COMS4040A & COMS7045A: High Performance Computing & Scientific Data Management
Introduction to MPI III

Hairong Wang

School of Computer Science & Applied Mathematics
University of the Witwatersrand, Johannesburg

2022-6-2

Contents

WITS UNIVERSITY

Outline

1. MPI Derived Datatypes
   - Typemap
   - Creating and Using a New Datatype
   - Contiguous Type
   - Vector Type
   - Indexed Type
   - Struct Type

2. Groups and Communicators
   - Groups
   - Contexts
   - Communicators
   - Example: Monte Carlo Computation of Π
   - Group Management
   - Communicator Management

WITS UNIVERSITY

MPI Derived Datatypes

Example 1

```
double x[1000];
....
for(i=0; i<1000; i++){
  if(my_rank == 0)
    MPI_Send(&x[i], 1, MPI_DOUBLE, 1, 0, comm);
  else
    MPI_Recv(&x[i], 1, MPI_DOUBLE, 0, 0,comm,&status);
}
/*the following is more efficient than using the for loop*/
if(my_rank == 0)
  MPI_Send(&x[0], 1000, MPI_DOUBLE, 1, 0, comm);
else
MPI_Recv(&x[0], 1000, MPI_DOUBLE, 0, 0, comm, &status);
```

In distributed-memory systems, communication can be much more expensive than local computation. Thus, if we can reduce the number of communications, we are likely to improve the performance of our programs.

WITS UNIVERSITY

MPI Built-in Datatypes

- The MPI standard defines many built in datatypes, mostly mirroring standard C/C++ or FORTRAN datatypes
- These are sufficient when sending single instances of each type
- They are also usually sufficient when sending contiguous blocks of a single type
- Sometimes, however, we want to send non-contiguous data or data that is comprised of multiple types
- MPI provides a mechanism to create derived datatypes that are built from simple datatypes

MPI Derived Datatypes cont.

- In MPI, a *derived datatype* can be used to represent any collection of data items in memory by storing both the types of the items and their relative locations in memory.
- Why use derived datatypes?
  - Primitive datatypes are contiguous;
  - Derived datatypes allow you to specify non-contiguous data in a convenient manner and treat it as though it is contiguous;
  - Useful to
    - Make code more readable
    - Reduce number of messages and increase their size (faster since less latency);
    - Make code more efficient if messages of the same size are repeatedly used.

Outline

Typemap

Formally, a derived datatype in MPI is described by a typemap consists of a sequence of basic MPI datatypes together with a *displacement* for each of the datatypes. That is,

- a sequence of basic datatypes: $\{type_0, ..., type_{n-1}\}$
- a sequence of integer displacements: $\{displ_0, ..., displ_{n-1}\}$.
- Typemap = $\{(type_0, disp_0), \cdots, (type_{n-1}, disp_{n-1})\}$

For example, a typemap might consist of (double,0),(char,8) indicating the type has two elements:

- a double precision floating point value starting at displacement 0,
- and a single character starting at displacement 8.

WITS UNIVERSITY

Typemap cont.

- Types also have extent, which indicates how much space is required for the type
- The extent of a type may be more than the sum of the bytes required for each component
- For example, on a machine that requires double-precision numbers to start on an 8-byte boundary, the type (double,0),(char,8) will have an extent of 16 even though it only requires 9 bytes

WITS UNIVERSITY

Outline

WITS UNIVERSITY

Creating and Using a New Datatype

Three steps are necessary to create and use a new datatype in MPI:

- Create the type using one of MPI's type construction functions
- Commit the type using `MPI_Type_commit()`.
- Release the datatype using `MPI_Type_free()` when it is not needed any more.

WITS UNIVERSITY

MPI Derived Datatypes cont.

MPI provides several methods for constructing derived datatypes to handle a wide variety of situations.

- Contiguous
- Vector
- Indexed
- Struct

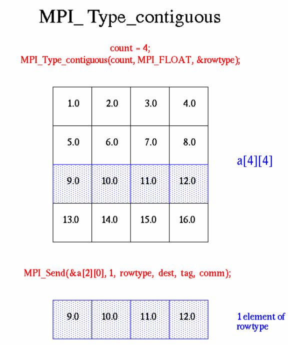
Outline

WITS UNIVERSITY

Contiguous Type

Contiguous: The contiguous datatype allows for a single type to refer to contiguous multiple elements of an existing datatype.

```
int MPI_Type_contiguous(
    int count,              //count
    MPI_Datatype oldtype,   //old datatype
    MPI_Datatype *newtype)  //new datatype
```



To define the new datatype in this example and release it after finished using it:

```
MPI_Datatype rowtype;
MPI_Type_contiguous(4, MPI_DOUBLE,
    &rowtype);
MPI_Type_commit(&rowtype);
......
MPI_Type_free(&rowtype);
```

Contiguous Type cont.

To define a new datatype:

- Declare the new datatype as `MPI_Datatype`.
- Construct the new datatype.
- Before we can use a derived datatype in a communication function, we must first **commit** it with a call to

  ```
  int MPI_Type_commit(MPI_Datatype* datatype);
  ```

  Commits new datatype to the system. Required for all derived datatypes.
- When we finish using the new datatype, we can free any additional storage used with a call to

  ```
  int MPI_Type_free(MPI_Datatype* datatype)
  ```

Contiguous Type cont.

The new datatype is essentially an array of `count` elements having type `oldtype`. For example, the following two code fragments are equivalent:

```
MPI_Send (a,n,MPI_DOUBLE,dest,tag,MPI_COMM_WORLD);
```

and

```
MPI_Datatype rowtype;
MPI_Type_contiguous(n, MPI_DOUBLE, &rowtype);
MPI_Type_commit(&rowtype);
MPI_Send(a, 1, rowtype, dest, tag, MPI_COMM_WORLD);
```

Example 2

```c
#define SIZE 4
float a[SIZE][SIZE] =
  {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
  9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0};
float b[SIZE];
MPI_Status stat;
MPI_Datatype rowtype;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
MPI_Type_commit(&rowtype);
if(numtasks == SIZE){
  if(rank == 0)
    for (i=0; i<numtasks; i++)
      MPI_Send(&a[i][0], 1, rowtype, i, tag, MPI_COMM_WORLD);
  MPI_Recv(b,SIZE,MPI_FLOAT,source,tag,MPI_COMM_WORLD,&stat);
  printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n",
    rank,b[0],b[1],b[2],b[3]);
}
MPI_Type_free(&rowtype);
MPI_Finalize();
```

Outline

WITS UNIVERSITY

Vector Type

Vector: The vector datatype is similar to the contiguous datatype but allows for a constant non-unit stride between elements.

```
int MPI_Type_vector(
    int count,
    int blocklength,
    int stride,
    MPI_Datatype oldtype,
    MPI_Datatype *newtype )
```

- Input parameters
  - `count`: number of blocks (nonnegative integer)
  - `blocklength`: number of elements in each block (integer)
  - `stride`: number of elements between each block (integer)
  - `oldtype`: old datatype

Vector Type cont.

- Output parameter
  - `newtype`: new datatype

MPI_ Type_vector

count = 4; blocklength = 1; stride = 4;
MPI_Type_vector(count, blocklength, stride, MPI_FLOAT,
&columntype);

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

a[4][4]

MPI_Send(&a[0][1], 1, columntype, dest, tag, comm);

2.0	6.0	10.0	14.0

1 element of columntype

WITS UNIVERSITY

Vector Type cont.

For example, the following two types can be used to communicate a single row and a single column of a matrix ($ny \times nx$):

```
MPI_Datatype rowType, colType;
MPI_Type_vector(nx, 1, 1, MPI_DOUBLE, &rowType);
MPI_Type_vector(ny, 1, nx, MPI_DOUBLE, &colType);
MPI_Type_commit(&rowType);
MPI_Type_commit(&colType);
```

WITS UNIVERSITY

Example 3

```c
#define SIZE 4
float a[SIZE][SIZE] =
  {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
  9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0};
float b[SIZE];
MPI_Status stat;
MPI_Datatype coltype;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Type_vector(SIZE, 1, SIZE, MPI_FLOAT, &coltype);
MPI_Type_commit(&coltype);
if(numtasks == SIZE){
  if(rank == 0){
    for (i=0; i<numtasks; i++)
      MPI_Send(&a[i][0], 1, coltype, i, tag, MPI_COMM_WORLD);
  }
  MPI_Recv(b,SIZE,MPI_FLOAT,source,tag,MPI_COMM_WORLD,&stat);
}
MPI_Type_free(&coltype);
MPI_Finalize();
```

WITS UNIVERSITY

Outline

Indexed Type

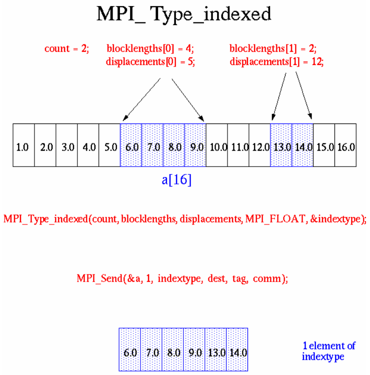
Indexed: The indexed datatype provides for varying strides between elements.

```
int MPI_Type_indexed(
    int count,
    int blocklens[],
    int indices[],
    MPI_Datatype oldtype,
    MPI_Datatype *newtype )
```

- Input parameters
  - `count`: number of blocks — also number of entries in `indices` and `blocklens`
  - `blocklens`: number of elements in each block (array of nonnegative integers)
  - `indices`: displacement of each block in multiples of `oldtype` (array of integers)
  - `oldtype`: old datatype

Indexed Type cont.

- Output parameters
  - `newtype`: new datatype



WITS UNIVERSITY

Indexed Type cont.

Indexed type generalizes the vector type; instead of a constant stride, blocks can be of varying length and displacements.

```
int blocklen[] = {4, 2, 2, 6, 6};
int disp[] = {0, 8, 12, 16, 23};
MPI_Datatype mytype;
MPI_Type_indexed(5, blocklen, disp, MPI_DOUBLE,
    &mytype);
MPI_Type_commit(&mytype);
......
MPI_Type_free(&mytype);
```

Outline

1. MPI Derived Datatypes
   - Typemap
   - Creating and Using a New Datatype
   - Contiguous Type
   - Vector Type
   - Indexed Type
   - Struct Type
2. Groups and Communicators
   - Groups
   - Contexts
   - Communicators
   - Example: Monte Carlo Computation of Π
   - Group Management
   - Communicator Management

Struct Type

Struct: The most general constructor allows for the creation of types representing general C/C++ structs/classes.

- We can use `MPI_Type_create_struct` to build a derived datatype that consists of individual elements that have different basic types:

```
int MPI_Type_create_struct(
    int count, //number of elements in the datatype
    int array_of_blocklengths[], //length of each element
    MPI_Aint array_of_displacements[], //displacements in bytes
    MPI_Datatype array_of_types[],
    MPI_Datatype* new_type_p)
```

Struct Type cont.

- `count`: number of blocks, also number of entries in arrays `array_of_types`, `array_of_displacements` and `array_of_blocklengths`
- `array_of_blocklengths`: number of elements in each block
- `array_of_displacements`: byte displacement of each block
- `array_of_types`: type of elements in each block
- Output Parameters: `newtype`: new datatype

Struct Type Cont.

- To find the displacements, we can use the function `MPI_Get_address`:

  ```
  int MPI_Get_address(
      void* location_p,
      MPI_Aint* address_p);
  ```

  - It returns the address of the memory location referenced by `location_p`.
  - `MPI_Aint` is an integer type that is big enough to store an address on the system.

Example 4 (Moving particles between processes)

In N-body problems, the force between particles become less with growing distance. At great enough distance, the influence of a particle on others is negligible. A number of algorithms for N-body simulation take advantage of this fact. These algorithms organize the particles in groups based on their locations using tree structures such quad-tree. One important step in the implementation of these algorithms is that of transferring particles from one process to another as they move. Here, we only discuss a way in which movement of particles can be done in MPI.

Assume a particle is defined by

```
typedef struct {
    int x,y,z;
    double mass;
}Particle;
```

- To send a particle from one process to another, or broadcast the particle, it makes sense in MPI to create a datatype instead of sending the elements in the struct individually.

Example 4 cont.

```
    Particle my_particle;
    MPI_Datatype particletype;
    Build_mpi_type(&my_particle.x, &my_particle.y, &my_particle.z, &
        my_particle.mass, &particletype);
    /*process 0 does some computation with my_particle */
    ......
    /*process 0 performs a broadcast*/
    MPI_Bcast(&my_particle, 1, particletype, 0, MPI_COMM_WORLD);
    ......
    MPI_Type_free(&particletype);
    }
```

Struct Type cont.

Example 4 cont.

```
void Build_mpi_type( int* x_p, int* y_p, int* z_p, double* mass_p,
    MPI_Datatype* particletype_p) {
  int array_of_blocklengths[4] = {1, 1, 1, 1};
  MPI_Datatype array_of_types[4] = {MPI_INT, MPI_INT, MPI_INT,
      MPI_DOUBLE};
  MPI_Aint array_of_displacements[4] = {0};
  MPI_Get_address(x_p, &array_of_displacements[0]);
  MPI_Get_address(y_p, &array_of_displacements[1]);
  MPI_Get_address(z_p, &array_of_displacements[2]);
  MPI_Get_address(mass_p, &array_of_displacements[3]);
  for(int i=3; i<=0; i++)
    array_of_displacements[i] -= array_of_displacements[0];
  MPI_Type_create_struct(4, array_of_blocklengths,
  array_of_displacements, array_of_types,
  particletype_p);
  MPI_Type_commit(particletype_p);
} /* Build_mpi_type */
```

Outline

WITS UNIVERSITY

Some Important MPI Features

- Groups
- Contexts
- Communicators

Outline

WITS UNIVERSITY

Groups

- A *group* is an ordered set of processes.
- A group is used within a communicator to describe the participants in a communication "universe" and to rank such participants.
- Special predefined group: `MPI_GROUP_EMPTY` — a group with no members.
- Predefined constant: `MPI_GROUP_NULL` — a value used for invalid group handles. For example, `MPI_GROUP_NULL` is returned when a group is freed.
- `MPI_GROUP_EMPTY` is a valid group handles. `MPI_GROUP_NULL` is invalid group handles.

WITS UNIVERSITY

Outline

WITS UNIVERSITY

Contexts

- A *context* is the communication environment.
- A *context* is a property of communicators that allows partitioning of the communication space.
- A message sent in one context cannot be received in another context. Separate contexts are entirely independent.
- Contexts are not explicit MPI objects; they appear only as part of the realization of communicators.
- A context is essentially a system-managed tag (or tags) needed to make a communicator safe for point-to-point and MPI-defined collective communication.

Outline

Communicators

- Communicators bring together the concepts of group and context.
- MPI communication operations reference communicators to determine the scope and the "communication universe" in which a point-to-point or collective operation is to operate.
- Each communicator contains a group of valid participants.
- For collective communication, the intra communicator specifies the set of processes that participate in the collective operation.
  - Intracommunicator: Refers to the regular communicators of communication within a group.
  - Intercommunicator: Communicators target group-to-group communication

WITS UNIVERSITY

Communicators cont.

- Predefined intracommunicator `MPI_COMM_WORLD` of all processes the local process can communicate with after initialization is defined once `MPI_Init` has been called.
- Predefined `MPI_COMM_NULL` is the value for invalid communicator handle. Used as an error result from some functions.
- Predefined `MPI_COMM_SELF` includes only the process itself.
- Avoid using two communicators that overlap.
- You always start with an existing communicator and subdivide it to make one or more new ones.

Why go beyond MPI_COMM_WORLD

- To use collective communication on only some processes
- Need to do a task on only some processes
- Want to do several tasks in parallel

WITS UNIVERSITY

Outline

Example 5 (Monte Carlo Computation of Π)

- If the radius of a circle is 1, then the area is π, and the area of the square around the circle, with the same center point as the circle, is 4.
- The ratio r of the area of the circle to that of the square is $\frac{\pi}{4}$.
- Compute r by generating random points (x, y) in the square and counting how many of them turn out to be in the circle $(x^2 + y^2 < 1)$.

Example 5 cont.

- Use only one process (called server) to generate the random numbers, and distribute these to the other processes.
- We want the processes other then the server to compute the ratio — need to use collective communication.
- We need to have two communicators.

```
MPI_Comm world=MPI_COM_WORLD, workers;
MPI_Group world_group, worker_group;
int ranks[1];
MPI_Init(&argc, &argv);
MPI_Comm_size(world, &numprocs);
MPI_Comm_rank(world, &myid);
server = numprocs - 1; // the last process
MPI_Comm_group(world, $world_group); //exatract the group
ranks[0] = server;
MPI_Group_excl(world_group, 1, ranks, &worker_group);
MPI_Comm_create(world, worker_group, &workers);
MPI_Group_free(&worker_group);
MPI_Group_free(&world_group);
```

Example 5 cont.

The program may continue in the following way:

- The `server` process receives requests from workers for chunks of random numbers, generate these numbers, and then sends a unique chunk of random numbers to each worker who sent their requests.
- A worker process sends a request for random numbers to the server, receives the numbers, proceeds to test whether a pair of points fall in the circle or not, and accumulate the number of points fall in the circle, and those fall outside the circle, respectively.
- After computing a chunk of random numbers, the workers do a collective communication - `MPI_Allreduce` in this case, to compute an estimation of number π. If the estimation is not good enough, a worker needs to send another round of request to the server for a new chunk of random numbers.
- The stopping criteria is the error of estimated number π is less than a threshold, or a total number of random points to be inspected.

Example 5 cont.

- Complete Example 5 as an exercise.

Outline

WITS UNIVERSITY

Group Management

- Group Accessors
  - `int MPI_Group_size(MPI_Group group, int *size)`
    Returns the number of processes in the group.
  - `int MPI_Group_rank(MPI_Group group, int *rank)`
    Returns the rank of calling process in the group

Group Management cont.

- Group Constructors: Construct new groups from existing groups using various set operations.
  - ```
    int MPI_Group_incl(
        MPI_Group group,
        int n,
        const int ranks[],
        MPI_Group *newgroup)
    ```
    Creates a group `newgroup` that consists of the `n` processes in `group` with ranks `ranks[0], ..., ranks[n-1]`. If `n = 0`, then `newgroup = MPI_GROUP_EMPTY`.
  - `int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)` Returns in `group` a handle to the group of `comm`.

WITS UNIVERSITY
    ```

Group Management cont.

- Group Constructors
  - ```
    int MPI_Group_excl(
        MPI_Group group,
        int n,
        const int ranks[],
        MPI_Group *newgroup)
    ```

    This function creates a group of processes `newgroup` that is obtained by deleting from `group` those processes with ranks `ranks[0], . . ., ranks[n-1]`.

WITS UNIVERSITY
    ```

Group Management cont.

More group constructing functions:

- ```
  int MPI_Group_union(
      MPI_Group group1,
      MPI_Group group2,
      MPI_Group* newgroup)
  ```
- ```
  int MPI_Group_intersection(
      MPI_Group group1,
      MPI_Group group2,
      MPI_Group* newgroup)
  ```
- ```
  int MPI_Group_difference(
      MPI_Group group1,
      MPI_Group group2,
      MPI_Group* newgroup)
  ```

WITS UNIVERSITY
  ```

Group Management cont.

Set operations in group construction

- Union: Returns in `newgroup` a group consisting of all processes in `group1` followed by all processes in `group2`, with no duplication
- Intersection: Returns in `newgroup` all processes that are in both groups, ordered as in `group1`
- Difference: Returns in `newgroup` all processes in `group1` that are not in `group2`, ordered as in `group1`

WITS UNIVERSITY

Group Management cont.

- Group Destructors

```
int MPI_Group_free(MPI_Group *group)
```

Communicator Constructors

- Communicator constructors
  - ```
    int MPI_Comm_create(
        MPI_Comm comm,
        MPI_Group group,
        MPI_Comm *newcomm)
    ```
  - Collective routine within the communicator `comm`
  - Creates a new communicator which is associated with `group`
  - `MPI_COMM_NULL` is returned to processes not in `group`
  - All `group` arguments must be the same on all calling processes
  - `group` must be a subset of the group associated with `comm`.

WITS UNIVERSITY
    ```

MPI_Comm_create() Example

Consider dividing the processes in the `MPI_COMM_WORLD` into two groups and create a new communicator for each group.

MPI_Comm_create() Example

```c
#define NPROCS 8

int rank, new_rank, sendbuf, recvbuf, comm_sz;
int ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
MPI_Group orig_group, new_group;
MPI_Comm new_comm;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

sendbuf = rank;
/* Extract the original group handle */
MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
```

MPI_Comm_create() Example

```
/* Divide tasks into two distinct groups based upon rank */
if (rank < NPROCS/2)
  MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
else
  MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);

/* Create new communicator and then perform collective communications
     */
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);

MPI_Group_rank(new_group, &new_rank);
printf("rank = %d new_rank = %d recvbuf = %d\n", rank, new_rank,
     recvbuf);
```

MPI_Comm_free()

- Communicator Destructor

```
int MPI_Comm_free(MPI_Comm *comm)
```

  When you have finished using a communicator, free (delete/destroy) it.

Outline

WITS UNIVERSITY

Communicator Management

- Communicator accessors
  - `int MPI_Comm_size(MPI_Comm comm, int *size)`
  - `int MPI_Comm_rank(MPI_Comm comm, int *rank)`

Communicator Constructors

```
MPI_Comm_dup(
    MPI_Comm oldcomm,
    MPI_Comm *newcomm)
```

Creates a new communicator that is an exact replica of an existing communicator.

MPI_Comm_split()

- Communicator constructors
  - `int MPI_Comm_split(`
    ```
    MPI_Comm comm,
    int color,
    int key,
    MPI_Comm *newcomm)
    ```
    - Partitions the group associated with the given communicator into disjoint subgroups.
    - `color` controls the subset assignment,
    - `key` controls the rank assignment.

WITS UNIVERSITY

MPI_Comm_split() cont..

- A collective operation.
- All the processes that pass in the same value of `color` will be placed in the same communicator, and that communicator will be the one returned to them.
- The `key` argument is used to assign ranks to the processes in the new communicator.
  - If all processes passing the same `color` value also pass the same `key` value, the order of the ranks in the new communicator will be the same as in the old one. Note that `color` ≥ 0.
  - If they pass in different values for `key`, then these values are used to determine their order in the new communicator.
  - For simplicity, `key = 0` — you don't care about the order.
  - `MPI_UNDEFINED` is used as the `color` for processes not to be included in any of the new groups.

WITS UNIVERSITY

MPI_Comm_split() cont..

- MPI_Comm_split creates several new communicators but each process is given access only to one of the new communicators.

Rank	0	1	2	3	4	5	6	7	8	9	10	11
Process	a	b	c	d	e	f	g	h	i	j	k	l
Color	U	3	1	1	3	2	3	3	1	2	U	2
Key	0	1	2	3	1	9	3	8	1	0	0	0

Both process a and k are returned MPI_COMM_NULL. 3 new groups are created: {b, e, g, h}, {c, d, i}, {f, j, l}.

WITS UNIVERSITY

MPI_Comm_split() Example.

Example 6

Suppose we create NROW $\times$ NCOL number of processes. We want to form a communicator for the processes in each row, and do some computation using each communicator. How? Similarly, we can also formulate a communicator for the processes in each column.

Example 6 cont.

```
#define NROW 3
#define NCOL 4

int irow, icol, color, key, rank_in_world;
MPI_Comm row_comm, col_comm;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank_in_world);

irow = rank_in_world%NROW;
icol = rank_in_world/NROW;
// Build row communicators
color = irow;
key = rank_in_world;
MPI_Comm_split(MPI_COMM_WORLD, color, key, &row_comm);
// Build column communicators
color = icol;
MPI_Comm_split(MPI_COMM_WORLD, color, key, &col_comm);
```

WITS UNIVERSITY

Example 6 cont.

```
int row_procs[NCOL], col_procs[NROW];
int max_rank_row, max_rank_col, my_max;
my_max = rank_in_world;

MPI_Allgather(&my_max,1,MPI_INT,row_procs,1,MPI_INT,row_comm);
MPI_Allgather(&my_max,1,MPI_INT,col_procs,1,MPI_INT,col_comm);

max_rank_row = row_procs[0];
for(int i = 1; i < NCOL; i++)
  if(row_procs[i] > max_rank_row) max_rank_row = row_procs[i];
max_rank_col = col_procs[0];
for(int i = 1; i < NROW; i++)
  if(col_procs[i] > max_rank_col) max_rank_col = col_procs[i];
```

WITS UNIVERSITY

Creating a Communicator

- For example, in matrix-vector or matrix-matrix multiplications, we can take tiled matrix decomposition approach.
- In such a case, we want to create a virtual mesh of processes – a Cartesian topology.
- Function `MPI_Dims_create` returns an array of integers specifying the number of nodes in each dimension of the grid.
- Its syntax:

```
int MPI_Dims_create(int nodes, int dims,
                               int* size)
```

  - `nodes`: input specifies the number of nodes in the grid;
  - `dims`: input specifies the number of dimensions in the grid;
  - `size`: both input and output for the size of each dimension; input: if `size[i]=0` (`i = 0, 1, ..., dims-1`), the function is free to decide the sizes of the grid dimensions; if `size[i]>0`, then the size of the dimension is determined by the value of `size[i]`. output: after the function call returns, `size` contains the sizes of grid dimensions.

WITS UNIVERSITY

Creating a Communicator cont.

For example,

```
int p = 16;
int size[2];
......
size[0]=size[1]=0;
MPI_Dims_create(p,2,size);
```

Creating a Communicator cont.

- After determining the process grid size, we can create a communicator with this topology using function `MPI_Cart_create`;
- Its syntax:

```
int MPI_Cart_create(MPI_Comm old_comm, int dims,
                    int *size, int *periodic,
                    int reorder, MPI_Comm *cart_comm)
```

  - `old_comm`: the old communicator
  - `dims`: the number of grid dimensions
  - `*size`: an array of sizes for each grid dimension
  - `*periodic`: an array of size `dims`. `periodic[j]` is 1 if dimension `j` is periodic and 0 otherwise. Periodic communication wraps around the edges of the grid.
  - `reorder`: it is a flag parameter. If 0, the rank order of processes in the new communicator is the same as in the old communicator. If it is 1, the rank can be reordered in the new communicator.

WITS UNIVERSITY

Creating a Communicator cont.

```c
MPI_Comm cart_comm;
int p;
int size[2], periodic[2];
...
size[0] = size[1] = 0;
MPI_Dims_create(p,2,size);
periodic[0] = periodic[1] = 0;
MPI_Cart_create(MPI_COMM_WORLD, 2, size, periodic, 1, &cart_comm)
```

Creating a Communicator cont.

- In a Cartesian topology grid communicator, we need to know the ranks of the processes according to their Cartesian coordinates.
- Function

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

when called, returns the rank of a process with `coord` in the Cartesian communicator `comm`.

WITS UNIVERSITY

Creating a Communicator cont.

For example, suppose the Cartesian grid has *r* rows, the data matrix has *m* rows, and Row *i* of input matrix is mapped to the Row *k* of the process grid. Then we can find out the rank of a process according to its coordinate in the virtual grid.

```
int dest_coords[2];
int dest_id, grid_id, i;
...
for(i=0; i<m; i++) {
  k = BLOCK_OWNER(i,r,m);
  dest_coord[0] = k;
  dest_coord[1] = 0;
  MPI_Cart_rank(Cart_comm, dest_coord, &dest_id);
   if(grid_id == 0){
     /* Read matrix row 'i' */
     ...
     /* Send matrix row 'i' to process 'dest_id' */
     ...
  } else if(grid_id == dest_id){
     /* Receive matrix row 'i' from process 0 */
     ...
  }
}
```

Creating a Communicator cont.

- Similarly, a process can determine its coordinate in the virtual grid using its rank.
- Function

```
int MPI_Cart_coords(MPI_Comm comm, int rank,
                        int dims, int *coords)
```

  - `comm`: Cartesian communicator being examined;
  - `rank`: the rank of the process whose coordinate we seek;
  - `dims`: the number of dimensions in the virtual process grid;
  - `coords`: holds the returned coordinate of the process.

References

- Using MPI-1: Portable Parallel Programming with the Message Passing Interface, William Gropp, Ewing Lusk, and Anthony Skjellum. MIT Press Cambridge, London, England.
- A. Grama, A. Gupta, G. Karypis and V. Kumar, Introduction to Parallel Computing, 2nd Edition, Chapter 6.
- `https://hpc-tutorials.llnl.gov/mpi/`
- MPI: The Complete Reference (`https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report/mpi31-report.htm`)
- Parallel Programming in C with MPI and OpenMP, Michael J. Quinn, McGraw-Hill Education Group, 2003.