8. Parallel programming with MPI - III

[1]

[1] M. Ganesh, MATH440/540, SPRING 2018

[2] Further MPI details and commands

- **Suppose that we are interested in, say, the `master` to receive a data sent from a source processing core, say, `from`, with tag, say, `my_tag` and data type, say `MPI_DOUBLE_PRECISION` or `MPI_DOUBLE`**
- **However, for example, the `master` does not know the size of the data to be received**
- **In such cases, we may probe the source first to get the `status` of the data with `my_tag` using `MPI_Probe` and then use the `status` to get the size of the data, say, `count`, using the command `MPI_Get_count`**
- **Using `count`, we may ask the `master` to allocate an array of the size `count` and then receive the data, for example, using `MPI_Recv`:**

```
CALL MPI_Probe(from, my_tag, MPI_COMM_WORLD, status, ierror)
CALL MPI_Get_count(status, MPI_DOUBLE_PRECISION, count, ierror)
MPI_Probe(from, my_tag, MPI_COMM_WORLD, &status);
MPI_Get_count(&status, MPI_DOUBLE, count);
MPI::COMM_WORLD.Probe(from, my_tag, status);
MPI::COMM_WORLD.Get_count(status, MPI::DOUBLE, count);
```

[2] M. Ganesh, MATH440/540, SPRING 2018

Example (Fortran):

```
.....................
.....................
.....................
REAL(KIND=KIND(0.0D0)), ALLOCATABLE, DIMENSION(:) :: my_buf
INTEGER :: ierror , count,  status ( MPI_STATUS_SIZE)
INTEGER , PARAMETER :: from = 5 , my_tag = 1234
.....................
.....................
.....................
CALL MPI_Probe ( from , my_tag , MPI_COMM_WORLD , status , ierror )
CALL MPI_Get_count ( status , MPI_DOUBLE_PRECISION , count , ierror)
ALLOCATE (my_buf( count ) )
CALL MPI_Recv (my_buf , count , MPI_DOUBLE_PRECISION ,from , my_tag , &
 MPI_COMM_WORLD , status , error )
.....................
.....................
.....................
```

Example (C):

```
.....................
.....................
.....................
double * my_buf ;
int from = 5 , my_tag = 1234 , ierror , count ;
MPI_Status status ;
.....................
.....................
.....................
ierror = MPI_Probe ( from , tag ,
       MPI_COMM_WORLD , & status ) ;
ierror = MPI_Get_count ( & status ,
       MPI_DOUBLE_PRECISION , count ) ;
my_buf = malloc ( sizeof ( double ) * count ) ;
.....................
ierror = MPI_Recv ( my_buf , count , MPI_DOUBLE ,
       from , tag , MPI_COMM_WORLD , & status ) ;
.....................
.....................
```

Example (C++):

```
.....................
.....................
.....................
double * my_buf ;
int from = 5 , my_tag = 1234 , count ;
MPI_Status status ;
.....................
.....................
MPI::COMM_WORLD.Probe ( from , my_tag , status ) ;
MPI::COMM_WORLD.Get_count ( status ,
     MPI::DOUBLE , count ) ;
my_buf = new double [ count ] ;
MPI::COMM_WORLD.Recv ( buffer , count ,
     MPI::DOUBLE , from , my_tag , status ) ;

.....................
.....................
.....................
```

Exercise 1:

Generalize your program in Topic 7, Exercise 3 so that the each processing core computes the square root of each element in `recv_array` and sends the `recv_array` to `master`. Then let the `master` : (i) probe the size of `recv_array` from the second processing core; (ii) allocate an array `got_from_second_core` with size same as that of `recv_array`; and (iii) receive `recv_array` in `got_from_second_core`.

Your program be such that the `master` should print
`my_mat`, and `got_from_second_core`
and the second processing core prints its rank and `recv_array`.

- [3]`MPI_Probe` is a blocking call and hence leads to waiting until the matching source and tag in a chosen communicator arrives
- If you would like to probe without blocking, use the asynchronous call `MPI_Iprobe`
- `MPI_Iprobe` returns immediately even if no matching is found
- If there is a matching, `MPI_Iprobe` behaves like `MPI_Probe` and if there no matching, then the `status` is not updated
- Additional argument, say, `flag`, in `MPI_Iprobe` allows one to check if `status` is updated or not
- `flag` is of the type `LOGICAL` in Fortran and `int` in C and C++:

```
CALL MPI_Iprobe(from, my_tag, MPI_COMM_WORLD, flag, status, ierror)
MPI_Iprobe(from, my_tag, MPI_COMM_WORLD, &flag, &status);
flag = MPI::COMM_WORLD.Iprobe(from, my_tag, status);
```

Exercise 2:
Repeat Exercise 1 with `MPI_Iprobe` instead with appropriate twists of your choice. (For example probe twice, once before sending and once after sending.)

[3]M. Ganesh, MATH440/540, SPRING 2018

- [4]So far, we focused on using specific sources (such as the second processing core etc.) and specific tag (such as `my_tag`)
- Data can be received from any source with any tag, without knowing their specific details.
  (Be careful here; especially avoid wild tag unless ....)
- In such cases, the intrinsic variables `MPI_ANY_SOURCE` and `MPI_ANY_TAG` can be used to wild card source and tag in all MPI commands that require source and tag arguments

- The `status` argument in `MPI_Recv` allows one to get the source and tag information, after receiving
- The actual source of a data in `status` is stored using the intrinsic name `MPI_SOURCE` and that of data in `MPI_TAG`
- To get source information for a received data with `status`, in Fortran use `status(MPI_SOURCE)` and C/C++ use `status.MPI_SOURCE`
- To get tag information for a received data with `status`, in Fortran use `status(MPI_TAG)` and C/C++ use `status.MPI_TAG`

[4]M. Ganesh, MATH440/540, SPRING 2018

Exercise 2:

Let P be the size of `MPI_COMM_WORLD`.

Generalize your program in Topic 7, Exercise 3 as follows:

The program generates two random numbers in $[0, 1)$. Then transplants the first random number to get a random integer in the range $[0, P - 1]$ for a random source (processing core) and the second to a random integer in the range [123, 1234] for a random tag.

The random processing core then divides each element in `recv_array` by the random tag number and sends the resulting array to `master` with the random tag.

Then let the `master` receive the data sent from the random source using `MPI_ANY_SOURCE` and `MPI_ANY_TAG` and then identifies the random source and tag numbers.

Your program be such that (i) the `master` should print the identified random source and tag numbers and the received data; and (ii) the random source should print the sent data and the tag number it used.

Execute your code using several nodes on MIO/BlueM and Sayers Lab machines and at various dates and times and validate your code by checking the output.

- [5]**Recall the point-to-point** `MPI_Send` **and** `MPI_Recv` **commands introduced in Topic 7**
- **The argument** `status` **in** `MPI_Recv` **facilitates extracting some information of the received data, for example, as in Exercise 1**
- **Mpi-2.* does allow the argument** `status` **in** `MPI_Recv` **to be optional. However, use the** `status` **to facilitate for all parallel environments**
- **Similar to** `MPI_Probe`**, the command** `MPI_Recv` **will block until a matching** `send` **arrives (Otherwise will hang - try this, with your code ..)**

- **However,** `MPI_Send` ***may*** **block until a matching** `receive` **found or it *may* copy the message and return with the assumption that MPI will transfer it later. (This part is left open and it is up to the implementation of the MPI.)**
- **For advanced programming, one may instead use block (synchronous) send, defined by the command** `MPI_Ssend` **or buffered send** `MPI_Bsend`
- **Usage of** `MPI_Ssend` **and** `MPI_Bsend` **are exactly same as** `MPI_Send`

[5]M. Ganesh, MATH440/540, SPRING 2018

- [6]**MPI uses the queue FIFO (First In First Out) and hence incoming data never overtake each other**

- **In case of a `not-so-easy-to-explain` send failure, you may use buffered sends to see if the problem gets fixed. (If yes, return back to the code and fix the send problem, especially some race condition.)**
- **For buffered sends, it is important that each processing core has a bigger buffer to avoid the buffer becoming full during a send**
- **The default buffer size in MPI is not defined and is MPI implementation dependent**
- **Accordingly, for `MPI_BSEND` (and `MPI_IBSEND`, see below for `MPI_*`), you need to attach a buffer to a processing core before using `MPI_BSEND` and then detach the buffer when not required:**

- **If you are further interested in this buffered process (to optimize the buffer size etc.), you may check on additional commands `MPI_BSEND_OVERHEAD`, `MPI_Pack_size`, `MPI_Sizeof` (only in Fortran)**

[6]M. Ganesh, MATH440/540, SPRING 2018

Example (Fortran):

```
.....................
.....................
.....................
INTEGER , PARAMETER :: big_siz = 10000
CHARACTER :: lar_buf ( big_size )
INTEGER :: my_size , ierror
.....................
.....................
CALL MPI_Buffer_attach (lar_buf , big_siz , ierror )
.....................
.....................
.....................
CALL MPI_Bsend(my_buf, ................, ierror)
.....................
.....................
my_siz = big_size
CALL MPI_Buffer_detach (lar_buf , my_siz , ierror )
.....................
.....................
```

Example (C):

```
.....................
.....................
.....................
#define   big_siz  10000
char lar_buf [ big_siz ] , * ptr_buf ;
int my_siz , ierror ;
.....................
.....................
ierror = MPI_Buffer_attach (lar_buf , big_siz) ;
.....................
.....................
MPI_Bsend(my_buf, ...................) ;
.....................
.....................
ptr_buf = lar_buf
my_siz = big_size
ierror = MPI_Buffer_detach (& ptr_buf ,& my_siz) ;
.....................
.....................
```

Example (C++):

```
.....................
.....................
.....................
#define   big_siz  10000
char lar_buf [ big_siz ] , * ptr_buf ;
int  ierror ;
.....................
.....................
MPI::Attach_buffer (lar_buf , big_siz) ;
.....................
.....................
MPI::COMM_WORLD.Bsend(my_buf, ..................) ;
.....................
.....................
ptr_buf = lar_buf

MPI::Detach_buffer (ptr_buf) ;
.....................
.....................
```

Exercise 3:
Consider your codes to solve exercises in Topic 6 and 7 that use point-to-point or collective (scatter) sends.

Take a copy of each of these codes.

Replace the non-buffered sends in these codes with buffered sends.

Use probing to find the size of each data, allocate appropriate arrays before receiving these using synchronized `MPI_Recv`.

Exercise 4:
Consider your codes to solve exercises in Topic 6 and 7 that use point-to-point sends and receives.

Take a copy of each of these codes.

Replace the non-buffered sends in these codes with a buffered composite send-receive and also with a buffered composite send-receive-replace (as discussed below).

- [7]**Recall MPI_Send and MPI_Recv commands, for example in Fortran, from Topic 6:**

```
CALL MPI_Send(send_buffer, send_count, MPI_DOUBLE_PRECISION, &
                    to, to_tag, MPI_COMM_WORLD, ierror)
CALL MPI_Recv(recv_buffer, recv_count, MPI_DOUBLE_PRECISION, &
                  from, from_tag, MPI_COMM_WORLD, status, ierror)
```

- **Some deadlock in separate send and receive may be avoided by using a composite blocking send and receive operations (by combining the above two commands into a single command)**

- **In addition to avoiding the deadlock, the composite send and receive matches the separate send and receive. This will do in the correct order (that is send first, receive next)**

- **The composite send and receive command also has a replace form. (This involves extra work of copying etc.)**

[7]M. Ganesh, MATH440/540, SPRING 2018

```
CALL MPI_Sendrecv(send_buffer, send_count, MPI_DOUBLE_PRECISION, to, &
        to_tag, recv_buffer, recv_count, MPI_DOUBLE_PRECISION, &
                 from, from_tag, MPI_COMM_WORLD, status, ierror)

CALL MPI_Sendrecv_replace(buffer, count, MPI_DOUBLE_PRECISION, to, &
                to_tag, from, from_tag, MPI_COMM_WORLD, status, ierror)


MPI_Sendrecv(send_buffer, send_count, MPI_DOUBLE, to, to_tag, recv_buffer,
         recv_count, MPI_DOUBLE, from, from_tag, MPI_COMM_WORLD, & status)

MPI_Sendrecv_replace(buffer, count, MPI_DOUBLE, to, to_tag,
                 from, from_tag, MPI_COMM_WORLD, & status)


MPI::COMM_WORLD.Sendrecv(send_buffer, send_count, MPI::DOUBLE, to, to_tag,
         recv_buffer, recv_count, MPI::DOUBLE, from, from_tag, status)

MPI::COMM_WORLD.Sendrecv_replace(buffer, count, MPI::DOUBLE, to, to_tag,
                 from, from_tag, status)
```

- [8]**Next we consider non-blocking (asynchronous) transfers**
- **Each main asynchronous transfer calls returns a handle, a `request`**
- **The `request` is an integer in Fortran and is a structure in C/C++**
- **In C and C++, use `MPI_request request` and in Fortran declare `request` as an integer for the handle `request`**
- **Below, for asynchronous send and receive (`MPI_Isend` and `MPI_Irecv`) we assume that the declaration `request` part is already in your code**
- **The asynchronous `MPI_Isend` and `MPI _Irecv` calls are returned immediately (similar to `MPI_Iprobe`) and return a `request`**
- **Unlike `MPI_Recv`, the `MPI_Irecv` does not return the `status`**
- **We may use `request` in `MPI_Wait` to block until the transfer is finished and get the `status`**
- **We may instead use `request` in `MPI_Test` without blocking, to check whether the requested transfer is complete by getting both the `status` and a `flag` with value indicating whether the transfer is finished**
- **`flag` is of the type `LOGICAL` in Fortran and `int` in C and C++:**

[8]M. Ganesh, MATH440/540, SPRING 2018

```
CALL MPI_Isend(buffer, count, MPI_DOUBLE_PRECISION, to, tag, &
               MPI_COMM_WORLD, request, ierror)
CALL MPI_Irecv(buffer, count, MPI_DOUBLE_PRECISION, from, tag, &
               MPI_COMM_WORLD, request, ierror)
CALL MPI_Wait (request,status,ierror)
CALL MPI_Test (request,flag, status,ierror)
```

```
MPI_Isend(buffer, count, MPI_DOUBLE, to, tag, MPI_COMM_WORLD, & request)
MPI_Irecv(buffer, count, MPI_DOUBLE, from, tag, MPI_COMM_WORLD, & request)
MPI_Wait (& request , & status)
MPI_Test (& request, & flag ,& status)
```

```
request = MPI::COMM_WORLD.Isend(buffer, count, MPI::DOUBLE, to, tag)
request = MPI::COMM_WORLD.Irecv(buffer, count, MPI::DOUBLE, from, tag)
request.Wait(status)
flag = request.Test(status)
```

[9]**Exercise 5:**

Take a copy of your code, implementing the the random source problem. Replace the synchronous transfers in the code with non-blocking (asynchronous) `MPI_Isend` and `MPI_Irecv`. Check the status of the transfer using non-blocking `MPI_Test` and blocking `MPI_Wait`.

- **We may test or wait for an array of requests**
- **The commands `MPI_Testall` and `MPI_Waitall` check for or complete all the requests and return an array of statuses**
- **The commands `MPI_Testany` and `MPI_Waitany` check for or complete only one request and return its index and status.**
  (The index is set to `MPI_UNDEFINED` if none of requests are active.)

- **Empty status is allowed in MPI.**
  The MPI empty status may have tag value `MPI_ANY_TAG`,
  source value `MPI_ANY_SOURCE` (or `MPI_PROC_NULL`),
  action of `MPI_GET_COUNT` on it returns the value zero etc.

[9]M. Ganesh, MATH440/540, SPRING 2018

Example (Fortran):

```
.....................
.....................
INTEGER :: i , ierror , requests(1000),   statuses(MPI_STATUS_SIZE, 1000)
.....................
.....................
DO i = 1 , 100
CALL MPI_Irecv ( . . . , MPI_COMM_WORLD , requests(i) , error )
END DO
CALL MPI_Waitall (1000, requests , statuses , ierror )

INTEGER :: ierror , index , requests(100 ) ,  status(MPI_STATUS_SIZE )
.....................
.....................
DO
    CALL MPI_Waitany ( 100 , requests , index , status , error )
    IF ( index /= MPI_UNDEFINED ) THEN
          CALL operate ( index , status )
    END IF
END DO
```

Example (C):

```
int i , ierror , requests [ 1000 ] ;
MPI_Status statuses [ 1000 ] ;
.....................
for ( i = 1 ; i < 1000 ; ++ i )
       ierror = MPI_Irecv ( . . . ,  MPI_COMM_WORLD , requests(i) ) ;
ierror = MPI_Waitall ( 1000 , requests , statuses ) ;
.....................
.....................
int ierror , index, requests [ 100 ] ;
MPI_Status status  ;
.....................
.....................
while ( 1 ) {
     ierror = MPI_Waitany ( 100 , requests , index , status ) ;
     if ( index == MPI_UNDEFINED )
            break ;
     else
            operate ( index , status ) ; }
```

Example (C++):

```
int i ,  requests [ 1000 ] ;
MPI_Status statuses [ 1000 ] ;
.....................
for ( i = 1 ; i < 100 ; ++ i )
      MPI::COMM_WORLD.Irecv ( . . . ,  requests(i) ) ;
 request[0].Waitall ( 1000 , requests , statuses ) ;
.....................
.....................
int index requests [ 100 ] ;
MPI_Status status  ;
.....................
.....................
while ( 1 ) {
     request[0].Waitany ( 100 , requests , index , status ) ;
     if ( index == MPI_UNDEFINED )
            break ;
     else
            operate ( index , status ) ; }
```

- **[10]Suppose that in a communicator consisting of P processing cores, the i-th core has rank, say, `index_i` and has a variable with value `value_i` for $i = 0, \cdots, P-1$**
- **Suppose that we want the MPI to search all these pairs `(value_i, index_i)`, $i = 0. \cdots, P-1$, in the communicator and let, for example, the `master` to receive a pair `(val, ind)`, where for example,**

$$val = \max\{value_i : i = 0, \cdots, P-1\}$$

  and `ind` is the index of the processing core with `val`
- **If, for example, two distinct processing cores have `val`, then the minimum number of index in these two cores is used for `ind`**
- **The `MPI_Reduce` and `MPI_MAXLOC` operations are useful for the above task. (Use `MPI_MINLOC` to find minimum among the pairs.)**
- **The `MPI_Reduce` reduce operation is defined to operate on arguments that consist of the pair `send_pair = (value_i, index_i)`**
- **Hence we need to provide the MPI datatype for the pair. To this end, we need to use derived datatypes (to be discussed later in detail)**

[10]M. Ganesh, MATH440/540, SPRING 2018

- [11]**The MPI datatype used for searching pairs for programming with MPI in Fortran is different from that in C/C++**

- **Useful MPI datatype in Fortran for the pair search are:**
  - ⋆ `MPI_2DOUBLE_PRECISION` **for a pair of DOUBLE PRECISIONs**
  - ⋆ `MPI_2INTEGER` **for a pair of INTEGERs**
  - ⋆ `MPI_2REAL` **for a pair of REALs (avoid)**
- **Useful MPI datatype in C/C++ for the pair search are:**
  - ⋆ `MPI_DOUBLE_INT` **for a pair of double and int**
  - ⋆ `MPI_LONG_DOUBLE_INT` **for a pair of long double and int**
  - ⋆ `MPI_LONG_INT` **for a pair of long and int**
  - ⋆ `MPI_2INT` **for a pair of int**
  - ⋆ `MPI_FLOAT_INT` **for a pair of float and int (avoid)**
  - ⋆ `MPI_SHORT_INT` **for a pair of short and int (avoid)**

[11]M. Ganesh, MATH440/540, SPRING 2018

[12]**Example**:

```
CALL MPI_Reduce(inpair, outpair, count, &
  MPI_2DOUBLE_PRECISION, MPI_MAXLOC, master, MPI_COMM_WORLD, ierror)

MPI_Reduce(&inpair,&outpair, count,
                MPI_DOUBLE_INT, MPI_MAXLOC, master, MPI_COMM_WORLD);

MPI::COMM_WORLD.Reduce(&inpair,&outpair, count,
                       MPI::DOUBLE_INT, MPI::MAXLOC, master);
```

Exercise 6:

Take a copy of your code for solving Exercise 3 in Topic 7.

Generalize the code so that the `master` deallocates `my_mat`. Then using two `MPI_Reduce` commands, receives the maximum and minimum values in `my_mat` and also the processing cores containing these values.

Your program be such that the `master` should print these received values. Check your answer with the original matrix.

[12]M. Ganesh, MATH440/540, SPRING 2018