9. Parallel programming with MPI - IV

[1]

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

[2] More MPI details and commands

- **Recall the `MPI_Scatter` command:**

```
MPI_Scatter( buffer, send_count, MPI_DOUBLE, recv_buffer, recv_count,
MPI_DOUBLE, master, MPI_COMM_WORLD)
```

- **For gather, we replace `MPI_Scatter` with `MPI_Gather`. (The send buffer is larger than the receive buffer for scatter and vice-versa for gather.) Similarly, recall other collectives:**
  `MPI_Allgather` and `MPI_Alltoall`
- **For all these commands, the `count` variable is a scalar (constant) and hence uniform distribution of the buffer is required**
- **For non-uniform distribution of a buffer/data with any one of the the above MPI collective commands, use**
  `MPI_Scatterv` or `MPI_Gatherv` or `MPI_Allgatherv`, or `MPI_Alltoallv`:
- **For non-uniform distribution of data in a communicator with size `num_cores`, we need a `num_cores` dimensional vector of counts. say, `count(0:num_cores-1)` so that there is one scalar-count for each processing core**

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

- [3]**The `num_cores` dimensional vector `count`**
  - ⋆ **in `MPI_Scatterv` is for the send buffer**
  - ⋆ **in `MPI_Gatherv` and `MPI_Allgatherv` for the receive buffer**
  - ⋆ **and for both the buffers in `MPI_Alltoallv`**
- **Each component of the vector `count` may be different but the scalar-count must match for the pairwise send and receive**
- **For example, in `MPI_Gatherv` the send scalar-count on the i-th processing core should match the i-th component of the receive vector `count` on the gathering processor, usually the `master`. (For `MPI_scatterv` the converse should hold.)**
- **In `MPI_Allgatherv`, the send count on the i-th processing core should match the i-th component of the receive vector `count` on all processing cores, for `i = 0, ..., num_cores`**
- **In `MPI_Alltoallv` (think of the matrix transpose problem discussed earlier), the j-th component of the send count vector `send_count` on the i-th processing core should match the j-th component of the receive count vector `recv_count` on the i-th processing core**

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

- [4]**In practice, for scatter it is convenient to first use `MPI_scatter` to equally scatter the `count` vector and then use `MPI_scatterv` on the buffer from the `master` to distribute non-uniform size data**
- **Similarly, for gather it is convenient to first use `MPI_gather` to gather all components of the `count` vector in `master` and then use `MPI_gatherv` from the `master` to gather non-uniform size data**
- **Corresponding to the vector `count`, the non-uniform transfers require a vector of offsets, henceforth `offset_vec`**
- **For `MPI_Alltoallv`, corresponding to two vectors `send_count` and `recv_count`, we need two offset vectors `send_offset_vec` and `recv_offset_vec`**
- **Each pairwise transfer must be contiguous and `offset_vec` facilitates non-uniform distribution of non-contiguous buffers**
- **Usually the first element of the `offset_vec` is zero (not necessary)**
- **Each offset element is of the data of each process and is not the offset of the basic elements**
- **The offsets are local and unlike counts, offset need not match on all processing cores. (`offset_vec` is used only for mapping layout.)**

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

Example

```
.....................
.....................
.....................
IF(my_rank == master)THEN
        ALLOCATE(count_vec(0:num_cores-1))
        ALLOCATE(offset_vec(0:num_cores-1))
END IF
.....................
.....................
.....................

loc_scalar_count  = my_rank+3

ALLOCATE(send_buf(loc_scalar_count))

send_buf = ........


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

```
CALL MPI_Gather(loc_scalar_count,1,MPI_INTEGER, count_vec,
                  1,MPI_INTEGER,  master, MPI_COMM_WORLD, ierror)

IF(my_rank == master)THEN

   offset_vec(0) = 0

   DO i=1,num_cores-1
         offset_vec(i)=count_vec(i-1)+offset_vec(i-1)
   END DO
   full_siz = sum(count_vec)
   ALLOCATE(recv_buf(full_siz))
END IF
.....................
CALL MPI_Gatherv(send_buf, count_vec(my_rank ), MPI_DOUBLE_PRECISION,
              recv_buf,  count_vec, offset_vec, MPI_DOUBLE_PRECISION,
              master , MPI_COMM_WORLD, ierror )
.....................
```

```
MPI_Gatherv(send_buf,loc_count,offset_vec,MPI_DOUBLE,
    recv_buf,count_vec,MPI_DOUBLE,master,MPI_COMM_WORLD);
MPI::COMM_WORLD.Gatherv(send_buf,loc_count,offset_vec,MPI::DOUBLE,
    recv_buf,count_vec,MPI::DOUBLE,master);
```

```
CALL MPI_Scatterv(send_buf,count_vec,offset_vec,MPI_DOUBLE_PRECISION, &
    recv_buf,loc_count,MPI_DOUBLE_PRECISION,master,MPI_COMM_WORLD,ierror)
MPI_Scatterv(send_buf,count_vec,offset_vec,MPI_DOUBLE,
    recv_buf,loc_count,MPI_DOUBLE,master,MPI_COMM_WORLD);
MPI::COMM_WORLD.Scatterv(send_buf,count_vec,offset_vec,MPI::DOUBLE,
    recv_buf,loc_count,MPI::DOUBLE,master);
```

```
CALL MPI_Allgatherv(send_buf, loc_count, MPI_DOUBLE_PRECISION, recv_buf,
 count_vec, offset_vec, MPI_DOUBLE_PRECISION, MPI_COMM_WORLD, ierror )

MPI_Allgatherv(send_buf,loc_count,offset_vec,MPI_DOUBLE,
    recv_buf,count_vec,MPI_DOUBLE,MPI_COMM_WORLD);

MPI::COMM_WORLD.Allgatherv(send_buf,loc_count,offset_vec,MPI::DOUBLE,
    recv_buf,count_vec,MPI::DOUBLE);

CALL MPI_Alltoallv(send_buf, send_count_vec, send_offset_vec, &
 MPI_DOUBLE_PRECISION, recv_buf,recv_count_vec, recv_offset_vec, &
 MPI_DOUBLE_PRECISION, MPI_COMM_WORLD, ierror )

MPI_Alltoallv(send_buf, send_count_vec, send_offset_vec, MPI_DOUBLE,
 recv_buf,recv_count_vec, recv_offset_vec, MPI_DOUBLE, MPI_COMM_WORLD);


MPI::COMM_WORLD.Alltoallv(send_buf, send_count_vec, send_offset_vec,
 MPI::DOUBLE, recv_buf,recv_count_vec, recv_offset_vec, MPI::DOUBLE);
```

[5]**Exercise 1:**

Create a file with some data containing, say, $N = 100$ numbers. Write a parallel (Fortran, C, or C++) program with MPI so that the `master` core reads the data from the file in a 10×10 matrix, say, `my_mat`.

Let `num_cores` be the total number of processing core in a communicator, with `master = 0` being the first processing core in the communicator.

The `master` core should scatter `my_mat` so that:

- **the first processing core receives the first two columns of `my_mat`;**
- **the second processing core receives the next three columns of `my_mat`;**
- **the third processing core receives the next four columns of `my_mat`;**
- **the fourth processing core receives the last column of `my_mat`.**

Each processing core should allocate and receive these elements in an array `recv_array` of appropriate dimension.

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

[6]Exercise 1 (continued):

Each processing core should reshape `recv_array` into a matrix `loc_matrix` with five rows and appropriate number of columns. Then each processing core should compute maximum value of each column in `loc_matrix` and store in an array `max_col_array`

The master should gather these unequal sized `max_col_array` from all processing cores in an array `max_all_col_array` of appropriate size.

Your program should check if `num_cores == 4`.
If this condition is violated, the program should abort.
Your program be such that all cores should print `max_col_array` and its rank and the `master` should in addition print `max_all_col_array`.

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

- [7]**So far, in all `MPI_*` commands we used the default and pre-defined communicator `MPI_COMM_WORLD`**

- **In these `MPI_*` commands, we may replace `MPI_COMM_WORLD` with any other MPI communicator of our choice:**

- **The default `MPI_COMM_WORLD` consists of *all* the processing cores induced by the command `mpiexec` or `mpiexec`**

- **Our new communicator may consist only of a *few* selected class of the processing cores induced by the command `mpirun` or `mpiexec`**

- **There are other pre-defined MPI communicators:**
  - ⋆ **`MPI_COMM_SELF` consists of just the local processing core**
  - ⋆ **`MPI_COMM_NULL` - an invalid communicator (useful for checking ...)**

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

- [8]**A communicator is a combination of a group and a context**
- **A group, with `size`, is a set of processing cores identified as integers `0, 1, ..., size-1`**
- **A context is the communication environment. Separate contexts are entirely independent and programs cannot view contexts directly**
- **Thus separate communicators are independent even if they have the same or intersecting group of processing cores**
- **In practice when using the MPI, we work only on communicators**

- **We may subdivide the global communicator `MPI_COMM_WORLD` to create one or more new local communicators**
- **Such local communicators facilitate a selective class of processing cores to work, for example, in collective calls such as `MPI_REDUCE`**
- **Avoid, whenever possible, using two communicators that overlap**

- **One approach to create new local communicators is to split an existing communicator such as `MPI_COMM_WORLD`:**

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

- [9]**Suppose that an existing communicator, say,** `old_comm` **(for example** `MPI_COMM_WORLD`**) consists of** P **processing cores**

- **To split** `old_comm` **into new local communicators, say,** `new_comm`**, we may use the command** `MPI_COMM_split`

- **For this purpose, first declare** `new_comm`**:**
  - `INTEGER :: new_comm` **(in Fortran with MPI)**
  - `MPI_Comm newcomm;` **(in C with MPI)**
  - `MPI::Intracomm newcomm;` **(in C++ with MPI)**

- **The format for** `MPI_COMM_split` **for MPI programming in Fortran, C, and C++ are respectively given by**

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

[10]
```
CALL MPI_Comm_split(old_comm,color_ind,rank_key,new_comm,ierror)
```

```
MPI_Comm_split(old_comm,color_ind,rank_key,& new_comm);
```

```
new_comm = old_comm.Split(color_ind,rank_key);
```

- **Here, `color_ind` is an input integer and all processing cores that pass the same `color_ind` will be placed in the same `new_comm`**

- **The input `rank_key` is an integer**
- **If all processing cores (each with a rank value) that pass the same value of `color_ind` also pass in the same value `rank_key`, then they are given ranks in `new_comm` that are in the same order as their ranks in `old_comm`**

- **If all processing cores (each with a rank value) that pass the same value of `color_ind` pass in the different values `rank_key`, then these values are used to determine their order in `new_comm`**

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

Example:

Recall Exercise 4 in Topic 6. To solve this exercise, all that we need is to take a code for solving Exercise 3 in Topic 6.

Then to compute the global `full_sum` in the `master`, using the `local_sum` from all processing cores in the global communicator MPI_COMM_WORLD with P processing cores, we just need to add one additional collective command:

```
CALL MPI_Reduce(local_sum, full_sum,  1, MPI_DOUBLE_PRECISION, &
         MPI_SUM, master,  MPI_COMM_WORLD, ierror)
```

Let $N \leq P/2$ be an integer.

Suppose that we want to split MPI_COMM_WORLD into N new local communicators, such that for $i = 0, \cdots, P-1$, the i-th processing goes to the $r+1$-th local communicator if $\mod(i, N) = r$.

(Think of, for example, $N = 2$ so that all processing cores with odd rank values go to the first communicator and with even rank values go the second communicator.)

Suppose that we want each local `master` (that is the processing core with rank zero in each local communicator) to sum up all the `local_sum` within its communicator. Write a program with MPI to achieve this.

To solve the above problem, first take a copy of your code for solving Exercise 4 in Topic 6.

Then add the following lines. (Add appropriate declaration statements. Check and make sure that sum of these `sub_full_sum` is equal to the `full_sum` obtained in Topic 6, Exercise 4.)

```
N = 3             !Replace with any N <= num_cores/2
rank_key = 0      !Try replacing with rank_key = my_rank
color_ind = mod(my_rank, N)
call MPI_COMM_SPLIT(MPI_COMM_WORLD, color_ind, 0,  MY_COMM, ierror)
call MPI_COMM_RANK( MY_COMM, my_new_rank, ierror)
call MPI_COMM_SIZE( MY_COMM, my_sub_num_cores, ierror)
call MPI_Reduce(local_sum, sub_full_sum,  1, MPI_DOUBLE_PRECISION, &
                    MPI_SUM, master,  MY_COMM, ierror)
IF (my_new_rank ==  master) THEN
WRITE(*,*)' FULL SUM IN COMMUNICATOR ', color_ind, &
                ' with size = ', my_sub_num_cores, '  = ', sub_full_sum
END IF
WRITE(*,*) 'old_rank = ', my_rank,  'new_rank = ', my_new_rank
CALL MPI_Comm_free(newcomm,ierror)
CALL MPI_Finalize(ierror)
```

- [11]**We may opt out of using certain processing cores in `color` vector (consisting of all `color_ind`) by using `MPI_UNDEFINED`**
- **For example, suppose that an existing communicator, say, `MPI_COMM_WORLD` consists of, say, $P = 8$ processing core and that we want to split `MPI_COMM_WORLD` into two sub communicators so that**
  - ⋆ **`core 0, core3, core 7` NOT in any new local communicator**
  - ⋆ **`core 1, core 4, core 5` are in the first of the local communicator**
  - ⋆ **`core 2, core 6` are in the second local communicator**
- **We can achieve the above in Fortran with MPI using**

```
.....................
.....................
INTEGER, PARAMETER :: first = 007
INTEGER, PARAMETER :: second = 008
INTEGER, PARAMETER :: color(0:7) = (/ MPI_UNDEFINED,first,   &
                 second,MPI_UNDEFINED,first,first,second,MPI_UNDEFINED /)


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

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

```
.....................
CALL MPI_Comm_split(MPI_COMM_WORLD,color(my_rank),0,my_comm,ierror)
.....................
.....................
IF (my_comm /= MPI_COMM_NULL) THEN
        CALL MPI_Reduce(..............)
END IF


.....................
.....................
CALL MPI_Comm_free(MY_COMM,ierror)
```

- [12]**One of the commands (in Fortran, C, and C++)**
  - ⋆ `CALL MPI_Comm_free(new_comm,ierror)`
  - ⋆ `MPI_Comm_free(& new_comm);`
  - ⋆ **new_comm.Free()**

  is required to delete the local communicator `new_comm` **(once we are finished with it) to set free the resources used by** `new_comm`

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

[13]**Exercise 2:**

Take a copy of your code for solving Exercise 6 in Topic 7. Let the number of nodes induced by `mpirun` and `mpiexec` command be N and, as in Exercise 6 in Topic 7, let $P = mN$ be the total size of `MPI_COMM_WORLD` and A_P be the matrix defined in (8.1) and each processing core in `MPI_COMM_WORLD` has one row of A_P. (Choose $m = 4, 12, 16$ respectively for `mpiexec` on Sayers Lab machines, MIO, and Blue-AuN.)

Generalize the code to create N local communicators (by splitting), one for each node so that each local communicator has size m.
Using `MPI_REDUCE` command, each local `master` should create a local vector `loc_col_max` of dimension P with entries being maximum of entries in each of the P columns from the m rows of A_P available within the local communicator.

Each local `master` should print its communicator number, size, and `loc_col_max`. In addition, create another local communicator called `full_master_comm` with each processing cores in the communicator being only the local `master`. Then using `loc_col_max`, the `master` of `full_master_comm` should compute the P dimensional vector with entries being the maximum of entries in `loc_col_max`.

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

- [14]**In case of complex creation of new communicators, for example, communicators obtained by splitting twice a new communicator, use** `MPI_Barrier` **to ensure proper tuning**

- **We may make an exact copy of a communicator**
- **In this case, the context - separate universe - facilitates, the copy communicator, say,** `copy_comm` **to be completely independent of the original communicator, say** `orig_comm`**:**

- ```
  CALL MPI_Comm_dup(orig_comm,copy_comm,ierror)
  MPI_Comm_dup(orig_comm, & copy_comm);
  ```

- **Some libraries use this to avoid some initial implementation bugs or for various other reasons**
- **For example, the SParse Object Oriented Linear Equations Solver (SPOOLES) and some FFT libraries such as FFTW use** `MPI_Comm_dup`**, perhaps to fix some implementation bugs**

[14]M. Ganesh, MATH440/540, SPRING 2018
  ```

- [15]**Another approach to create a new communicator** `new_comm` **is to first create a new group of processing cores from an existing group of processing cores by including or excluding certain cores**
- **First to identify a group of processing cores** `pres_group` **in an existing communicator,** `pres_comm`**, use:**

```
CALL MPI_COMM_GROUP(pres_comm,pres_group,ierror)
MPI_Comm_group(pres_comm,pres_group,ierror)
```

- **For example, using our earlier notation, with** `pres_comm` **being** `MPI_COMM_WORLD` **the above command will yield a** `num_cores` **dimensional vector** `pres_group` **with** i**-th component being the integer** $i-1$**, for** `i = 1, ..., num_cores`

- **Suppose that we want to use a subset of processing cores in** `pres_comm`**, consisting of, say, only** `my_num` **number of specific cores to create a new communicator**
- **In this case, create a** `my_num` **cores dimensional vector, say,** `my_use` **with elements being integers corresponding to the specific processing cores**

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

- **[16]For example, with** `my_num`$= 4$**, and the specific cores with ranks** $5, 9, 15, 22$**, create a** 4**-dimensional vector:** `my_use`$= (/5, 9, 15, 22/)$
- **Then to create a new group of processors, say,** `new_group`**, (declare first as integer in Fortran and in C as** `MPI_Group new_group`**) and use:**

```
CALL MPI_GROUP_INCL(pres_group,my_num,my_use,new_group,ierror)
MPI_Group_incl(pres_group,my_num,my_use,&new_group);
```

- **To exclude certain group of processors, say,** `dont_use` **(with total number** `dont_num`**) from** `pres_group` **and create** `new1_group` **we may use:**

```
CALL MPI_GROUP_EXCL(pres_group,dont_num,dont_use,new1_group,ierror)
MPI_Group_excl(pres_group,dont_num,dont_use,& new1_group);
```

- **To check whether a processing core from** `MPI_COMM_WORLD` **belongs to** `new_group`**, we may use (after declaring an integer variable** `rank_id`**)**

```
CALL MPI_GROUP_RANK(new_group,rank_id, ierror)
MPI_Group_rank(new_group,rank_id, ierror);
```

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

- **[17]If a processing core from `MPI_COMM_WORLD` encounters the above commands and if it does not belong to the `new_group`, then the value of `rank_id` will be the intrinsic value `MPI_UNDEFINED`**
- **We may use `rank_id` and, say, `IF` statements to proceed and do ....**
- **We may use the subset of processing cores in `pres_comm` collected in `new_group` to create a new communicator, say `my_comm` (declaration in Fortran and C as before) :**

```
CALL MPI_COMM_CREATE(pres_comm,new_group,my_comm, ierror)
```

```
CALL MPI_Comm_create(pres_comm,new_group,&my_comm);
```

- **To release groups, such as `new_group`, and hence free resources, use**

```
CALL MPI_Group_free(new_group, ierror)
MPI_Group_free(new_group)
```

- **There are several `MPI_Group_*` commands .....**

Exercise 3:
Take a copy of your code for solving Exercise 2. Create the N new communicators by replacing `MPI_Comm_split` with appropriate MPI group and create commands discussed above.

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

- [18]So far, we used contiguous arrays and used standard data types such as `MPI_DOUBLE_PRECISION` for transfer of data between processing cores
- Several programming languages allow creation of arbitrary datatypes

- Corresponding to these MPI allows definition of associated datatypes, using derived datatypes
- MPI derived datatypes allow:
  - ⋆ transfer of variables with extended precision and range
  - ⋆ transfer of contiguous blocks of the same datatype lumped together
  - ⋆ transfer of noncontiguous blocks of the same datatype lumped together
  - ⋆ transfer of Fortran and C/C++ structures
- One may avoid the second and third lumping process items above by using instead language defined RESHAPE, PACK/UNPACK to create appropriate variables and use standard send and receive

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

- [19]**For example, as discussed earlier in this course, FORTRAN 90+ allow us to select precision for** `INTEGER` **datatype and for** `REAL` **and** `COMPLEX` **datatypes precision and/or range by using, respectively,** `SELECTED_INTEGER_KIND` **and** `SELECTED_REAL_KIND`

- **MPI derived datatypes allow creation of associated data types using the command** `MPI_Type_create_f90_integer`,
  `MPI_Type_create_f90_real`,
  `MPI_Type_create_f90_complex`

- **Once we create a particular derived kind, we need to use**
  `MPI_Type_committ`
  before appropriate transfers
- **If we no longer need our derived MPI datatypes, we can free the committment by using the command**
  `MPI_Type_free`
- **The format for using the above commands are discussed in the following examples**

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

[20]**Example**:

```
.....................
.....................
INTEGER, PARAMETER : precis = 15
INTEGER, PARAMETER :: my_int_kind = SELECTED_INTEGER_KIND(precis)
.....................
INTEGER(KIND= my_int_kind) , DIMENSION(100) :: my_array
INTEGER :: master, my_int_type , ierror
.....................
.....................
my_array = .........
CALL MPI_Type_create_f90_integer(precis, my_int_type, ierror )
CALL MPI_Type_commit (my_int_type , ierror )
.....................
.....................
CALL MPI_Bcast (my_array, 100, my_int_type, master, MPI_COMM_WORLD, ierror
.....................
.....................
CALL MPI_Type_free (my_int_type, ierror )
```

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

[21]**Example**:

```
.....................
.....................
INTEGER, PARAMETER : precis = 15
INTEGER, PARAMETER : range = 300
INTEGER, PARAMETER :: my_real_kind = SELECTED_REAL_KIND(precis,range)
INTEGER, PARAMETER :: my_real1_kind = SELECTED_REAL_KIND(precis)
.....................
.....................
REAL(KIND= my_real_kind) , DIMENSION(100) :: my_array
REAL(KIND= my_real1_kind) , DIMENSION(1000) :: my_array1
INTEGER :: master, my_real_type , my_real1_type,  ierror
.....................
.....................
my_array = .........
my_array1 = .........
.....................
.....................
.....................
```

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

```
.....................
CALL MPI_Type_create_f90_real(precis, range, my_real_type, ierror)
CALL MPI_Type_commit (my_real_type , ierror )
.....................
.....................
CALL MPI_Bcast (my_array, 100, my_real_type, master, MPI_COMM_WORLD, ierro
CALL MPI_Type_create_f90_real(precis, MPI_UNDEFINED, my_real1_type, ierror
.....................
.....................
CALL MPI_Type_free (my_real_type, ierror )
.....................
CALL MPI_Type_commit (my_real1_type , ierror )
.....................
.....................
CALL MPI_Bcast (my_array1,1000,my_real1_type,master,MPI_COMM_WORLD, ierror
.....................
.....................
CALL MPI_Type_free (my_real1_type, ierror )
.....................
.....................
```

[22]**Example**:

```
.....................
.....................
INTEGER, PARAMETER : precis = 15
INTEGER, PARAMETER : range = 300
INTEGER, PARAMETER :: my_real_kind = SELECTED_REAL_KIND(precis,range)
INTEGER, PARAMETER :: my_real1_kind = SELECTED_REAL_KIND(precis)
.....................
.....................
COMPLEX(KIND= my_real_kind) , DIMENSION(100) :: my_array
complex(KIND= my_real1_kind) , DIMENSION(1000) :: my_array1
INTEGER :: master, my_complex_type , my_complex1_type,  ierror
.....................
.....................
my_array = .........
my_array1 = .........
.....................
.....................
.....................
```

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

```
.....................
CALL MPI_Type_create_f90_complex(precis, range, my_complex_type, ierror)
CALL MPI_Type_commit (my_complex_type , ierror )
.....................
.....................
CALL MPI_Bcast (my_array, 100, my_complex_type, master, MPI_COMM_WORLD, ie
CALL MPI_Type_create_f90_complex(precis, MPI_UNDEFINED, my_complex1_type,
.....................
.....................
CALL MPI_Type_free (my_complex_type, ierror )
.....................
CALL MPI_Type_commit (my_complex1_type , ierror )
.....................
.....................
CALL MPI_Bcast (my_array1,1000,my_complex1_type,master,MPI_COMM_WORLD, ier
.....................
.....................
CALL MPI_Type_free (my_complex1_type, ierror )
.....................
.....................
```

[23]**Exercise 4: Take a copy of at least two of your MPI codes. Replace all `INTEGER` and `DOUBLE_PRECISION` variables in your codes with `my_int_type` and `my_real_type` as described in the last example. Compile and run your code with MPI derived datatypes.**

- **Recall, for example**

  ```
  CALL MPI_Send(buf, count, datatype, to, to_tag, comm, ierror)
  ```

- **Here, the array `buf` is assumed to be of contiguous with `count` number of elements each with a fixed `datatype`**

- **The MPI derived datatype command `MPI_Type_contiguous` faciliates concatenation datatypes of all elements in a continguous array (that is, with no offset) to create a new datatype `new_datatype`**

- **In Fortran, declare `new_datatype` as an integer and in C use `MPI_Datatype new_datatype`**

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

- [24]**For example,**
  `CALL MPI_Send(buf, count, old_datatype, to, to_tag, comm, ierror)`
  is same as the four commands

```
CALL MPI_Type_contiguous(count, old_datatype, new_datatype, ierror)
CALL MPI_Type_commit (new_datatype , ierror)
CALL MPI_Send(buf, 1, new_datatype, to, to_tag, comm, ierror)
CALL MPI_Type_free(new_datatype , ierror)
```

- **The** `new_datatype` **is produced by making** `count` **copies of** `old_datatype` **, with the displacements incremented by the extent of the** `old_datatype`

- **The extent of** `old_datatype` **is the distance in bytes to skip/stride from the start of one instance of the datatype to the start of another instance of a datatype**
- **This extent (uniform stride integer) can be obtained using the command**
  `CALL MPI_Type_extent (old_datatype, extent, ierror)`

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

- **The advantage of `MPI_Type_contiguous` is of course in concatenation of various contiguous datatypes:**
- **The general format is:**

```
CALL MPI_Type_contiguous(count, old_datatype, new_datatype,ierror)
MPI_Type_contiguous(count, old_datatype, new_datatype)
```

- **Suppose that we are interested in transferring non-contiguous blocks**

- **For example, consider an array `buf` with with 100 elements. Suppose that we are interested in transferring i-th element of the buffer for $i = 1, 11, 21, ..., 91$. That is a total of ten numbers. In this case `count` $= 10$, `block_len` $= 1$, `stide` $= 10$, `old_datatype` is the datatype of `buf`**
- **Such a transfer, say send, can be easily achieved by first creating a non-contiguous derived datatype:**

```
CALL MPI_Type_vector( &
count, block_len, stride, old_datatype, new_datatype,ierror)
CALL MPI_Send(buf,block_len,new_datatype, to,tag,comm,ierror)
```

- [25]**Note that the above transfers can be done without derived datatypes**

- **Suppose that we have a structure consisting of four distinct datatypes, say,** `INTEGER, DOUBLE_PRECISION, CHARACTER, LOGICAL` **and the respective number of elements is, say,** $2, 7, 12, 22$ **and the byte offset of the start of each type is** $0, 16, 100, 500$
- **To create an associated MPI structure, let (i)** `count` $= 4$**;**
  (ii) `bloc_arr` **be a** `count` **dim. array of blocklengths with** $2, 7, 12, 22$**;**
  (iii) `disp_arr` **be a** `count` **dim. array of displacements with** $0, 16, 100, 500$**;**
  (iv) and `type_arr` **be a** `count` **dimensional array**
  with entries `MPI_INTEGER, MPI_DOUBLE_PRECISION, MPI_CHARCTER, MPI_LOGICAL`
- **Then to create** `MPI_my_str` **datatype, use:**

```
CALL MPI_Type_struct( &
count, bloc_arr, disp_arr, type_arr, MPI_my_str, ierror)
CALL MPI_Type_commit (MPI_my_str , ierror)
CALL MPI_Type_free (MPI_my_str, ierror )
MPI_Type_struct(count, bloc_arr, disp_arr, type_arr, MPI_my_str)
MPI_Type_commit (MPI_my_str)          MPI_Type_free (MPI_my_str)
```

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