to break up the 3D domain into blocks: "slices"

e.g. 8 slices of 3D data. One slice to each processor...



Domain Decomposition

Steps:

1. **Break up the domain into blocks.**

2. **Assign blocks to MPI-processes one-to-one.**

3. **Write or modify your code so it only updates a single block.**

4. **Provide a "map" of neighbors to each process.**

5. **Insert communication subroutine calls where needed.**

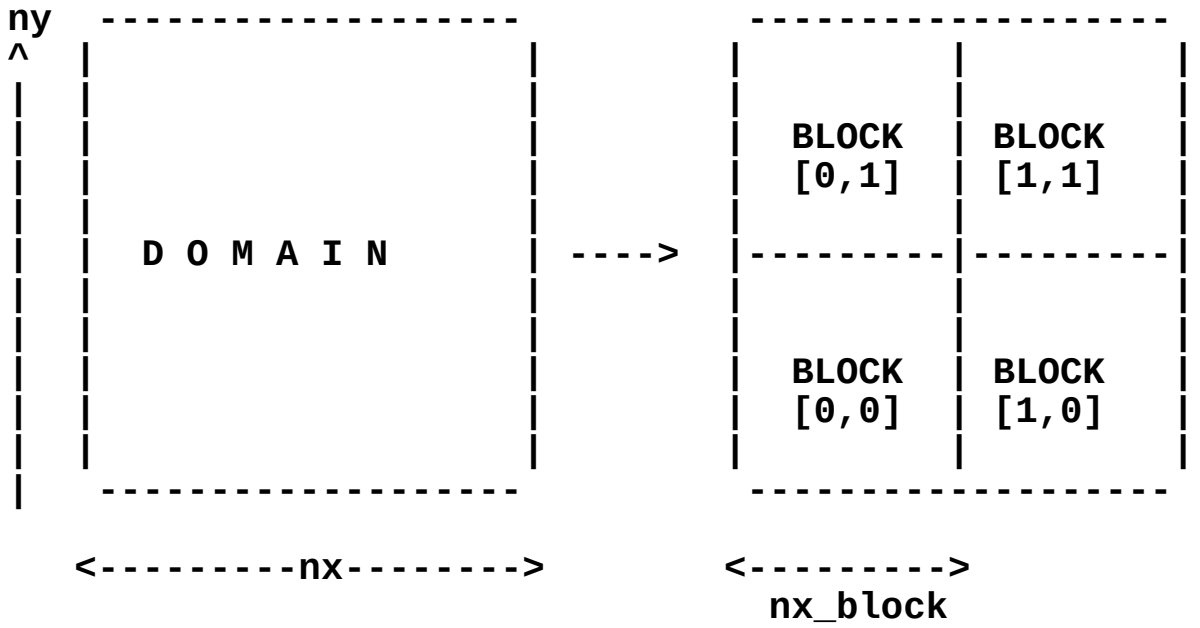
6. **Adjust the boundary conditions code.**

7. **Can your code use "guard cells"?**

Note: For ease of illustration, we will use mostly 1D and 2D examples. The concepts and techniques extend naturally to 3D, the most common type of parallel application.

First Step:Break up the Domain

e.g. 4 blocks (2D):

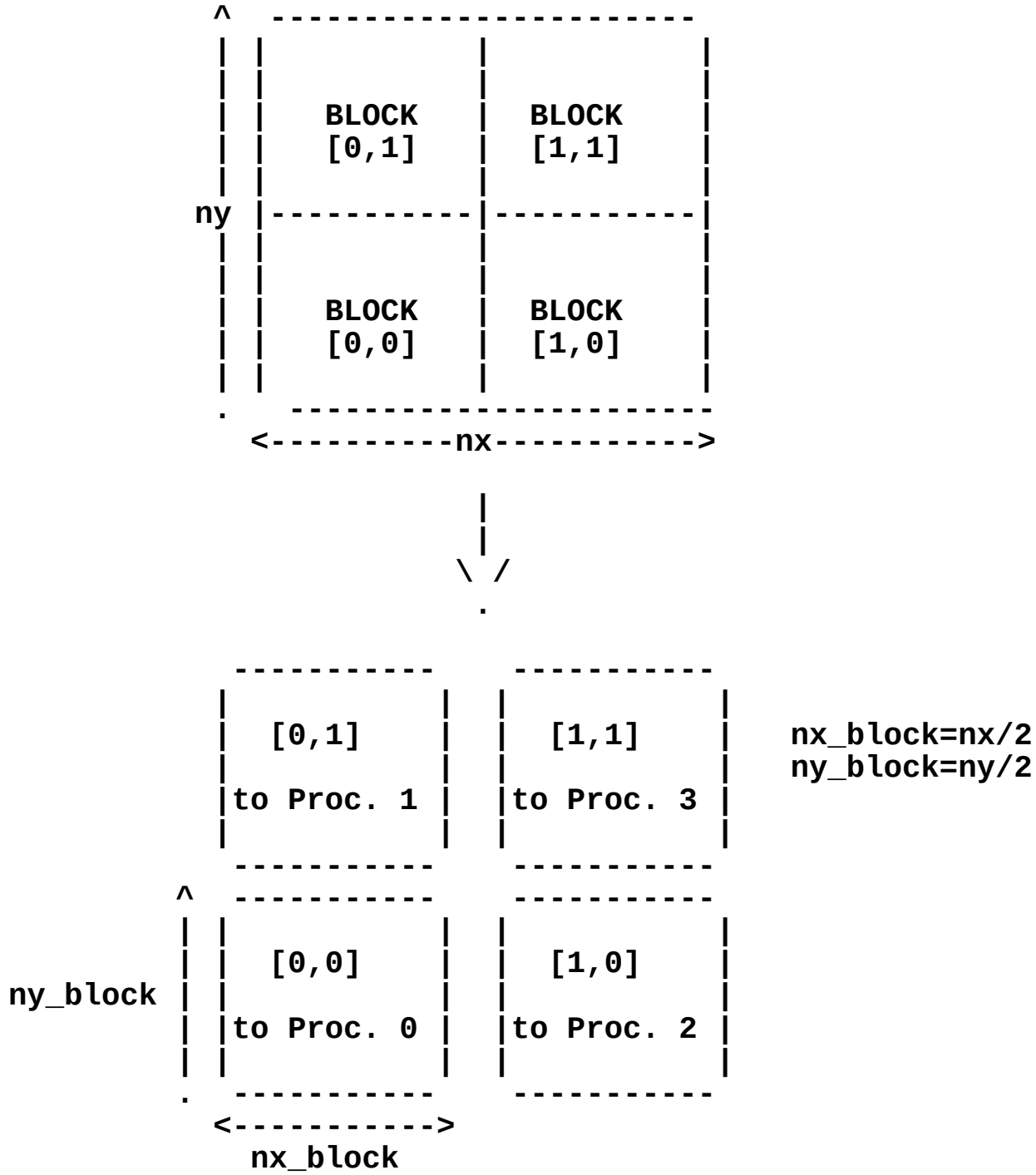


In the code:

rho(nx,ny) ---------> rho(nx_block,ny_block)

Second Step:Distribute Blocks

e.g. 4 processes: Assign one block to each of the 4 processes


BLOCK [0,1]
BLOCK [1,1]
BLOCK [0,0]
BLOCK [1,0]
ny
nx
[0,1] to Proc. 1
[1,1] to Proc. 3
[0,0] to Proc. 0
[1,0] to Proc. 2
nx_block=nx/2
ny_block=ny/2
ny_block
nx_block

Third Step:Code for 1 Block

REMEMBER: Each process will run the same code to update its block! Thus,

- **Adjust array dimensions to block size. e.g.:**

  rho(nx,ny) --> rho(nx_block,ny_block)

- **Code explicitly for specific blocks (i.e. processes) where necessary. e.g.**

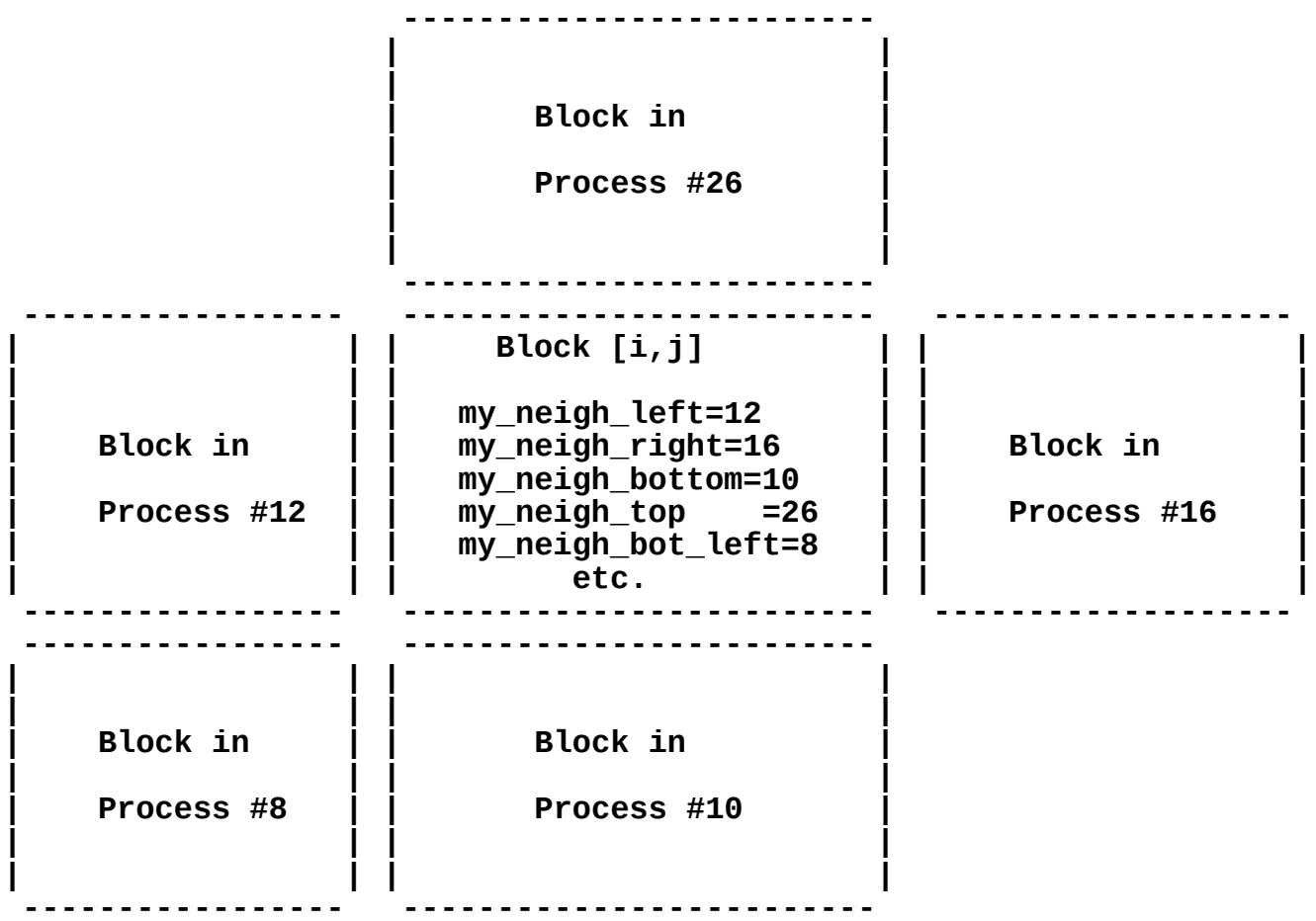
```
.
.
if (my_rank == 0) then
   {assign initial data for  block [0,0]..}
else if (my_rank == 1) then
   {assign initial data for  block [0,1]..}
else if (my_rank == 2) then
   .
   .
end if
```

Fourth Step:"Map" of Neighbors

I am block [i,j] and I need data from my neighbors.

WHO are my neighbors?

=======> "map" of neighboring processes to each process

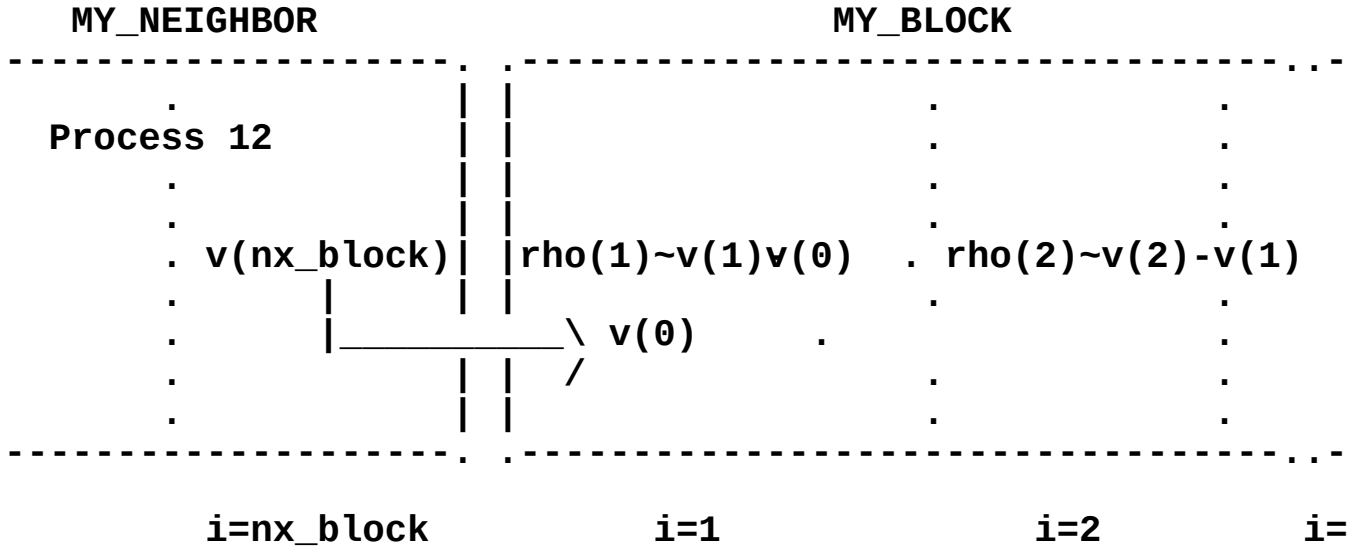


Fourth Step:"Map" of Neighbors - cont.

1D Example (internal block):

```
      .
      .
do i=1,nx_block
   rho(i)= ( v(i) - v(i-1) ) *dt/dx
end do
```

- Need "v(0)" to update rho(1).
- v(0) corresponds to v(nx_block) at my neighboring block.
- Fetch v(0) from my neighbor.
- ..but first find out which process holds my neighbor!



```
===> my_neigh_left=12
```


Fifth Step:Communication Calls

- **Examine the code/algorithm.**
- **Does this process need data from its neighbor(s)?**
- **Yes? ==> Insert communication calls where needed.**

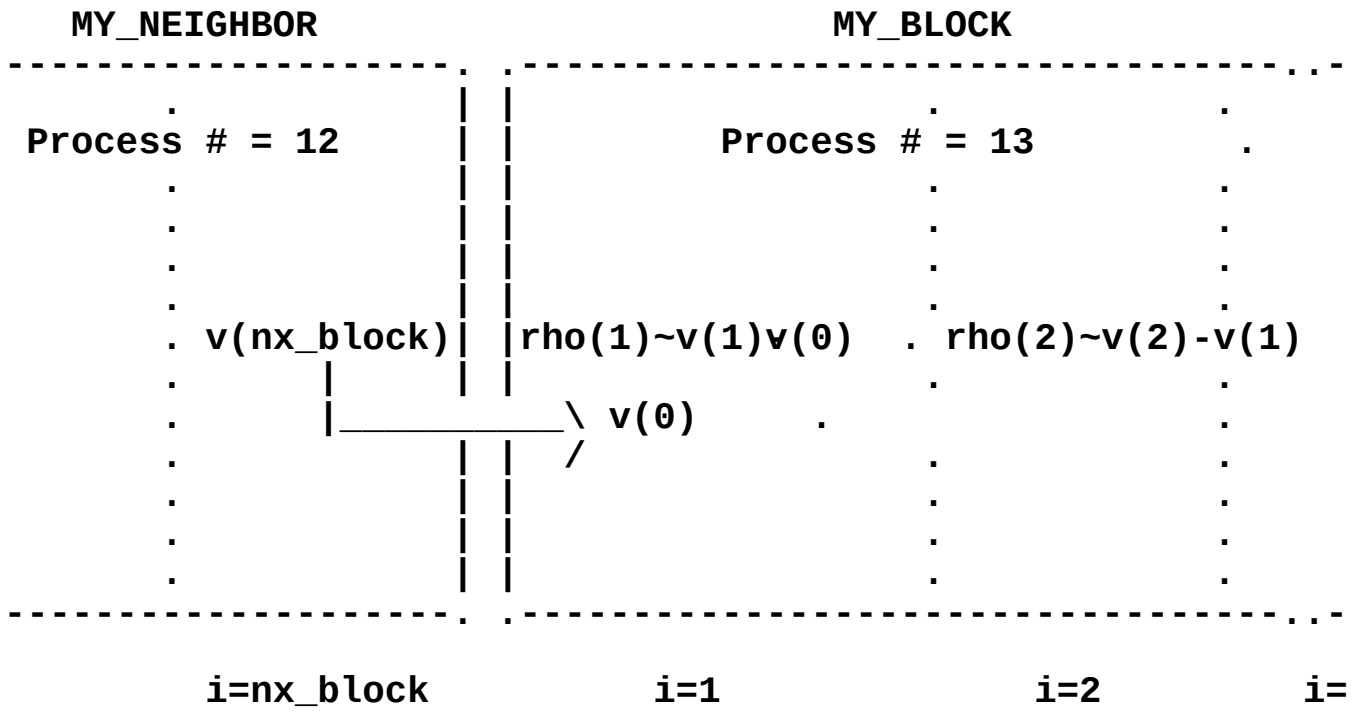
- **There are two types of MPI communications routines:**
  - **MPI collective routines (e.g. global average).**
  - **MPI Point to Point routines (see the example below).**

Fifth Step:Communication Calls

Example (internal block):

```
      .
      .
do i=1,nx_block
   rho(i)= ( v(i) - v(i-1) ) *dt/dx
end do
```

- **Need "v(0)" to update rho(1).**
- **v(0) corresponds to v(nx_block) at my neighboring block.**
- **Fetch v(0) from my neighbor.**



Pseudocode:

```
if (my_rank==12) call mpi_send( v(nx-block),....,13...
if (my_rank==13) then
  call mpi_recv( v(0),........,12,.....)
  do i=1,nx_block
   rho(i)=.....
```

```
      .
      .
   end do
end if
```


Pseudocode:

The very simplified advection code of our example may proceed as follows:

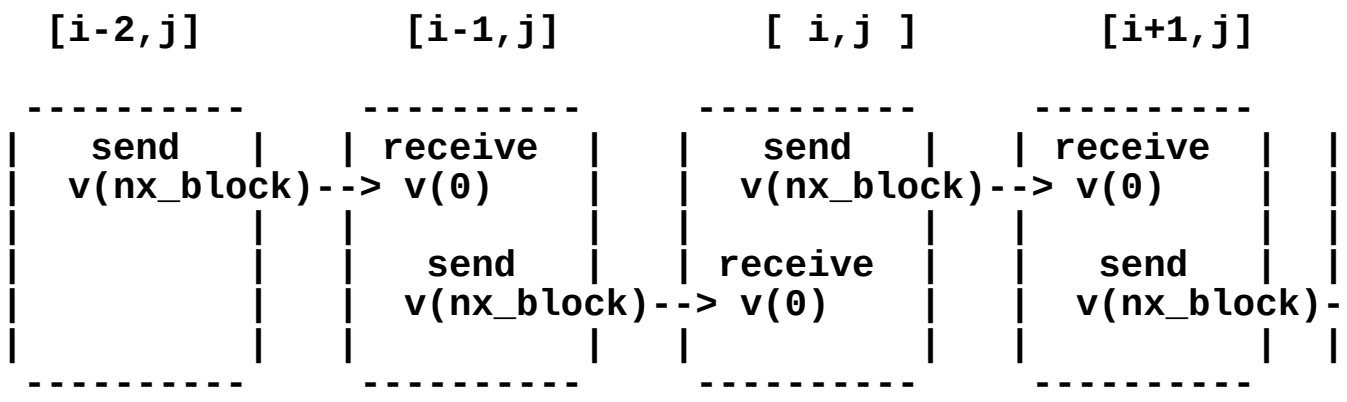
1. **Identify my neighbor's process (my_neighbor_left)**
2. **Post an mpi_receive of v(0) from my_neighbor_left:**
3. **My_neighbor_left posts a corresponding mpi_send to me of v(nx_block)**
4. **Update rho(1:nx_block)**

```
do i=1,nx_block
rho(i)=(v(i) - v(i-1) ) *dt/dx
end do
```

5. **Update v(1:nx_block)**
6. **Repeat for next time step -- Return to step 2.**

Coordinate Update of all Blocks:

All blocks send v(nx_block) to the right, and receive v(0) from the left.




```
call mpi_sendrecv( v(nx_block),1,...my_neighbor_right,...
                   v(0), 1,........,my_neighbor_left,...)
```

Note:

- **The choice of dimensioning v(0:nx_block) is arbitrary, a special case of:**
  v(start_block : start_block+nx_block)
- **with start_block=0**

....what if my block is at a physical boundary?

Domain Boundaries

What if my block is at a physical boundary?

In addition to "knowing" who its neighbors are, each process needs to "know" whether any of its boundaries is in fact a physical boundary.

Why?

1. **To avoid programming errors.**
2. **To apply boundary conditions correctly.**

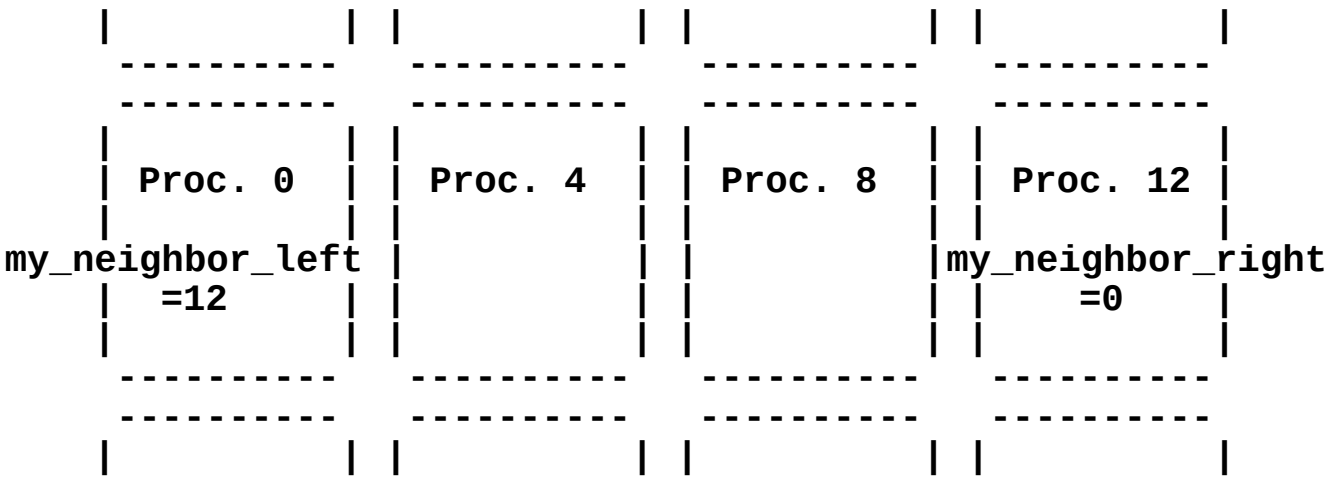
We will discuss the two cases:

- **Periodic Boundary Conditions**
- **Non-Periodic Boundary Conditions**

Domain Boundaries:

Periodic Boundary Conditions:

The "map" of neighbors should reflect the periodic boundary conditions.



Under periodic boundary conditions, the blocks at the ends are treated as any other "internal" block by just defining the correct map of neighbors:

- **For process 12, my_neighbor_right=0**
- **For process 0, my_neighbor_left=12**

Domain Boundaries:

Non-Periodic Boundaries

On a physical boundary you should:

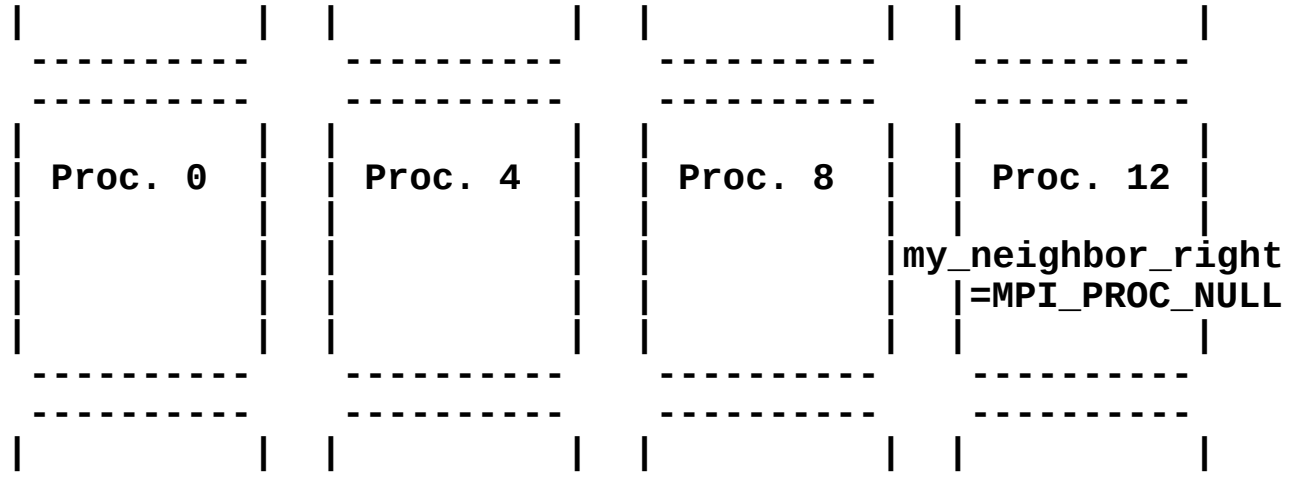
1. **Avoid sending or receiving data from a non-existent block.**
2. **Apply boundary conditions.**

A simple way to accomplish both tasks is by using the MPI constant handle:

MPI_PROC_NULL .

Let us illustrate with an example:

e.g. suppose process 12 corresponds to a right boundary block:



1. Avoid sending (or receiving) data from a bdry.

```
if (my_rank == 12) &
    my_neighbor_right = MPI_PROC_NULL
 .
 .
call mpi_send(......my_neighbor_right...)
                          !no data is sent by proc. 12*
```

2. Boundary conditions.... e.g. (Pseudocode)

```
if (my_neighbor_right == MPI_PROC_NULL) &
      [ APPLY APPROPRIATE BOUNDARY CONDITIONS ]
```

**** When mpi_send is called with MPI_PROC_NULL as a "destination" argument, the mpi_send routine does nothing.***

Guard Cells:

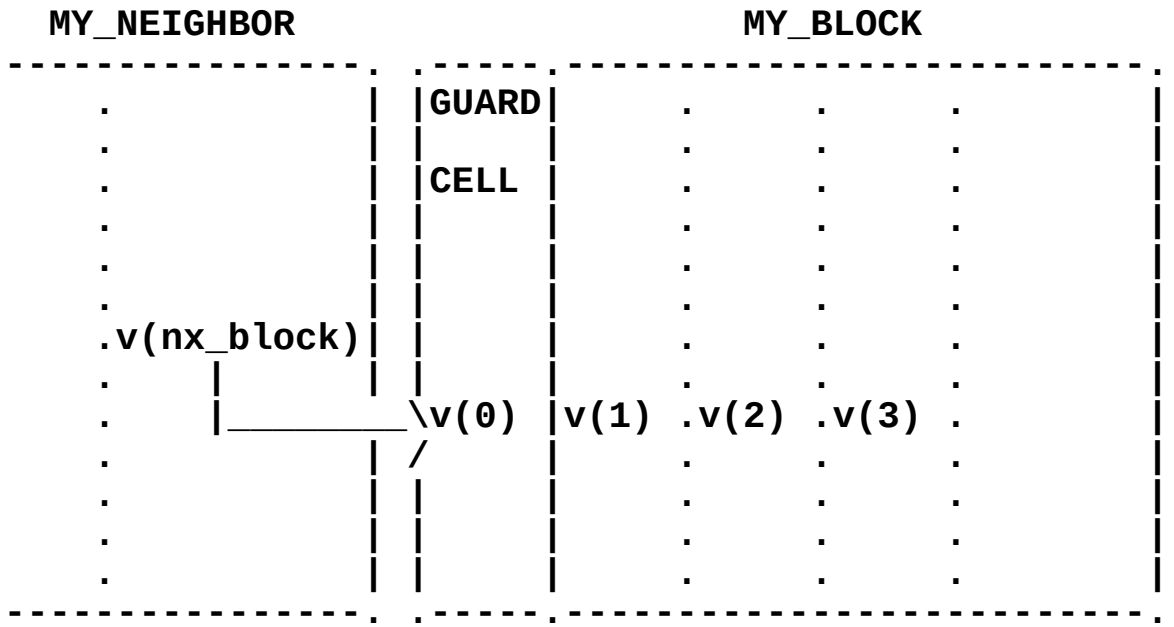
- **What are "Guard Cells"?**
  - **Guard cells are variables, arrays, or array sections that are defined with the sole purpose of storing data from a neighboring block.**
- **How are guard cells used?**
  - **They avoid repeated "fetching" of data from a**

neighboring block, when that data is needed more than once before it is updated by the neighbor.

- They allow for more "tidy" coding, as in our simple 1D advection example:

In our ID advection example:

```
do i=1,nx_block
rho(i)=(v(i) - v(i-1) ) *dt/dx
end do
```



v(0) is a guard cell.

MPI_CART Routines

In principle, you are ready to write your code in parallel using domain decomposition. You now know the basic issues to consider when writing a parallel code using domain decomposition. Together with basic MPI communication routines (e.g. mpi_send mpi_recv, etc) you have all the tools you need to write your parallel code.

If you ask:

Do I have to design a "map" of neighbors for each process/block?

The answer is: ***Not Necessarily.***

You can use MPI_CART routines to manage domain decomposition for you.

Minimal set of routines:

- **MPI_CART_CREATE**
- **MPI_CART_GET**
- **MPI_CART_RANK**

Very Useful MPI_CART routines:

- **MPI_CART_SHIFT**
- **MPI_CART_COORDS**

MPI_CART Routines - 2

How does it work?

1. **MPI_CART_CREATE creates a new communicator with Cartesian topology according to the programmer's specifications.**
    - **(All processes in an existing communicator make an identical call to MPI_CART_CREATE)**
    - **The new communicator (e.g. comm_cart) "stores" all the necessary information about the "grid" --- size, shape, map of processes to blocks,etc. -- in each process.**
2. **Each process can then access grid information about comm_cart by calling an assortment of local MPI_CART routines, e.g.**
    - **MPI_CART_GET**

- **MPI_CART_RANK**
- **MPI_CART_SHIFT**
- **MPI_CART_COORDS**

MPI_CART Routines - 3

In other words...

- **I start with an existing communicator (MPI_COMM_WORLD) with n processes.**
- **I call MPI_CART_CREATE and:**
  - **It logically arranges the n processes on an ndims-dimensional Cartesian grid of dimensions: dims(1:ndims)**
  - **Each process knows the position of all of the processes in this Cartesian grid.**
  - **To access this and other Cartesian grid information, a process needs to call the "inquiring" MPI_CART routines.**

MPI_CART Routines -- 4

1. **Information available through comm_cart on each process, includes:**

   - **Coordinate parameters* of each process.**
   - **Type of boundaries (periodic or non-periodic).**
   - **Topological dimension of the grid.**
   - **How many processes in each dimension.**
   - **etc.**

2. **Information can be accessed by calling:**

   - **MPI_CART_GET**
   - **MPI_CART_RANK**
   - **MPI_CART_SHIFT**
   - **MPI_CART_COORDS**

***Note that the coordinate parameters are the coordinates of each process on the grid of processes. Not to be confused with the *physical* coordinates of the cells or grid points in the model or simulation.**

MPI_CART_CREATE

MPI_CART_CREATE(comm_old,ndims,dims,periods,reo

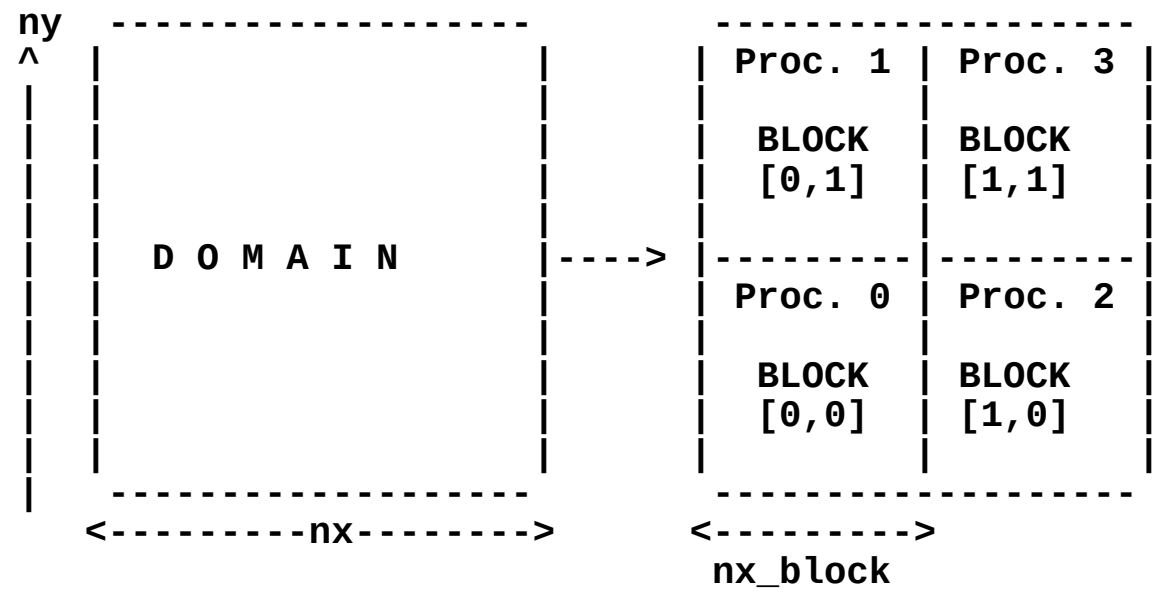
***comm_cart*,ierror)**

- **comm_old =input communicator (handle)**
- **ndims = # of dimensions of Cartesian grid (integer)**
- **dims(ndims) = # of processes in each dimension (int. array)**
- **periods = are boundaries periodic? (logical)**
- **reorder = ranks may be reordered (true) or not (false) (logical)**
- **comm_cart =communicator with new Cartesian topology(handle)**
- **ierror =return error code (integer)**

NOTE: The only return argument -- other than ierror -- is the handle to the new Cartesian communicator (comm_cart)!

MPI_CART_CREATE: An Example

In order to obtain the Cartesian topology for domain decomposition of our example (4 blocks):




```
.
.
.

ndims=2
dims(1:ndims)=2
periods(1:ndims)=.true.
```

```
call MPI_CART_CREATE(MPI_COMM_WORLD,ndi
                     dims,periods,.fals
                     COMM_CART,ierror)
```


Other MPI_CART Routines

MPI_CART_CREATE returns a handle (comm_cart). Each process can use comm_cart to inquire:

- **What are my coords in the grid? ==>**
  MPI_CART_GET(comm_cart....
  MPI_CART_COORDS(comm_cart....
- **What are the coords of any other process in the grid?**
  MPI_CART_COORDS(comm_cart....
- **Who are my neighbors? ==>**
  MPI_CART_SHIFT(comm_cart....
  MPI_CART_RANK(comm_cart....
- **Number of dimensions of the grid (1D,2D,3D..)? ==>**
  MPI_CARTDIM_GET(comm_cart....
- **What are the dimensions of the grid? ==>**
  MPI_CART_GET(comm_cart....

- **etc.**

Inquiring MPI_CART Routines: MPI_CART_COORDS

Q:Which block was assigned to process with rank=*some_rank*?
A:Call MPI_CART_COORDS and get the block's coordinate parameters.

MPI_CART_COORDS(comm_cart,rank,m coords,ierror)

- **comm_cart =Cartesian communicator (handle)**
- **rank = rank of process we are inquiring about(integer)**
- **maxdims = length of vector coords in the calling program**

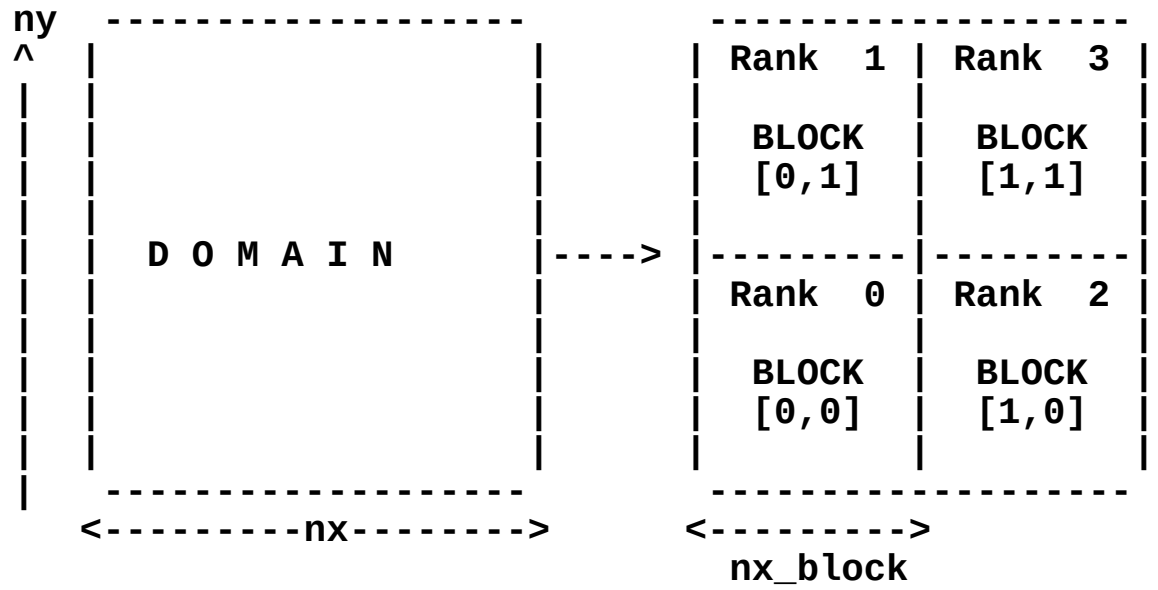
(integer)

- **coords(1:maxdims) = coordinate parameters of the the specified process (integer)**

- **ierror =return error code (integer)**

MPI_CART_COORDS: Illustration

In our example of a 2D nx x ny domain decomposed into 4 blocks:



call MPI_CART_COORDS(comm_cart,....rank=0,..) ==> coords(1:2) = 0,0

call MPI_CART_COORDS(comm_cart,....rank=1,..) ==> coords(1:2) = 0,1

call MPI_CART_COORDS(comm_cart,....rank=2,..) ==> coords(1:2) = 1,0

call MPI_CART_COORDS(comm_cart,....rank=3,..) ==> coords(1:2) = 1,1

MPI_CART_COORDS: Example 1

- **Create a 2x2 2D Cartesian communicator (COMM_CART)**
- **Print out the coordinate parameters of each process in COMM_CART.**

```
      .
      parameter(ndims=2)
      .
!--How many processes in the global group?
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )

!--Create a Cartesian topology

       dims(1:ndims)=2    ! 2x2 grid of sub-domains
      periods(1)=.true.  ! periodic bdry. cond. along x
      periods(2)=.false. ! non-periodic bdry. cond. along y

       call MPI_CART_CREATE(MPI_COMM_WORLD,ndims, &
                            dims,periods,.false., &
                            COMM_CART,ierror)

! --Find and print the coordinate parameters of each p
!   Cartesian topology

      call MPI_CART_COORDS(COMM_CART,my_rank,2,coords,i
      write(*,*)'The coords of process ',my_rank,' are
```

.
.

- **The code above is an excerpt from example1.f90.**
- **The full program can be copied from jsimpson at: /scr/mpi-class2/example1.f90.**
- **It can also be downloaded by anonymous ftp to UniTree.**

(In this example, each process finds and prints its own coordinate parameters. Keep in mind that any single process could have printed all the coordinate parameters -- see exercise 1)

MPI_CART_COORDS: Exercise 1

- **a. Copy and run example1.f90**
  You can find it in jsimpson at /scr/mpi-class2, or
  It can also be downloaded by anonymous ftp to UniTree.

  Note that each process "produces" its own coordinate parameters in the Cartesian group.

  Output (if running 4 MPI processes):

```
jsimpson% mpirun -np 4 example1
 The coords of process  1  are:  0,  1
```

```
The coords of process  0  are:  2*0
The coords of process  3  are:  2*1
The coords of process  2  are:  1,  0
```

Note that processes might be assigned different coordinate parameters on different computers.

- **b. Modify example1.f90 so that process 0 "finds" and prints the Cartesian parameters of each of the processes in the Cartesian grid. (Find a solution in exercise1.f90 at /scr/mpi-class2, or download it by anonymous ftp to UniTree.**

Defining and Assigning Data to Arrays after Domain Decomposition

- **Suppose you have a 2D domain of 4x4 cells.**
- **You want to initialize the array A(4,4) as some function of the physical coordinates:**

  x(4,4)
  y(4,4)

  e.g. A(i,j)=x(i,j)2 + y(i,j)**2**

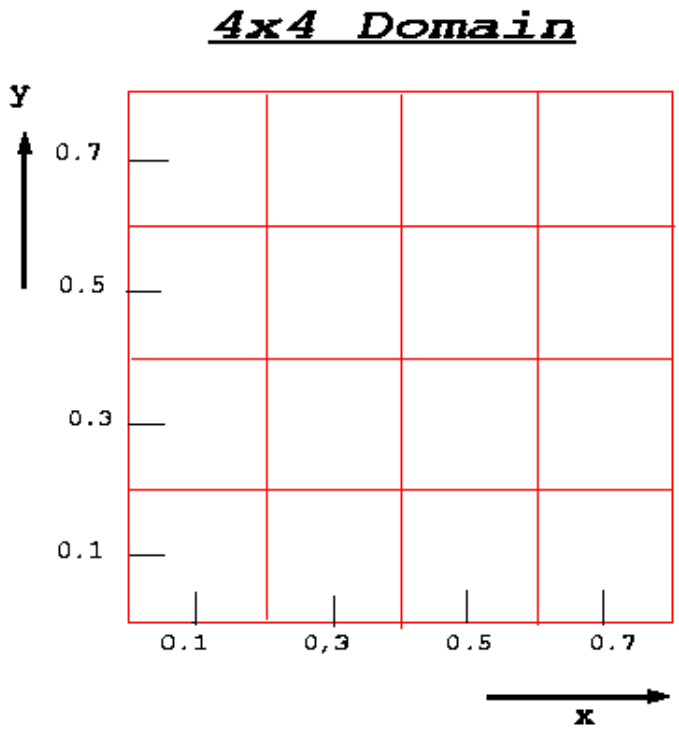
- **After domain decomposition (e.g. into 4 blocks), you will need to define local arrays on each block:**

x(2,2)
y(2,2)
A(2,2)

Defining and Assigning Data to Arrays after Domain Decomposition: Example 2

Given a 4x4 regular domain of cells, we will do domain decomposition into 4 processes.

Before decomposition:



- **The coordinate arrays are:**

  x(1:4,j)= (0.1, 0.3, 0.5, 0.7)

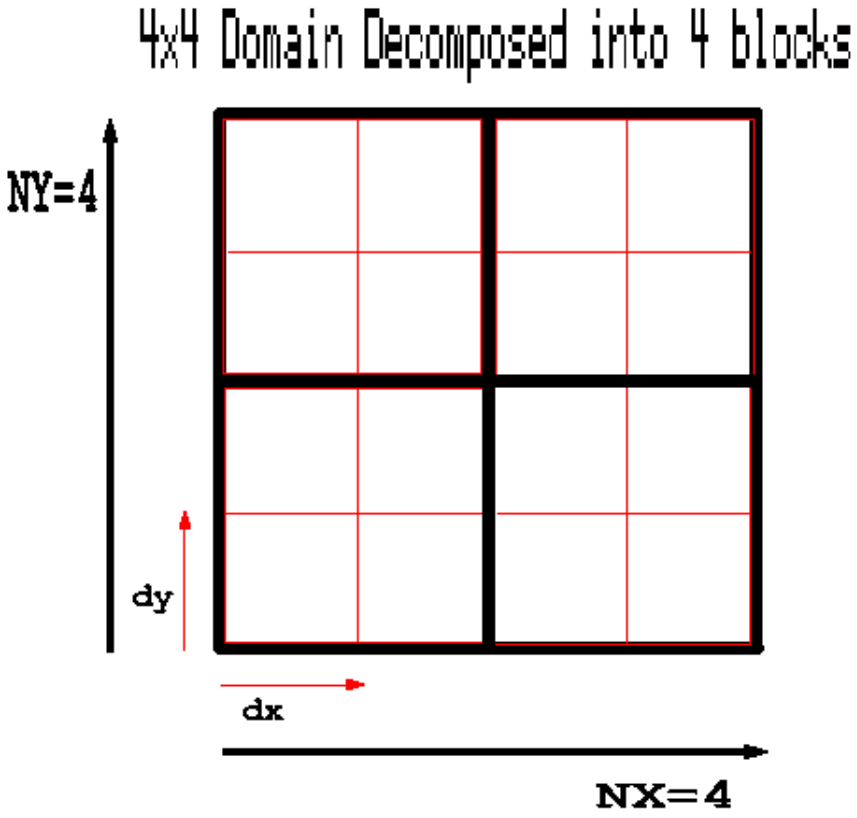
  y(i,1:4)= (0.1, 0.3, 0.5, 0.7)

- **The cell size is dx=dy=0.2**

Example2 - 1.cont.

Defining and Assigning Data to Arrays after Domain Decomposition: Example 2-cont.

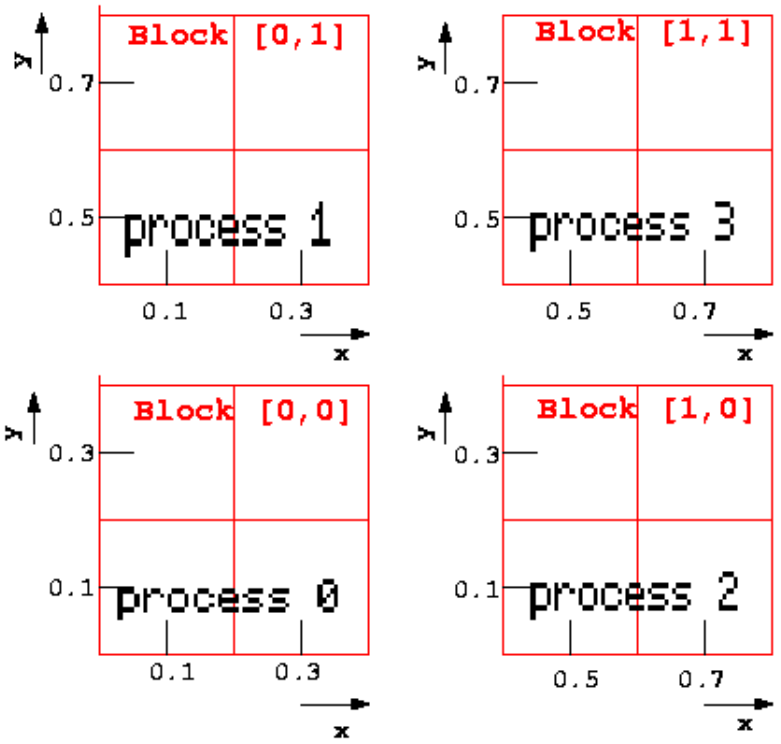
We choose to decompose the 4x4 domain into 4 blocks as shown in the figure below:



Defining and Assigning Data to Arrays after Domain Decomposition: Example 2 - cont.

After decomposition:

Decomposition into 4 Blocks



- **The coordinate arrays become x(1:2,1:2), y(1:2,1:2)**

- **x(1:2,1:2), y(1:2,1:2) are different on each block:**
  - **Block [0,0]:**
    x(1:2,j)=(0.1, 0.3)
    y(i,1:2)=(0.1, 0.3)
  - **Block [0,1]:**
    x(1:2,j)=(0.1, 0.3)
    y(i,1:2)=(0.5, 0.7)
  - **etc.**

Domain Decomposition: Exercise 2.

1. **Using example1.f90 as a template, have each process compute its own physical coordinate* arrays [x(1:2,1:2), y(1:2,1:2)].**
2. **Have each process print its process #, its Cartesian coordinate parameters, and its x and y arrays.**
3. **Verify your results. Do they correspond to the previous figure?**

HINT: Define the physical coordinates of the left-bottom corner of each process first.

```
! --The physical coordinates of the left-bottom corner
       x_corner= float( nx/dims(1) * coords(1)  ) * dx
       y_corner= float( ny/dims(2) * coords(2)  ) * dy
```

- **nx=number of cells along x on the whole domain**
- **ny=number of cells along y on the whole domain**
- **dims(1)=number of blocks along x**
- **dims(2)=number of blocks along y**
- **coords(1:2)=coord. parameters of this block.**

SUGGESTION: Use the following output statements:

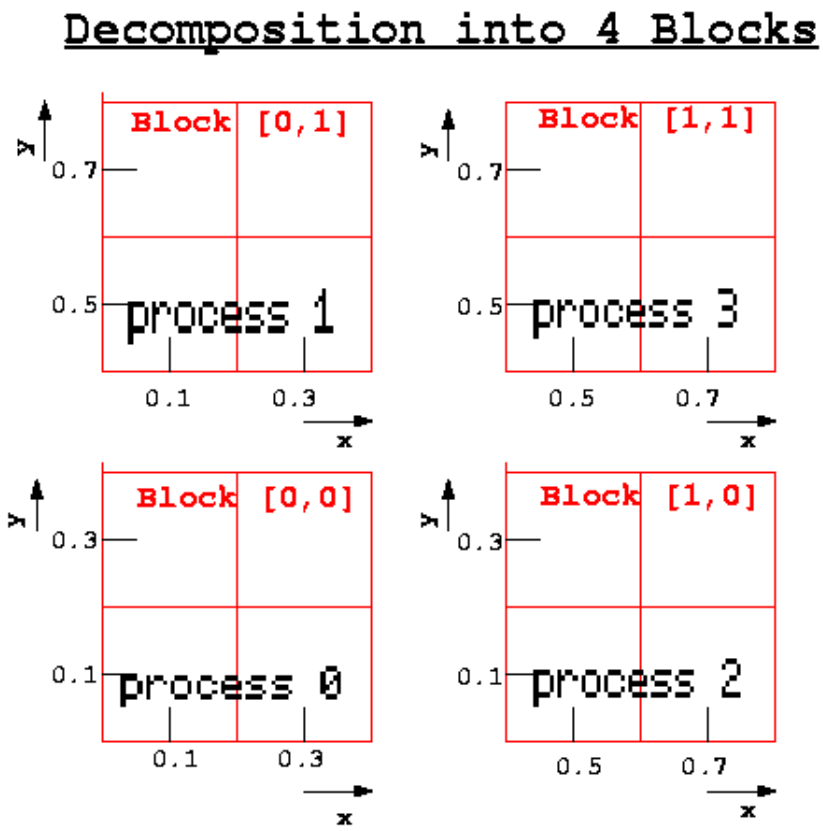
```
      write(*,"('my rank.=',i2,', my coords=[',2i2,']
                                        & x= ',4f4.1)"
     &         my_rank, coords, x

      write(*,"('my rank.=',i2,', my coords=[',2i2,']
                                        & y= ',4f4.1)"
     &         my_rank, coords, y
```

***Do not confuse the physical coordinate arrays (x,y), with the coordinate parameters of each block. The coordinate parameters are a pair of integers that "locate" the block within the Cartesian grid of blocks. The physical coordinate arrays (x,y) are the coordinates of the cells or points where physical data is defined.**

Output from Exercise 2.

```
mpirun -np 4 exercise2
my rank.= 0, my coords=[ 0 0] , x=  0.1 0.3 0.1 0.3
my rank.= 0, my coords=[ 0 0] , y=  0.1 0.1 0.3 0.3
my rank.= 2, my coords=[ 1 0] , x=  0.5 0.7 0.5 0.7
my rank.= 2, my coords=[ 1 0] , y=  0.1 0.1 0.3 0.3
my rank.= 1, my coords=[ 0 1] , x=  0.1 0.3 0.1 0.3
my rank.= 1, my coords=[ 0 1] , y=  0.5 0.5 0.7 0.7
my rank.= 3, my coords=[ 1 1] , x=  0.5 0.7 0.5 0.7
my rank.= 3, my coords=[ 1 1] , y=  0.5 0.5 0.7 0.7
```



Excerpt from exercise2.f90

Note that this "solution" is not unique!

```
.
.

nx=ny=4
dims(1)=dims(2)=2
```

```
          .

     call MPI_CART_COORDS(COMM_CART,my_rank,2,coords,&
                          & ierror)

!--The physical coordinates of my left-bottom corner:
       dx=0.2
       dy=0.2
       x_corner= float( nx/dims(1) * coords(1)  ) * dx
       y_corner= float( ny/dims(2) * coords(2)  ) * dy

!--Use my corner coordinates to compute the cell
!  coordinates in my block

       do i=1,2
        x(i,:)=x_corner+(float(i)-0.5)*dx
       end do

       do j=1,2
        y(:,j)=y_corner+(float(j)-0.5)*dy
       end do
          .
          .
          .
```

(See the full program at /scr/mpi-class2/exercise2.f90 in jsimpson, or use anonymous ftp to UniTree to retrieve exercise2.f90)

Exercises 2a-d

Below is a list of the next four exercises you will be working through. In all of them, cell sizes are

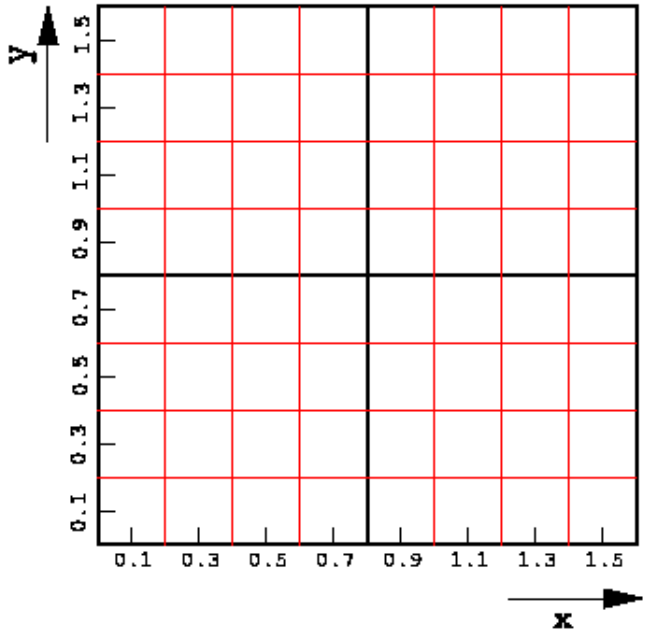
dx=dy(=dz)=0.2. To begin (and get some hints), please go to the next page .

- **2a. Modify exercise2.f90 so it decomposes an 8x8 grid into 4 blocks (*along x and y*).**
- **2b. Modify exercise2.f90 so it decomposes an 8x8 grid into 16 blocks (*along x and y*).**
- **2c. Modify exercise2.f90 to do 3D decomposition of a 4x4x4 grid into 8 blocks (*along x, y, and z*).**
- **2d. Modify exercise2.f90 to do 2D decomposition of a 4x4x4 grid into 4 blocks (*along x and y*).**

Exercise 2a

Exercise2a.f90

2a. Modify exercise2.f90 so it decomposes an 8x8 grid into 4 blocks (along x and y).



HINT: The grid of blocks is identical as that on exercise 2 (2x2). Only the data on each block will change.

- **Array dimension and values of x and y.**
- **Values of nx and ny**
- **do loop parameters**
- **(output format...)**

You can check your output against the next page --- note that the line order of output coming from different processes is bound to be different.

Output from Exercise 2a

Output from exercise2a.f90

```
mpirun -np 4 exercise2a
my rank.= 3, my coords=[ 1 1] , x=  0.9 1.1 1.3 1.5 0
 1.5 0.9 1.1 1.3 1.5 0.9 1.1 1.3 1.5
my rank.= 2, my coords=[ 1 0] , x=  0.9 1.1 1.3 1.5 0
 1.5 0.9 1.1 1.3 1.5 0.9 1.1 1.3 1.5
my rank.= 3, my coords=[ 1 1] , y=  0.9 0.9 0.9 0.9 1
 1.1 1.3 1.3 1.3 1.3 1.5 1.5 1.5 1.5
my rank.= 1, my coords=[ 0 1] , x=  0.1 0.3 0.5 0.7 0
 0.7 0.1 0.3 0.5 0.7 0.1 0.3 0.5 0.7
my rank.= 0, my coords=[ 0 0] , x=  0.1 0.3 0.5 0.7 0
 0.7 0.1 0.3 0.5 0.7 0.1 0.3 0.5 0.7
my rank.= 2, my coords=[ 1 0] , y=  0.1 0.1 0.1 0.1 0
 0.3 0.5 0.5 0.5 0.5 0.7 0.7 0.7 0.7
my rank.= 1, my coords=[ 0 1] , y=  0.9 0.9 0.9 0.9 1
 1.1 1.3 1.3 1.3 1.3 1.5 1.5 1.5 1.5
my rank.= 0, my coords=[ 0 0] , y=  0.1 0.1 0.1 0.1 0
 0.3 0.5 0.5 0.5 0.5 0.7 0.7 0.7 0.7
 STOP (PE 1)   executed at line 89 in Fortran routine
 STOP (PE 0)   executed at line 89 in Fortran routine
 STOP (PE 2)   executed at line 89 in Fortran routine
 STOP (PE 3)   executed at line 89 in Fortran routine
jsimpson%
```


Solution of exercise2a.f90

A solution code is available at jsimpson in:
/scr/mpi-class2/exercise2a.f90,
or by fetching exercise2a.f90 through anonymous ftp to UniTree

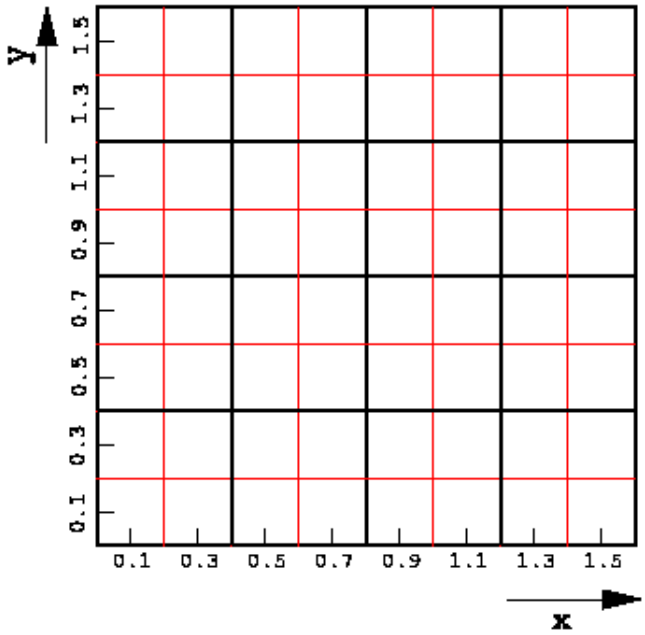
Only modifications pertaining to the data in each block are necessary:

```
jsimpson% diff exercise2.f90 exercise2a.f90
22c22
<       real x(2,2),y(2,2),dx,dy
---
```

```
>       real x(4,4),y(4,4),dx,dy
36,37c36,37
<          nx=4
<          ny=4
---
>          nx=8
>          ny=8
67c67
<         do i=1,2
---
>         do i=1,4
71c71
<         do j=1,2
---
>         do j=1,4
78c78
<                                 & x= ',4f4.1)" ) &
---
>                                 & x= ',16f4.1)" ) &
82c82
<                                 & y= ',4f4.1)" ) &
---
>                                 & y= ',16f4.1)" ) &
jsimpson%
```


Exercise2b

2b. Modify exercise2.f90 so it decomposes an 8x8 grid into 16 blocks (along x and y).



HINT: The grid of blocks is larger than that on exercise 2 (4x4). The size of the x and y arrays in each block are the same as in exercise2. The variables and arrays that change are:

- **Values of x and y. -- although you won't need to make modifications**
- **Values of nx and ny**
- **Values of array dims**

Output from exercise2b.f90

```
jsimpson% mpirun -np 16 exercise2b
```

```
my rank.=12, my coords=[ 3 0] , x=  1.3 1.5 1.3 1.5
my rank.=12, my coords=[ 3 0] , y=  0.1 0.1 0.3 0.3
my rank.= 5, my coords=[ 1 1] , x=  0.5 0.7 0.5 0.7
my rank.= 5, my coords=[ 1 1] , y=  0.5 0.5 0.7 0.7
my rank.= 8, my coords=[ 2 0] , x=  0.9 1.1 0.9 1.1
my rank.= 8, my coords=[ 2 0] , y=  0.1 0.1 0.3 0.3
my rank.= 6, my coords=[ 1 2] , x=  0.5 0.7 0.5 0.7
my rank.= 6, my coords=[ 1 2] , y=  0.9 0.9 1.1 1.1
my rank.= 3, my coords=[ 0 3] , x=  0.1 0.3 0.1 0.3
my rank.= 3, my coords=[ 0 3] , y=  1.3 1.3 1.5 1.5
my rank.=15, my coords=[ 3 3] , x=  1.3 1.5 1.3 1.5
my rank.=15, my coords=[ 3 3] , y=  1.3 1.3 1.5 1.5
my rank.=13, my coords=[ 3 1] , x=  1.3 1.5 1.3 1.5
my rank.=13, my coords=[ 3 1] , y=  0.5 0.5 0.7 0.7
my rank.=10, my coords=[ 2 2] , x=  0.9 1.1 0.9 1.1
my rank.=10, my coords=[ 2 2] , y=  0.9 0.9 1.1 1.1
my rank.= 0, my coords=[ 0 0] , x=  0.1 0.3 0.1 0.3
my rank.= 0, my coords=[ 0 0] , y=  0.1 0.1 0.3 0.3
my rank.=14, my coords=[ 3 2] , x=  1.3 1.5 1.3 1.5
my rank.=14, my coords=[ 3 2] , y=  0.9 0.9 1.1 1.1
my rank.=11, my coords=[ 2 3] , x=  0.9 1.1 0.9 1.1
my rank.= 2, my coords=[ 0 2] , x=  0.1 0.3 0.1 0.3
my rank.=11, my coords=[ 2 3] , y=  1.3 1.3 1.5 1.5
my rank.= 2, my coords=[ 0 2] , y=  0.9 0.9 1.1 1.1
my rank.= 4, my coords=[ 1 0] , x=  0.5 0.7 0.5 0.7
my rank.= 4, my coords=[ 1 0] , y=  0.1 0.1 0.3 0.3
my rank.= 1, my coords=[ 0 1] , x=  0.1 0.3 0.1 0.3
my rank.= 7, my coords=[ 1 3] , x=  0.5 0.7 0.5 0.7
my rank.= 9, my coords=[ 2 1] , x=  0.9 1.1 0.9 1.1
my rank.= 1, my coords=[ 0 1] , y=  0.5 0.5 0.7 0.7
my rank.= 7, my coords=[ 1 3] , y=  1.3 1.3 1.5 1.5
my rank.= 9, my coords=[ 2 1] , y=  0.5 0.5 0.7 0.7
.
.
```


Solution of exercise2b

A solution code is available at jsimpson in: /scr/mpi-class2/exercise2b.f90 or by fetching exercise2b.f90 through anonymous ftp to UniTree

Only parameters nx,ny, and dims need to be modified:

```
jsimpson% diff exercise2.f90 exercise2b.f90
<       integer, parameter :: dims_x=2, dims_y=2  !sar
---
>       integer, parameter :: dims_x=4, dims_y=4  !sar
36,37c36,37
<         nx=4
<         ny=4
---
>         nx=8
>         ny=8
39a40
>    !   dims(1:ndims)=(dims_x,dims_y)    ! 2x2 grid c
52a54
>
jsimpson%
```

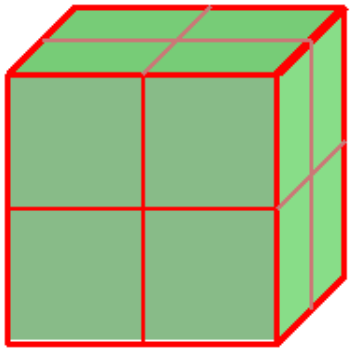

Exercise 2c

2c. Modify exercise2.f90 to do 3D decomposition of a 4x4x4 grid into 8 blocks (along x, y, and z).

HINTS:

- **Modify the calling parameters for MPI_CART_CREATE to correspond to a 3D decomposition.**
- **Make the x and y arrays 3 dimensionsal.**
- **Define a third array: z=third coordinate.**

- Compute z following the steps for x and y.
- You will have a 2x2x2 topology of blocks
- Each block (process) will hold 2x2x2 arrays.



Output from exercise2c.f90

A solution to exercise2c can be found at /scr/mpi-class2/exercise2c.f90
It can also be downloaded through anonymous ftp to UniTree

```
jsimpson% mpirun -np 8 exercise2c
my rank.= 5, my coords=[ 1 0 1] , x=  0.5 0.7 0.5 0.7
0.5 0.7
my rank.= 5, my coords=[ 1 0 1] , y=  0.1 0.1 0.3 0.3
0.3 0.3
my rank.= 5, my coords=[ 1 0 1] , z=  0.5 0.5 0.5 0.5
0.7 0.7
my rank.= 0, my coords=[ 0 0 0] , x=  0.1 0.3 0.1 0.3
```

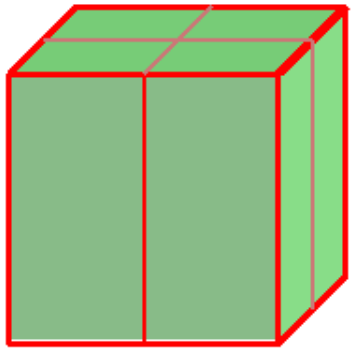
```
0.1 0.3
my rank.= 0, my coords=[ 0 0 0] , y=  0.1 0.1 0.3 0.3
0.3 0.3
my rank.= 0, my coords=[ 0 0 0] , z=  0.1 0.1 0.1 0.1
0.3 0.3
my rank.= 2, my coords=[ 0 1 0] , x=  0.1 0.3 0.1 0.3
0.1 0.3
my rank.= 6, my coords=[ 1 1 0] , x=  0.5 0.7 0.5 0.7
0.5 0.7
my rank.= 2, my coords=[ 0 1 0] , y=  0.5 0.5 0.7 0.7
0.7 0.7
my rank.= 6, my coords=[ 1 1 0] , y=  0.5 0.5 0.7 0.7
0.7 0.7
my rank.= 2, my coords=[ 0 1 0] , z=  0.1 0.1 0.1 0.1
0.3 0.3
my rank.= 3, my coords=[ 0 1 1] , x=  0.1 0.3 0.1 0.3
0.1 0.3
my rank.= 7, my coords=[ 1 1 1] , x=  0.5 0.7 0.5 0.7
0.5 0.7
my rank.= 6, my coords=[ 1 1 0] , z=  0.1 0.1 0.1 0.1
0.3 0.3
my rank.= 1, my coords=[ 0 0 1] , x=  0.1 0.3 0.1 0.3
0.1 0.3
my rank.= 3, my coords=[ 0 1 1] , y=  0.5 0.5 0.7 0.7
0.7 0.7
my rank.= 4, my coords=[ 1 0 0] , x=  0.5 0.7 0.5 0.7
0.5 0.7
my rank.= 7, my coords=[ 1 1 1] , y=  0.5 0.5 0.7 0.7
0.7 0.7
my rank.= 1, my coords=[ 0 0 1] , y=  0.1 0.1 0.3 0.3
0.3 0.3
my rank.= 3, my coords=[ 0 1 1] , z=  0.5 0.5 0.5 0.5
0.7 0.7
my rank.= 4, my coords=[ 1 0 0] , y=  0.1 0.1 0.3 0.3
0.3 0.3
my rank.= 7, my coords=[ 1 1 1] , z=  0.5 0.5 0.5 0.5
0.7 0.7
my rank.= 1, my coords=[ 0 0 1] , z=  0.5 0.5 0.5 0.5
0.7 0.7
my rank.= 4, my coords=[ 1 0 0] , z=  0.1 0.1 0.1 0.1
0.3 0.3
.
```

Exercise 2d

2d. Modify exercise2.f90 to do 2D decomposition of a 4x4x4 grid into 4 blocks (along x and y).

HINTS:

- **The calling parameters for MPI_CART_CREATE should be the same.**
- **Make the x and y arrays 3-dimensional -- each (2x2x4).**
- **Define a third array: z=third coordinate.**
- **Compute z following the steps for x and y.**
- **You will have a 2x2 topology of blocks**
- **Each block (process) will hold 2x2x4 arrays.**



Output from exercise2d.f90

A solution to exercise2d can be found at /scr/mpi-class2/exercise2d.f90
It can also be downloaded through anonymous ftp to UniTree.

```
jsimpson% !!
mpirun -np 4 exercise2d
my rank.= 2, my coords=[ 1 0] , x=  0.5 0.7 0.5 0.7 0
 0.7 0.5 0.7 0.5 0.7 0.5 0.7 0.5 0.7
my rank.= 2, my coords=[ 1 0] , y=  0.1 0.1 0.3 0.3 0
 0.3 0.1 0.1 0.3 0.3 0.1 0.1 0.3 0.3
my rank.= 2, my coords=[ 1 0] , z=  0.1 0.1 0.1 0.1 0
 0.3 0.5 0.5 0.5 0.5 0.7 0.7 0.7 0.7
my rank.= 3, my coords=[ 1 1] , x=  0.5 0.7 0.5 0.7 0
 0.7 0.5 0.7 0.5 0.7 0.5 0.7 0.5 0.7
my rank.= 1, my coords=[ 0 1] , x=  0.1 0.3 0.1 0.3 0
 0.3 0.1 0.3 0.1 0.3 0.1 0.3 0.1 0.3
my rank.= 3, my coords=[ 1 1] , y=  0.5 0.5 0.7 0.7 0
 0.7 0.5 0.5 0.7 0.7 0.5 0.5 0.7 0.7
my rank.= 0, my coords=[ 0 0] , x=  0.1 0.3 0.1 0.3 0
 0.3 0.1 0.3 0.1 0.3 0.1 0.3 0.1 0.3
my rank.= 1, my coords=[ 0 1] , y=  0.5 0.5 0.7 0.7 0
 0.7 0.5 0.5 0.7 0.7 0.5 0.5 0.7 0.7
my rank.= 3, my coords=[ 1 1] , z=  0.1 0.1 0.1 0.1 0
 0.3 0.5 0.5 0.5 0.5 0.7 0.7 0.7 0.7
my rank.= 0, my coords=[ 0 0] , y=  0.1 0.1 0.3 0.3 0
 0.3 0.1 0.1 0.3 0.3 0.1 0.1 0.3 0.3
my rank.= 1, my coords=[ 0 1] , z=  0.1 0.1 0.1 0.1 0
 0.3 0.5 0.5 0.5 0.5 0.7 0.7 0.7 0.7
my rank.= 0, my coords=[ 0 0] , z=  0.1 0.1 0.1 0.1 0
 0.3 0.5 0.5 0.5 0.5 0.7 0.7 0.7 0.7
 STOP (PE 3)   executed at line 100 in Fortran routine
 STOP (PE 0)   executed at line 100 in Fortran routine
 STOP (PE 1)   executed at line 100 in Fortran routine
 STOP (PE 2)   executed at line 100 in Fortran routine
```

Communication within the Cartesian "Grid"

Regroup:

- **You know how to decompose your data into a Cartesian grid of blocks.**
- **If you have an embarrassingly parallel program -- where processes contain all the data they need in local memory -- you are done.**
- **Otherwise, you will need to do inter-block communication.**

Most programs require inter-block communications.

When a process needs to send or receive data from a neighboring process, the first thing needed is to find that neighbor's rank...

Finding a Neighbor's Rank

- **I am a process of rank=my_rank**
- **I need to send/receive data to/from a neighbor.**
- **One way to do it:**

1. **Figure out my coordinates using:**
   - **MPI_CART_GET, or**
   - **MPI_CART_COORDS**
2. **Figure out the coordinates of the neighbor process e.g. If my coords are [i,j], my RHS neighbor is [i+1,j]**
3. **Use those coordinates to obtain the rank (e.g. n_rank) of the neighbor.**
   - **MPI_CART_RANK**

   (described on the next page)
4. **Send/receive the data to/from process with rank n_rank.**

Inquiring MPI_CART Routines:

MPI_CART_RANK

Q:What is the rank of the process with coordinates parameters=coords?
A:Call MPI_CART_RANK and get the process's rank.

MPI_CART_RANK(comm_cart,coords,ran

- **comm_cart = Cartesian communicator (handle)**
- **coords = coords of process we are inquiring about(integer array)**
- **rank = rank of the process with the specified coords (integer)**
- **ierror =return error code (integer)**

MPI_CART_RANK: Example 2

On a 2D Cartesian Grid of 4 Processes, each process finds the rank of its RHS neighbor (along x).

Excerpts from example2.f90 located at jsimpson in: /scr/mpi-class2/example2.f90

```
! --Given my rank (my_rank), find what my coords are.
       call MPI_CART_COORDS(COMM_CART,my_rank,ndims, &
                            &coords,ierror)
            .

!--Who is my RHS neighbor?
!  Well... Since my coords. are [coords(1),coords(2)]
!          my RHS neighbor's coords are:

       r_coords(1)=coords(1)+1
       r_coords(2)=coords(2)

!--Get its rank (r_rank):
       call MPI_CART_RANK(COMM_CART,r_coords, &
          & r_rank,ierror)
```


Output from Example 2

```
jsimpson% mpirun -np 4 example2
 my_rank= 2  my coordinates, coords, are  1,  0
 my_rank= 2  my RHS neighbor is:  0
 my_rank= 1  my coordinates, coords, are  0,  1
 my_rank= 1  my RHS neighbor is:  3
 my_rank= 3  my coordinates, coords, are  2*1
```

```
 my_rank= 0  my coordinates, coords, are  2*0
 my_rank= 3  my RHS neighbor is:  1
 my_rank= 0  my RHS neighbor is:  2
 STOP (PE 0)   executed at line 80 in Fortran routine
 STOP (PE 3)   executed at line 80 in Fortran routine
 STOP (PE 2)   executed at line 80 in Fortran routine
 STOP (PE 1)   executed at line 80 in Fortran routine
jsimpson%
```


MPI_CART_RANK: Exercise 3

Copy and modify example2.f90 to:

- **3a.Find the nearest neighbor along y+ of each process.**

  Verify your answers. Do they make sense? Note the value of periods in the code.

- **3b.Change the value of periods(2) to .false. and rerun exercise 3a. (The code will break -- do you know why?)**

- **3c.Find the nearest neighbor diagonally along the (x+,y+) direction (use periods=.true.).**
  Verify your answers.

Solution of exercise 3a

Modify the r_coords array:

- r_coords(1)=coords(1)
- r_coords(2)=coords(2)+1

Output from Exercise 3a

```
jsimpson% mpirun -np 4 exercise3a
 my_rank= 0  my coordinates, coords, are  2*0
 my_rank= 0  my TOP neighbor is:  1
 my_rank= 1  my coordinates, coords, are  0,  1
 my_rank= 1  my TOP neighbor is:  0
 my_rank= 3  my coordinates, coords, are  2*1
 my_rank= 3  my TOP neighbor is:  2
 my_rank= 2  my coordinates, coords, are  1,  0
 my_rank= 2  my TOP neighbor is:  3
 STOP (PE 2)   executed at line 80 in Fortran routine
 STOP (PE 3)   executed at line 80 in Fortran routine
 STOP (PE 0)   executed at line 80 in Fortran routine
 STOP (PE 1)   executed at line 80 in Fortran routine
jsimpson%
```


Solution to Exercise 3c

Modify the r_coords array:

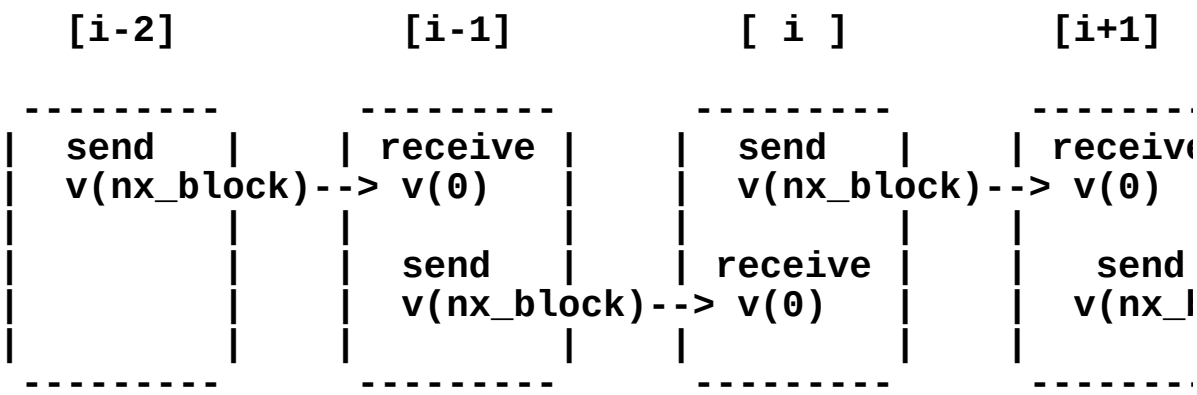
- r_coords(1)=coords(1)+1
- r_coords(2)=coords(2)+1

Output from Exercise 3c

```
jsimpson% f90 -o exercise3c exercise3c.f90
jsimpson% mpirun -np 4 exercise3c
 my_rank= 2  my coordinates, coords, are  1,  0
 my_rank= 2  my TOP-RIGHT neighbor is:  1
 my_rank= 0  my coordinates, coords, are  2*0
 my_rank= 0  my TOP-RIGHT neighbor is:  3
 my_rank= 1  my coordinates, coords, are  0,  1
 my_rank= 3  my coordinates, coords, are  2*1
 my_rank= 1  my TOP-RIGHT neighbor is:  2
 my_rank= 3  my TOP-RIGHT neighbor is:  0
 STOP (PE 1)   executed at line 80 in Fortran routine
 STOP (PE 2)   executed at line 80 in Fortran routine
 STOP (PE 0)   executed at line 80 in Fortran routine
 STOP (PE 3)   executed at line 80 in Fortran routine
jsimpson%
```


Send-Receive along a Coordinate Direction

On a 1D domain decomposition*, all processes (blocks) will be sending v(nx_block) to their neighbor to the right [x+], and receiving v(0) from their neighbor to the left [x-].




```
call mpi_sendrecv( v(nx_block),1,.....my_neighbor_r
                   v(0), 1,.........,my_neighbor_l
```

Each process needs the dest (*my_neighbor_right*) and source (*my_neighbor_left*) arguments for mpi_sendrecv. How can it get them?

- **Option 1: Use MPI_COORDS and MPI_RANK as explained before.**
  - **(somewhat cumbersome), or**

- **Option 2: Use MPI_SHIFT:**
  - **get source and dest arguments along a coordinate direction with a single subroutine call!**

***In a 2D decomposition, a similar sendrecv exchange would take place along the y coordinate as well [e.g. v(:,ny_block) --> v(:,0)].**

Inquiring MPI_CART Routines: MPI_CART_SHIFT

Send-Receive Communication along a Cartesian Coordinate
Use the MPI_SHIFT routine to obtain the source (e.g. my_neighbor_left), and the dest (e.g. my_neighbor_right) argument for the MPI_SENDRECV call.

MPI_CART_SHIFT(comm_cart,direction,d rank_source,rank_dest,ierror)

- **comm_cart =Cartesian communicator (handle)**
- **direction = coordinate dimension of shift (integer)**
  - ***direction=0 --> shift along "x"***
  - ***direction=1 --> shift along "y"***
  - ***etc.***
- **disp = displacement (>0: Up shift, <0: Down shift) (Integer)**
  - ***disp=1 --> dest=nearest neighbor to the "right" (source=nearest neighbor to the "left")***
  - ***disp=-1 --> dest=nearest neighbor to the "left" (source=nearest neighbor at "right")***
  - ***disp=2 --> dest=second nearest neighbor to the "right" .....***

- *etc.*

- **rank_source = rank of source process (integer)**
- **rank_dest = rank of destination process (integer)**
- **ierror =return error code (integer)**

MPI_CART_SHIFT: Example 4

On a 1D Cartesian grid of 4 blocks. Each block stores 2 cells, and 1 guard cell.
Update guard cell data (v0) for array vel(0:2)

Excerpt from example4.f90:

```
   parameter (ndims=1)
     .
   dims(1:ndims)=4
     .
!--Find the coordinates of this block
   call MPI_CART_COORDS(comm_cart,my_rank,
                        &ndims,coords,ierror)

!--Assign some values to vel(0:2)
       do i=0,2
        vel(i)=float(i+100*coords(1))
       end do

!--Find rank of neighbors along x
   direction=0              !(along x)
   disp=1                   !immediate neighbors
```

```
   call MPI_CART_SHIFT(comm_cart,direction, &
                       &disp,source,dest,ierror)

!--Send vel(2) to the block to my right and receive
!  v0 from my left
   sendtag=1
   recvtag=1
   call MPI_SENDRECV(vel(2),1,MPI_REAL, &
                     &dest,sendtag, &
                     &v0,1, &
                     &MPI_REAL,source,recvtag, &
                     &comm_cart,status,ierror)
```


Output from Example 4

Copy example4 from /scr/mpi_class2/example4.f90 in jsimpson and run it on 4 processes.
You can also get example4.f90 through anonymous ftp to UniTree.

The output should look like this:

```
jsimpson% f90 -o example4 example4.f90
jsimpson% mpirun -np 4 example4
 Process  3  of  4  is alive
 Process  2  of  4  is alive
 Process  0  of  4  is alive
 Process  1  of  4  is alive
 my_rank= 0  source= 3  dest= 1
 my_rank= 2  source= 1  dest= 3
 my_rank= 3  source= 2  dest= 0
 my_rank= 1  source= 0  dest= 2
```

```
 MPI_PROC_NULL= -1
 my coords are:  3    v0=  202.
 my coords are:  2    v0=  102.
 my coords are:  0    v0=  302.
 my coords are:  1    v0=  2.
 STOP (PE 1)   executed at line 84 in Fortran routine
 STOP (PE 0)   executed at line 84 in Fortran routine
 STOP (PE 2)   executed at line 84 in Fortran routine
 STOP (PE 3)   executed at line 84 in Fortran routine
jsimpson%
```


Exercise 4

Modify example4.f90 to update RHS guard cells.

Hints:

- Get "v3" from RHS neighbor.
- Modify "disp" in MPI_CART_SHIFT

Output from Exercise4:

```
jsimpson% f90 -o exercise4 exercise4.f90
jsimpson% mpirun -np 4 exercise4
 Process  3  of  4  is alive
 Process  0  of  4  is alive
 Process  2  of  4  is alive
 Process  1  of  4  is alive
 my_rank= 3  source= 0  dest= 2
 my_rank= 0  source= 1  dest= 3
```

```
 my_rank= 2  source= 3  dest= 1
 my_rank= 1  source= 2  dest= 0
 my coords are:  3    v3=  1.
 my coords are:  2    v3=  301.
 my coords are:  0    v3=  101.
 my coords are:  1    v3=  201.
 STOP (PE 1)   executed at line 81 in Fortran routine
 STOP (PE 2)   executed at line 81 in Fortran routine
 STOP (PE 3)   executed at line 81 in Fortran routine
 STOP (PE 0)   executed at line 81 in Fortran routine
jsimpson%
```


Solution to Exercise 4

Excerpt from exercise4.f90*:

```
.
.
      disp=-1          !MODIFIED!
      call MPI_CART_SHIFT(comm_cart,direction,disp,sou
                         & dest,ierror)
.
.
!--Send vel(1) to the block to my left  and receive v3
!  my right -->Modified
    sendtag=1
   recvtag=1
   call MPI_SENDRECV(vel(1),1,MPI_REAL,dest,sendtag, &
                  v3,1,MPI_REAL,source,recvtag, &
                  comm_cart,status,ierror)

!--Print out the updated v3
   write(*,*)'my coords are: ',coords,'   v3= ',v3
```

***The full code of exercise4.f90 can be found in jsimpson at /scr/mpi-class2/exercise4.f90, or through anonymous ftp to UniTree**

Exercise 4a

Make the following change in example4.f90:

```
periods=.false.
```

And run the program.

- **Compare the output you get for v0 to the periodic case of example4.**
- **Note how MPI_PROC_NULL is output by MPI_CART_SHIFT**
- **Note the effect of having**

  ```
  source=MPI_PROC_NULL
  ```

  and

  ```
  dest=MPI_PROC_NULL
  ```

  on the physical boundary blocks.

Output from Exercise 4a

Output....

```
    .
dest=MPI_PROC_NULL at proc  3
my coords are:  2    v0=  3.
source=MPI_PROC_NULL at proc  0
my coords are:  3    v0=  4.
my coords are:  0    v0=  0.E+0
my coords are:  1    v0=  2.
    .
```


"Echo" MPI_CART Routines

"Echo" routines inquire about the definition parameters of a Cartesian communicator, such as:

- **ndims: Is the topology 1D,2D, ...?,**
- **periods(ndims): periodic?**
- **dims(ndims): Dimensions along each coordinate direction**

There are two "echo" routines: MPI_CARTDIM_GET and MPI_CART_GET.

1.What is the Number of Dimensions in the Cartesian communicator comm_cart?

MPI_CARTDIM_GET(comm_cart, ndims,ierror)

- **comm_cart =communicator with Cartesian topology(handle)**
- **ndims = # of dimensions of Cartesian grid (integer)**
- **ierror =return error code (integer)**

MPI_CART_GET

2.Number of processes and periodicity along each Cartesian direction. Also, coordinate parameters of the calling process.

MPI_CART_GET(comm_cart,maxdims, dims,

periods,coords, ierror)

- **comm_cart =communicator with Cartesian topology(handle)**
- **maxdims = length of vectors dims, periods, coords (integer)**
- **dims = # of processes in each dimension (int. array)**
- **periods = are boundaries periodic? (logical array)**
- **coords =coords of calling process (int. array)**
- **ierror =return error code (integer)**

CONCLUSIONS

CONCLUSIONS

We covered the essentials of Domain Decomposition Using MPI_CART routines:

- **What is Domain Decomposition?**

- **Basics of a Domain Decomposition Code.**
- **Using MPI_CART Routines for Managing Domain Decomposition.**
- **Explained the various MPI_CART routines available.**
- **Examples and Exercises of Domain Decomposition into 1D, 2D, 3D Cartesian Topologies of "blocks".**
- **2D Decomposition of a 3D Domain.**

REFERENCES

REFERENCES

- **m.Snir et.al., "MPI -- The Complete Reference", 2nd Edition, 1998.**
- **Argonne Natl. Lab MPI web-site**
- **Compendium of MPI tutorials' web-sites at NCCS User Pages**