Distributed Computing and Introduction to High Performance Computing

Imad Kissami[1]
[1]Mohammed VI Polytechnic University, Benguerir, Morocco

Derived datatypes

Introduction

- In communications, exchanged data have different datatypes : MPI_INTEGER, MPI_REAL, MPI_COMPLEX, etc.
- We can create more complex data structures by using subroutines such as
- MPI_TYPE_CONTIGUOUS(), MPI_TYPE_VECTOR(), MPI_TYPE_INDEXED() or MPI_TYPE_CREATE_STRUCT()
- Derived datatypes allow exchanging non-contiguous or non-homogenous data in the memory and limiting the number of calls to communications subroutines

Derived datatypes

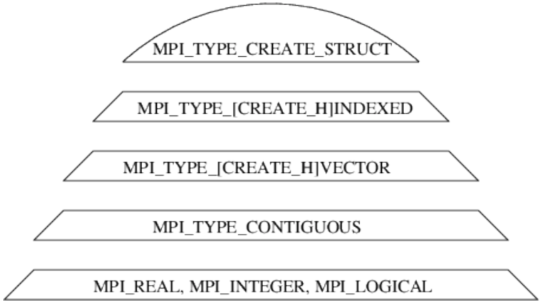

MPI_TYPE_CREATE_STRUCT
MPI_TYPE_[CREATE_H]INDEXED
MPI_TYPE_[CREATE_H]VECTOR
MPI_TYPE_CONTIGUOUS
MPI_REAL, MPI_INTEGER, MPI_LOGICAL

Figure: Hierarchy of the MPI constructors

Derived datatypes

Contiguous datatypes : MPI_TYPE_CONTIGUOUS()

- MPI_TYPE_CONTIGUOUS() creates a data structure from a homogenous set of existing datatypes contiguous in memory.

```
MPI_TYPE_CONTIGUOUS(count,old_type,new_type,code)

integer, intent(in) :: count, old_type
integer, intent(out) :: new_type,code
```

```
Datatype.Create_contiguous(self, int count)
```

Derived datatypes

MPI_TYPE_COMMIT and MPI_TYPE_FREE()

- Before using a new derived datatype, it is necessary to validate it with the MPI_TYPE_COMMIT() subroutine.

```fortran
MPI_TYPE_COMMIT(new_type,code)

integer, intent(inout) :: new_type
integer, intent(out) :: code
```

```python
new_type.Commit()
```

- The freeing of a derived datatype is made by using the MPI_TYPE_FREE() subroutine.

```fortran
MPI_TYPE_FREE(new_type,code)
integer, intent(inout) :: new_type
integer, intent(out) :: code
```

```python
new_type.Free()
```

Derived datatypes

Contiguous datatypes : MPI_TYPE_CONTIGUOUS()

- Example :

```python
count = 5
size = 10

type_ligne = MPI.DOUBLE.Create_contiguous(count)
type_ligne.Commit()

if rank==0:
    data = np.array([i for i in range(size)], dtype=np.float64)
    comm.Send([data,1,type_ligne],dest=1)
    print("Original data",data, rank)

elif rank == 1:
    data = -1*np.ones(size, dtype=np.float64)
    comm.Recv([data,1,type_ligne],source=0)
    print("Received data",data, rank)

type_ligne.Free()
```

```
mpirun -n 2 python3 create_contiguous.py
Original data [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.] 0
Received data [ 0.  1.  2.  3.  4. -1. -1. -1. -1. -1.] 1
```

Derived datatypes

Constant stride : MPI_TYPE_VECTOR()

- MPI_TYPE_VECTOR() creates a data structure from a homogenous set of existing datatypes separated by a constant stride in memory. The stride is given in number of elements.

```fortran
MPI_TYPE_VECTOR(count,block_length,stride,old_type,new_type,code)

integer, intent(in) :: count,block_length
integer, intent(in) :: stride
integer, intent(in) :: old_type

integer, intent(out) :: new_type,code
```

```python
Datatype.Create_vector(self, int count, int blocklength, int stride)
```

Derived datatypes

Constant stride : MPI_TYPE_VECTOR()

- Example :

```python
stride = 2
count = 5
blocklen = 1
size = 10

type_colum = MPI.DOUBLE.Create_vector(count,blocklen,stride)
type_colum.Commit()

if rank==0:
    data = np.array([i for i in range(size)], dtype=np.float64)
    comm.Send([data,1,type_colum],dest=1)
    print("Original data",data, rank)
elif rank==1:
    data = -1*np.ones(size, dtype=np.float64)
    comm.Recv([data,1,type_colum],source=0)
    print("Received data",data, rank)

type_colum.Free()
```

```
mpirun -n 2 python3 create_vector.py
Original data [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.] 0
Received data [ 0. -1.  2. -1.  4. -1.  6. -1.  8. -1.] 1
```

Derived datatypes

Constant stride : MPI_TYPE_CREATE_HVECTOR()

- MPI_TYPE_CREATE_HVECTOR() creates a data structure from a homogenous set of existing datatype separated by a constant stride in memory. The stride is given in bytes.
- This call is useful when the old type is no longer a base datatype (MPI_INTEGER, MPI_REAL,...) but a more complex datatype constructed by using MPI subroutines, because in this case the stride can no longer be given in number of elements.

```
MPI_TYPE_CREATE_HVECTOR(count,block_length,stride,old_type,↩
     new_type,code)

integer, intent(in) :: count,block_length
integer(kind=MPI_ADDRESS_KIND), intent(in) :: stride
integer, intent(in) :: old_type

integer, intent(out) :: new_type, code
```

```
Datatype.Create_hvector(self, int count, int blocklength, Aint stride)
```

Derived datatypes

Homogenous datatypes of variable strides

- MPI_TYPE_INDEXED() allows creating a data structure composed of a sequence of blocks containing a variable number of elements separated by a variable stride in memory. The stride is given in number of elements.
- MPI_TYPE_CREATE_HINDEXED() has the same functionality as MPI_TYPE_INDEXED() except that the strides separating two data blocks are given in bytes. This subroutine is useful when the old datatype is not an MPI base datatype(MPI_INTEGER, MPI_REAL, ...). We cannot therefore give the stride in number of elements of the old datatype.
- For MPI_TYPE_CREATE_HINDEXED(), as for MPI_TYPE_CREATE_HVECTOR(), use MPI_TYPE_SIZE() or MPI_TYPE_GET_EXTENT() in order to obtain in a portable way the size of the stride in bytes.

Derived datatypes

Homogenous datatypes of variable strides : MPI_TYPE_INDEXED()

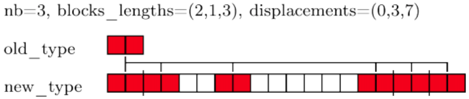


Figure: The MPI_TYPE_INDEXED constructor

```fortran
MPI_TYPE_INDEXED(nb,block_lengths,displacements,old_type,new_type,↩
      code)

integer,intent(in) :: nb
integer,intent(in),dimension(nb) :: block_lengths
integer,intent(in),dimension(nb) :: displacements
integer,intent(in) :: old_type

integer,intent(out) :: new_type,code
```

```python
Datatype.Create_indexed(self, blocklengths, displacements)
```

Derived datatypes

Homogenous datatypes of variable strides : MPI_TYPE_INDEXED()

- Example :

```python
size = 10
count = 3

counts = [2, 1, 3]
displacements = [0, 3, 7]

indexedtype = MPI.INT64_T.Create_indexed(counts, displacements)
indexedtype.Commit()

if rank==0:
    data = np.array([i for i in range(size)], dtype=np.float64)
    comm.Send([data,1,indexedtype],dest=1)
    print("Original data",data, rank)
elif rank==1:
    data = -1*np.ones(size, dtype=np.float64)
    comm.Recv([data,1,indexedtype],source=0)
    print("Received data",data, rank)

indexedtype.Free()
```

```
mpirun -n 2 python3 create_indexed.py
Original data [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.] 0
Received data [ 0.  1. -1.  3. -1. -1. -1.  7.  8.  9.] 1
```

Derived datatypes

Homogenous datatypes of variable strides : MPI_TYPE_CREATE_HINDEXED()

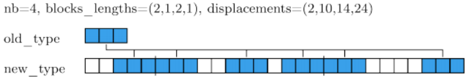


Figure: The MPI_TYPE_CREATE_HINDEXED constructor

```
MPI_TYPE_CREATE_HINDEXED(nb, block_lengths,displacements, old_type↩
     ,new_type,code)

integer,intent(in) :: nb
integer,intent(in),dimension(nb) :: block_lengths
integer(kind=MPI_ADDRESS_KIND),intent(in),dimension(nb) :: ↩
     displacements
integer,intent(in) :: old_type

integer,intent(out) :: new_type,code
```

```
Datatype.Create_hindexed(self, blocklengths, displacements)
```

Derived datatypes

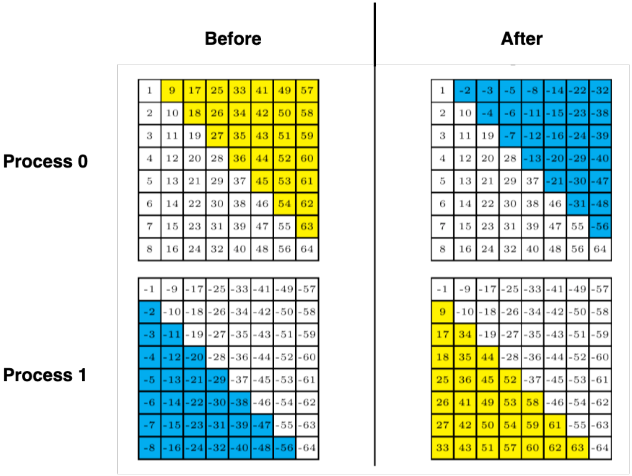
Homogenous datatypes of variable strides : MPI_TYPE_INDEXED()

- Example : triangular matrix

In the following example, each of the two processes :

1. Initializes its matrix (positive growing numbers on process 0 and negative decreasing numbers on process 1).
2. Constructs its datatype : triangular matrix (superior for the process 0 and inferior for the process 1).
3. Sends its triangular matrix to the other process and receives back a triangular matrix which it stores in the same place which was occupied by the sent matrix.
4. Frees its resources.

Derived datatypes



1	9	17	25	33	41	49	57
2	10	18	26	34	42	50	58
3	11	19	27	35	43	51	59
4	12	20	28	36	44	52	60
5	13	21	29	37	45	53	61
6	14	22	30	38	46	54	62
7	15	23	31	39	47	55	63
8	16	24	32	40	48	56	64

1	-2	-3	-5	-8	-14	-22	-32
2	10	-4	-6	-11	-15	-23	-38
3	11	19	-7	-12	-16	-24	-39
4	12	20	28	-13	-20	-29	-40
5	13	21	29	37	-21	-30	-47
6	14	22	30	38	46	-31	-48
7	15	23	31	39	47	55	-56
8	16	24	32	40	48	56	64

-1	-9	-17	-25	-33	-41	-49	-57
-2	-10	-18	-26	-34	-42	-50	-58
-3	-11	-19	-27	-35	-43	-51	-59
-4	-12	-20	-28	-36	-44	-52	-60
-5	-13	-21	-29	-37	-45	-53	-61
-6	-14	-22	-30	-38	-46	-54	-62
-7	-15	-23	-31	-39	-47	-55	-63
-8	-16	-24	-32	-40	-48	-56	-64

-1	-9	-17	-25	-33	-41	-49	-57
9	-10	-18	-26	-34	-42	-50	-58
17	34	-19	-27	-35	-43	-51	-59
18	35	44	-28	-36	-44	-52	-60
25	36	45	52	-37	-45	-53	-61
26	41	49	53	58	-46	-54	-62
27	42	50	54	59	61	-55	-63
33	43	51	57	60	62	63	-64

Figure: – Exchange between the two processes

Derived datatypes

Homogenous datatypes of variable strides : MPI_TYPE_INDEXED()

- Example :

```python
from mpi4py import MPI
import numpy as np
comm = MPI.COMM_WORLD
nb_procs = comm.Get_size()
rank = comm.Get_rank()
n = 8; sign = -1
if rank == 0: sign = 1

a = [sign*i for i in range(1,n*n+1,1)]
Matrix = np.array(a)
Matrix = np.reshape(Matrix, (n,n)).transpose()

if rank == 0:
    displacements = [n*i for i in range(n)]
    block_lengths = [i for i in range(n)]
else:
    displacements = [n*i+i+1 for i in range(n)]
    block_lengths = [n-i-1 for i in range(n)]

type_triangle = MPI.DOUBLE.Create_indexed(block_lengths, displacements)
type_triangle.Commit()

num_proc = (rank+1)%2
comm.Send([Matrix,1,type_triangle],dest=num_proc)
comm.Recv([Matrix,1,type_triangle],source=num_proc)

type_triangle.Free()
print(Matrix, rank)
```

Derived datatypes

Homogenous datatypes of variable strides : MPI_TYPE_INDEXED()

- Example : Matrix after permutation

```
mpirun -n 2 python3 matrixExchange.py
[[  1  -2  -3  -5  -8 -14 -22 -32]
[  2  10  -4  -6 -11 -15 -23 -38]
[  3  11  19  -7 -12 -16 -24 -39]
[  4  12  20  28 -13 -20 -29 -40]
[  5  13  21  29  37 -21 -30 -47]
[  6  14  22  30  38  46 -31 -48]
[  7  15  23  31  39  47  55 -56]
[  8  16  24  32  40  48  56  64]] 0

[[ -1  -9 -17 -25 -33 -41 -49 -57]
[  9 -10 -18 -26 -34 -42 -50 -58]
[ 17  34 -19 -27 -35 -43 -51 -59]
[ 18  35  44 -28 -36 -44 -52 -60]
[ 25  36  45  52 -37 -45 -53 -61]
[ 26  41  49  53  58 -46 -54 -62]
[ 27  42  50  54  59  61 -55 -63]
[ 33  43  51  57  60  62  63 -64]] 1
```

Communicators

Introduction

The purpose of communicators is to create subgroups on which we can carry out operations such as collective or point-to-point communications. Each subgroup will have its own communication space.

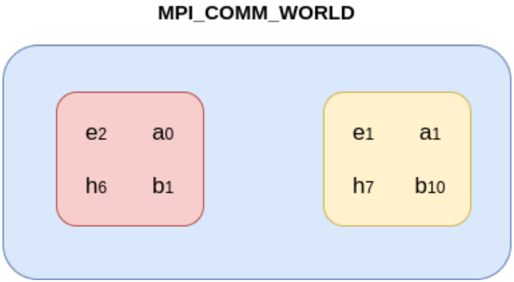


Figure: Communicator partitioning

Communicators

Example

For example, we want to broadcast a collective message to even-ranked processes and another message to odd-ranked processes.

- Looping on send/recv can be very detrimental especially if the number of processes is high. Also a test inside the loop would be compulsory in order to know if the sending process must send the message to an even or odd process rank.
- A solution is to create a communicator containing the even-ranked processes, another containing the odd-ranked processes, and initiate the collective communications inside these groups.

Communicators

Default communicator

- A communicator can only be created from another communicator. The first one will be created from the MPI_COMM_WORLD.
- After the MPI_INIT() call, a communicator is created for the duration of the program execution.
- Its identifier MPI_COMM_WORLD is an integer value defined in the header files.
- This communicator can only be destroyed via a call to MPI_FINALIZE().
- By default, therefore, it sets the scope of collective and point-to-point communications to include all the processes of the application.

Communicators

Groups and communicators

- A communicator consists of :
  - A group, which is an ordered group of processes.
  - A communication context put in place by calling one of the communicator construction subroutines, which allows determination of the communication space.
- The communication contexts are managed by MPI (the programmer has no action on them : It is a hidden attribute).
- In the MPI library, the following subroutines exist for the purpose of building communicators : MPI_COMM_CREATE(), MPI_COMM_DUP(), MPI_COMM_SPLIT()
- The communicator constructors are collective calls.
- Communicators created by the programmer can be destroyed by using the MPI_COMM_FREE() subroutine.

Communicators

Partitioning of a communicator

In order to solve the problem example :

- Partition the communicator into odd-ranked and even-ranked processes.
- Broadcast a message inside the odd-ranked processes and another message inside the even-ranked processes.

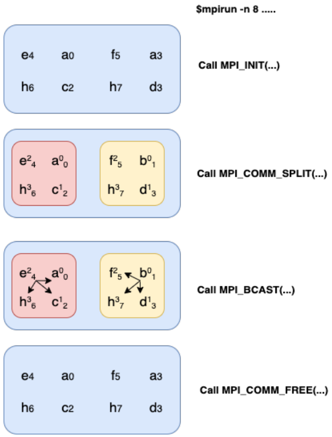


Figure: Communicator creation/destruction

Communicators

Partitioning of a communicator with MPI_COMM_SPLIT()

The MPI_COMM_SPLIT() subroutine allows :

- Partitioning a given communicator into as many communicators as we want.
- Giving the same name to all these communicators : The process value will be the value of its communicator.
- Method :
  1. Define a colour value for each process, associated with its communicator number.
  2. Define a key value for ordering the processes in each communicator
  3. Create the partition where each communicator is called new_comm

```
    MPI_COMM_SPLIT(comm,color,key,new_comm,code)

    integer, intent(in) :: comm, color, key

    integer, intent(out) :: new_comm, code
```

```
Comm.Split(self, int color=0, int key=0)
```

A process which assigns a color value equal to MPI_UNDEFINED will have the invalid communicator MPI_COMM_NULL for new_com.

Communicators

Example

Let's look at how to proceed in order to build the communicator which will subdivide the communication space into odd-ranked and even-ranked processes via the MPI_COMM_SPLIT() constructor.

Communicators

Partitioning of a communicator with MPI_COMM_SPLIT()

Example :

```python
...
world_rank = comm.Get_rank(); world_size = comm.Get_size()

m = 5; a = np.zeros(m)

if world_rank==2: a[:] = 2.
if world_rank==5: a[:] = 5.

key = world_rank
if world_rank==2 or world_rank==5 :
    key=-1
color = world_rank%2

newcomm = comm.Split(color, key)

row_rank = newcomm.Get_rank(); row_size = newcomm.Get_size()

commname = "Comm-"+str(color)
newcomm.Set_name(commname)

print("WORLD RANK/SIZE: {RANK}/{SIZE} \t ROW RANK/SIZE: {ROW_RANK}/{ROW_SIZE} \t Comm name: {commname}".format(RANK=world_rank, SIZE=world_size, ROW_RANK=row_rank, ROW_SIZE=row_size, commname=commname))

newcomm.Bcast(a, root=0);

newcomm.Free()
```

Communicators

Partitioning of a communicator with MPI_COMM_SPLIT()

Results :

```
mpirun -n 6 python3 split_communicator.py

WORLD RANK/SIZE: 5/6   ROW RANK/SIZE: 0/3   Comm name: Comm-1
WORLD RANK/SIZE: 0/6   ROW RANK/SIZE: 1/3   Comm name: Comm-0
WORLD RANK/SIZE: 1/6   ROW RANK/SIZE: 1/3   Comm name: Comm-1
WORLD RANK/SIZE: 2/6   ROW RANK/SIZE: 0/3   Comm name: Comm-0
WORLD RANK/SIZE: 3/6   ROW RANK/SIZE: 2/3   Comm name: Comm-1
WORLD RANK/SIZE: 4/6   ROW RANK/SIZE: 2/3   Comm name: Comm-0
```

Communicators

Communicator built from a group

- We can also build a communicator by defining a group of processes : Call to MPI_COMM_GROUP(), MPI_GROUP_INCL(), MPI_COMM_CREATE(), MPI_GROUP_FREE()
- This process is however far more cumbersome than using MPI_COMM_SPLIT() whenever possible.

Communicators

Topologies

- In most applications, especially in domain decomposition methods where we match the calculation domain to the process grid, it is helpful to be able to arrange the processes according to a regular topology.
- MPI allows defining virtual cartesian or graph topologies.
  - Cartesian topologies :
    - I Each process is defined in a grid.
    - I Each process has a neighbour in the grid.
    - I The grid can be periodic or not.
    - I The processes are identified by their coordinates in the grid.
  - Graph topologies :
    - I Can be used in more complex topologies.

Communicators

Cartesian topologies

- A Cartesian topology is defined from a given communicator named comm_old, calling the MPI_CART_CREATE() subroutine.
- We define :
  - An integer ndims representing the number of grid dimensions.
  - An integer array dims of dimension ndims showing the number of processes in each dimension.
  - An array of ndims logicals which shows the periodicity of each dimension.
  - A logical reorder which shows if the process numbering can be changed by MPI.

```
MPI_CART_CREATE(comm_old, ndims,dims,periods,reorder,comm_new,code↩
     )

integer, intent(in) :: comm_old, ndims
integer, dimension(ndims),intent(in) :: dims
logical, dimension(ndims),intent(in) :: periods
logical, intent(in) :: reorganization

integer, intent(out) :: comm_new, code
```

```
Intracomm.Create_cart(self, dims, periods=None, bool reorder=False)
```

Communicators

2D Example

Example on a grid having 2 domains along x and 2 along y, periodic in y.

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
nb_procs = comm.Get_size()
rank = comm.Get_rank()

periods = tuple([False,False])
reorder = False
dims = [2,2]


cart2d = comm.Create_cart(
dims    = dims,
periods = periods,
reorder = reorder
)
```

- If reorder = .false. then the rank of the processes in the new communicator (comm_2D) is the same as in the old communicator (MPI_COMM_WORLD).
- If reorder = .true., the MPI implementation chooses the order of the processes.

Communicators

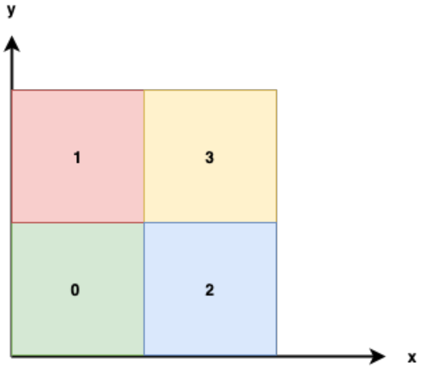


Figure: A 2D non-periodic Cartesian topology

Communicators

3D Example

Example on a 3D grid having 2 domains along x, 2 along y and 2 along z, non periodic.

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
nb_procs = comm.Get_size()
rank = comm.Get_rank()

periods = tuple([False,False, False])
reorder = False
dims = [2,2,2]


cart3 = comm.Create_cart(
dims    = dims,
periods = periods,
reorder = reorder
)
```

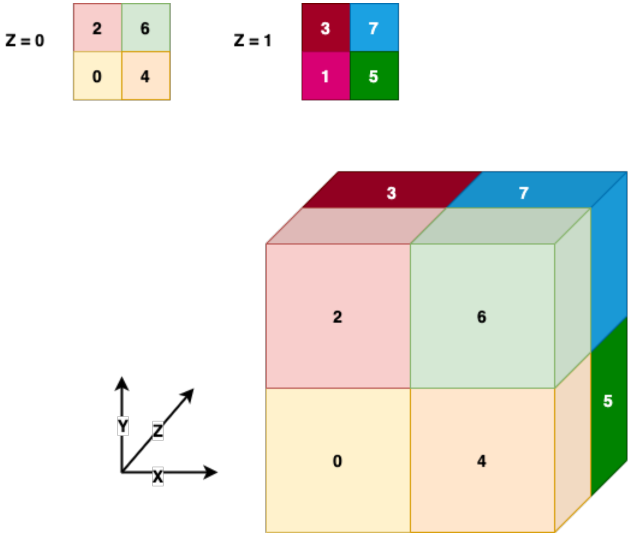


Figure: A 3D non-periodic Cartesian topology

Communicators

Process distribution

The MPI_DIMS_CREATE() subroutine returns the number of processes in each dimension of the grid according to the total number of processes.

```fortran
MPI_DIMS_CREATE(nb_procs,ndims,dims,code)

integer, intent(in) :: nb_procs, ndims
integer, dimension(ndims),intent(inout) :: dims

integer, intent(out) :: code
```

Remark : If the values of dims in entry are all 0, then we leave to MPI the choice of the number of processes in each direction according to the total number of processes.

Communicators

Rank od a process

In a Cartesian topology, the MPI_CART_RANK() subroutine returns the rank of the associated process to the coordinates in the grid.

```
MPI_CART_RANK(comm,coords,rank,code)

integer, intent(in) :: comm
integer, dimension(ndims),intent(in) :: coords

integer, intent(out) :: rank, code
```

```
Cartcomm.Get_cart_rank(self, coords)
```

Communicators

Coordinates of a process

In a cartesian topology, the MPI_CART_COORDS() subroutine returns the coordinates of a process of a given rank in the grid.

```
MPI_CART_COORDS(comm, rank, ndims, coords, code)

integer, intent(in) :: comm, rank, ndims
integer, dimension(ndims),intent(out) :: coords

integer, intent(out) :: code
```

```
Cartcomm.Get_coords(self, int rank)
```

Communicators

Example : MPI_CART_COORDS()

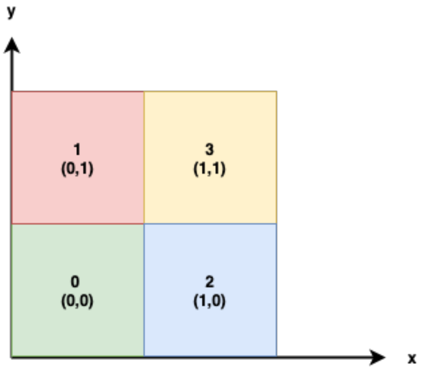


Figure: A 2D non-periodic Cartesian topology

Communicators

Example : MPI_CART_COORDS()

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

periods = tuple([False,False])
reorder = False
dims = [2,2]

cart2d = comm.Create_cart(
dims    = dims,
periods = periods,
reorder = reorder
)

coord2d = cart2d.Get_coords(rank)

print("I'm rank", rank, "my 2d coords are", coord2d)
```

```
mpirun -n 4 python3 coordinate_2d_cart.py

I'm rank 0 my 2d coords are [0, 0]
I'm rank 1 my 2d coords are [0, 1]
I'm rank 2 my 2d coords are [1, 0]
I'm rank 3 my 2d coords are [1, 1]
```

Communicators

Rank of neighbours

In a Cartesian topology, a process that calls the MPI_CART_SHIFT() subroutine can obtain the rank of a neighboring process in a given direction.

```fortran
MPI_CART_SHIFT(comm, direction, step, rank_previous, rank_next, ↩
      code)

integer, intent(in) :: comm, direction, step
integer, intent(out) :: rank_previous,rank_next

integer, intent(out) :: code
```

```python
Cartcomm.Shift(self, int direction, int disp)
```

- The direction parameter corresponds to the displacement axis (xyz).
- The step parameter corresponds to the displacement step.
- If a rank does not have a neighbor before (or after) in the requested direction, then the value of the previous (or following) rank will be MPI_PROC_NULL.

Communicators

Example : MPI_CART_SHIFT()

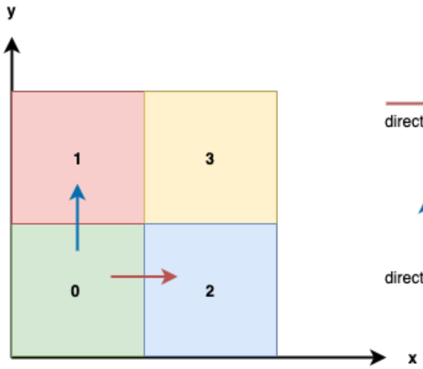


Figure: Call of the MPI_CART_SHIFT() subroutine

Communicators

Example : MPI_CART_SHIFT()

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

periods = tuple([False,False])
reorder = False
dims = [2,2]

cart2d = comm.Create_cart(
dims    = dims,
periods = periods,
reorder = reorder
)

left,right = cart2d.Shift(direction = 0, disp=1)
low,high   = cart2d.Shift(direction = 1, disp=1)

print("I'm rank", rank, "my (left,right) neighbours are",( left, right),
"my (low,high) neighbours are",( low, high))
```

Communicators

Example : MPI_CART_SHIFT()

```
mpirun -n 4 python3 neighbours_2d_cart.py

I'm rank 0
my (left,right) neighbours are (-2, 2) my (low,high) neighbours are (-2, 1)

I'm rank 1
my (left,right) neighbours are (-2, 3) my (low,high) neighbours are (0, -2)

I'm rank 2
my (left,right) neighbours are (0, -2) my (low,high) neighbours are (-2, 3)

I'm rank 3
my (left,right) neighbours are (1, -2) my (low,high) neighbours are (2, -2)
```

Communicators

3D Example : coordinates and neighbours

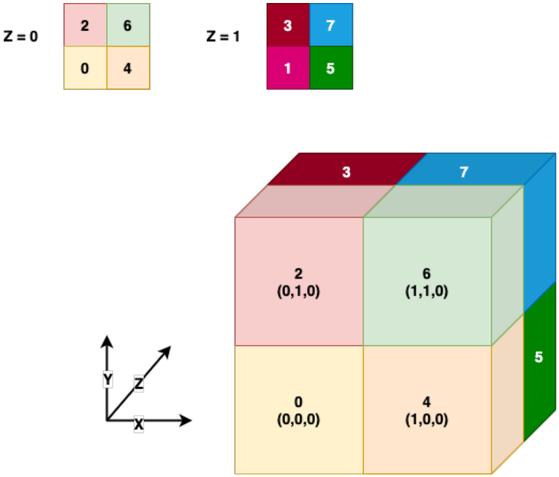


Figure: 3D Coordinates and Neighbours

Communicators

3D Example : coordinates and neighbours

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

periods = tuple([False, False, False])
reorder = False
dims = [2,2,2]

cart3d = comm.Create_cart(
dims    = dims,
periods = periods,
reorder = reorder
)

coord3d        = cart3d.Get_coords(rank)
left,right     = cart3d.Shift(direction = 0, disp=1)
low,high       = cart3d.Shift(direction = 1, disp=1)
ahead,before   = cart3d.Shift(direction = 2, disp=1)

print("I'm rank", rank, "my 3d coords are", coord3d, "my (left,right) neighbours ↩
    are",( left, right), "my (low,high) neighbours are", (low, high), "my (ahead↩
    ,before) neighbours are", (ahead,before))
```

Communicators

3D Example : coordinates and neighbours

```
mpirun -n 8 --oversubscribe python3 create_3d_cart.py

I'm rank 0 my 3d coords are [0, 0, 0]
my (left,right) neighbours are (-2, 4)
my (low,high) neighbours are (-2, 2)
my (ahead,before) neighbours are (-2, 1)

I'm rank 1 my 3d coords are [0, 0, 1]
my (left,right) neighbours are (-2, 5)
my (low,high) neighbours are (-2, 3)
my (ahead,before) neighbours are (0, -2)

I'm rank 2 my 3d coords are [0, 1, 0]
my (left,right) neighbours are (-2, 6)
my (low,high) neighbours are (0, -2)
my (ahead,before) neighbours are (-2, 3)

...
```