Parallel Programming with MPI - Day 4

Science & Technology Support
High Performance Computing

Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212-1163
Table of Contents

• Virtual Topologies

• Problem Set
Virtual Topologies

- Virtual Topologies
- Topology Types
- Creating a Cartesian Virtual Topology
- Cartesian Example

- Cartesian Mapping Functions
  - MPI_CART_RANK*
  - MPI_CART_COORDS*
  - MPI_CART_SHIFT*

- Cartesian Partitioning

*includes sample C and Fortran programs
Virtual Topologies

• Convenient process naming
• Naming scheme to fit the communication pattern
• Simplifies writing of code
• Can allow MPI to optimize communications
• Rationale: access to useful topology routines
How to use a Virtual Topology

• Creating a topology produces a new communicator

• MPI provides “mapping functions”

• Mapping functions compute processor ranks, based on the topology naming scheme
Example - 2D Torus

(dim 1) 0 (0,0)  3 (1,0)  6 (2,0)  9 (3,0)  
         1 (0,1)  4 (1,1)  7 (2,1)  10 (3,1)  
         2 (0,2)  5 (1,2)  8 (2,2)  11 (3,2)  
(dim 0)
Topology types

• **Cartesian topologies**
  – Each process is connected to its neighbors in a virtual grid
  – Boundaries can be cyclic
  – Processes can be identified by Cartesian coordinates

• **Graph topologies**
  – General graphs
  – Will not be covered here
Creating a Cartesian Virtual Topology

**C:**

```c
int MPI_Cart_create (MPI_Comm comm_old, int ndims,
                     int *dims, int *periods, int reorder,
                     MPI_Comm *comm_cart)
```

**Fortran:**

```fortran
INTEGER COMM_OLD,NDIMS,DIMS(*),COMM_CART,IERROR
LOGICAL PERIODS(*),REORDER

CALL MPI_CART_CREATE(COMM_OLD,NDIMS,DIMS,PERIODS,REORDER,
                      COMM_CART,IERROR)
```
## Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>comm_old</td>
<td>existing communicator</td>
</tr>
<tr>
<td>ndims</td>
<td>number of dimensions</td>
</tr>
<tr>
<td>periods</td>
<td>logical array indicating whether a dimension is cyclic</td>
</tr>
<tr>
<td></td>
<td>(If TRUE, cyclic boundary conditions)</td>
</tr>
<tr>
<td>reorder</td>
<td>logical</td>
</tr>
<tr>
<td></td>
<td>(If FALSE, rank preserved)</td>
</tr>
<tr>
<td></td>
<td>(If TRUE, possible rank reordering)</td>
</tr>
<tr>
<td>comm_cart</td>
<td>new cartesian communicator</td>
</tr>
</tbody>
</table>
Cartesian Example

MPI_Comm vu;
int dim[2], period[2], reorder;

dim[0]=4; dim[1]=3;
period[0]=TRUE; period[1]=FALSE;
reorder=TRUE;

MPI_Cart_create(MPI_COMM_WORLD, 2, dim, period, reorder, &vu);
Cartesian Mapping Functions

Mapping process grid coordinates to ranks

C:

```c
int MPI_Cart_rank (MPI_Comm comm, init *coords, int *rank)
```

Fortran:

```fortran
INTEGER COMM,COORDS(*),RANK,IERROR

CALL MPI_CART_RANK(COMM,COORDS,RANK,IERROR)
```
Cartesian Mapping Functions

Mapping ranks to process grid coordinates

C:

```c
int MPI_Cart_coords (MPI_Comm comm, int rank, int maxdims,
                     int *coords)
```

Fortran:

```fortran
INTEGER COMM,RANK,MAXDIMS,COORDS(*),IERROR

CALL MPI_CART_COORDS(COMM,RANK,MAXDIMS,COORDS,IERROR)
```
Sample Program #9 - C

```c
#include<mpi.h>
/* Run with 12 processes */
void main(int argc, char *argv[]) {
    int rank;
    MPI_Comm vu;
    int dim[2],period[2],reorder;
    int coord[2],id;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    dim[0]=4; dim[1]=3;
    period[0]=TRUE; period[1]=FALSE;
    reorder=TRUE;
    MPI_Cart_create(MPI_COMM_WORLD,2,dim,period,reorder,&vu);
    if(rank==5){
        MPI_Cart_coords(vu,rank,2,coord);
        printf("P:%d My coordinates are %d %d\n",rank,coord[0],coord[1]);
    }
    if(rank==0) {
        coord[0]=3; coord[1]=1;
        MPI_Cart_rank(vu,coord,&id);
        printf("The processor at position (%d, %d) has rank %d\n",coord[0],coord[1],id);
    }
    MPI_Finalize();
}
```

The processor at position (3,1) has rank 10
P:5 My coordinates are 1 2
Sample Program #9 - Fortran

```fortran
PROGRAM Cartesian
    C Run with 12 processes
    INCLUDE 'mpif.h'
    INTEGER err, rank, size
    integer vu,dim(2),coord(2),id
    logical period(2),reorder
    CALL MPI_INIT(err)
    CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
    CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,err)
    dim(1)=4
    dim(2)=3
    period(1)=.true.
    period(2)=.false.
    reorder=.true.
    call MPI_CART_CREATE(MPI_COMM_WORLD,2,dim,period,reorder,vu,err)
    if(rank.eq.5) then
        call MPI_CART_COORDS(vu,rank,2,coord,err)
        print*,'P:',rank,' my coordinates are',coord
    end if
    if(rank.eq.0) then
        coord(1)=3
        coord(2)=1
        call MPI_CART_RANK(vu,coord,id,err)
        print*,'P:',rank,' processor at position',coord,' is',id
    end if
    CALL MPI_FINALIZE(err)
END
```

P:5 my coordinates are 1, 2
P:0 processor at position 3, 1 is 10
Cartesian Mapping Functions

Computing ranks of neighboring processes

C:

```c
int MPI_Cart_shift (MPI_Comm comm, int direction, int disp,
                   int *rank_source, int *rank_dest)
```

Fortran:

```fortran
INTEGER COMM,DIRECTION,DISP,RANK_SOURCE,RANK_DEST,IERROR

CALL MPI_CART_SHIFT(COMM,DIRECTION,DISP,RANK_SOURCE,
                     RANK_DEST,IERROR)
```
MPI_Cart_shift

- Does **not** actually shift data: returns the correct ranks for a shift which can be used in subsequent communication calls

- **Arguments:**
  - `direction` - dimension in which the shift should be made
  - `disp` - length of the shift in processor coordinates (+ or -)
  - `rank_source` - where calling process should receive a message **from** during the shift
  - `rank_dest` - where calling process should send a message **to** during the shift

- If shift off of the topology, `MPI_Proc_null` is returned
Sample Program #10 - C

```c
#include<mpi.h>
#define TRUE 1
#define FALSE 0
void main(int argc, char *argv[]) {
    int rank;
    MPI_Comm vu;
    int dim[2],period[2],reorder;
    int up,down,right,left;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    dim[0]=4; dim[1]=3;
    period[0]=TRUE; period[1]=FALSE;
    reorder=TRUE;
    MPI_Cart_create(MPI_COMM_WORLD,2,dim,period,reorder,&vu);
    if(rank==9){
        MPI_Cart_shift(vu,0,1,&left,&right);
        MPI_Cart_shift(vu,1,1,&up,&down);
        printf("P:%d My neighbors are r: %d d:%d 1:%d u:%d\n",rank,right,down,left,up);
    }
    MPI_Finalize();
}
```

P:9 my neighbors are r:0 d:10 1:6 u:-1
PROGRAM neighbors
C
C Run with 12 processes
C
INCLUDE 'mpif.h'
INTEGER err, rank, size
integer vu
integer dim(2)
logical period(2),reorder
integer up,down,right,left
CALL MPI_INIT(err)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,err)
dim(1)=4
dim(2)=3
period(1)=.true.
period(2)=.false.
reorder=.true.
call MPI_CART_CREATE(MPI_COMM_WORLD,2,dim,period,reorder,vu,err)
if(rank.eq.9) then
  call MPI_CART_SHIFT(vu,0,1,left,right,err)
call MPI_CART_SHIFT(vu,1,1,up,down,err)
  print*,'P:',rank,' neighbors (rdlu)are',right,down,left,up
end if
CALL MPI_FINALIZE(err)
END

P:9 neighbors (rdlu) are 0, 10, 6, -1
Cartesian Partitioning

- Often we want to do an operation on only part of an existing Cartesian topology
- Cut a grid up into ‘slices’
- A new communicator is produced for each slice
- Each slice can then perform its own collective communications
- \texttt{MPI\_Cart\_sub} and \texttt{MPI\_CART\_SUB} generate new communicators for the slice
MPI_Cart_sub

C:

    int MPI_Cart_sub (MPI_Comm comm, int *remain_dims,
                       MPI_Comm *newcomm)

Fortran:

    INTEGER   COMM,NEWCOMM,IERROR
    LOGICAL   REMAIN_DIMS(*)

    CALL MPI_CART_SUB(COMM,REMAIN_DIMS,NEWCOMM,IERROR)

• If `comm` is a 2x3x4 grid and `remain_dims`={TRUE,FALSE,TRUE},
  then three new communicators are created each being a 2x4 grid

• Calling processor receives back only the new communicator it is in
Problem Set

1) Write a program that will make a virtual topology using 8 processors. The topology should consist of 4 processor rows and 2 processors columns with no wrap-around (cyclic) in either dimension. Each processor in the topology should have an integer variable with the same values used in Problem 4 (Day 3).

After your program has created the topology, it should use virtual topology utility functions to have each processor calculate the average value of its integer and the integers contained in it’s neighbors. Each processor should then output its calculated average. (NOTE: do not use “diagonal” neighbors in the averaging. Only use “up”, “down”, “left”, and “right” neighbors, if they exist).