Week 2 - Introduction to Parallel Programming

With increasingly large solution fields it becomes impractical to run numeric simulations on a single processor due to the required time. Thus it is essential that most numerical programs are able to run efficiently and accurately in parallel.

This week we will be introduced to the Message Passing Interface (MPI) libraries.

Required Reading

Introduction to High Performance Computing for Scientists and Engineers by George Hager; Chapter 9, pg 203-213

The lab sheet will discuss the routines and their uses in a generalized way. It is good practise to aquint yourself with each new routine from the official manual of the OPEN-MPI web page. The lab sheet will provide links to each new routine.

`https://www.open-mpi.org/doc/`

MPI Initialization/Termination

All MPI programs have a basic framework that initializes and terminates the program. All MPI programs must include the MPI libraries at the very start of the program by either including the header file:

```
INDLUDE 'mpif.h'
```

OR using the mpi module.

```
USE mpi
```

In the PC Lab the mpi library installed requires the 'mpif.h' call. The mpi module call is recommended though: `https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report/node411.htm`. Without either of these calls, the compiler will not understand the MPI routines you write and will exit with an error.

All MPI program begin and end with the commands:

```
call MPI_INIT(mpierror)
```

`https://www.open-mpi.org/doc/v3.1/man3/MPI_Init.3.php`

```
call MPI_FINALIZE(mpierror)
```

`https://www.open-mpi.org/doc/v3.0/man3/MPI_Finalize.3.php`
Both commands have the INTEGER output 'mpierror' which will have a specific non-zero value if any error occurs within the routine. Further all MPI routines within GFORTRAN must output an 'mpierror' value with the variable declared at the start of the program.

MPI Communicators, Rank and Size

Three key concepts to any MPI programs are Communicators, Rank and Size.

A Communicator is a defined group of processes which allows communication within that group. Initially all processes in the program are defined by the global communicator:

`MPI_COMM_WORLD`

Almost all MPI routines must be supplied the communicator for which the routine is to be run. It is possible to further subdivide the global communicator into smaller groups. This can be used for example to more easily organise group (collective) communication between certain processors. This concept is shown below in Figure 1.

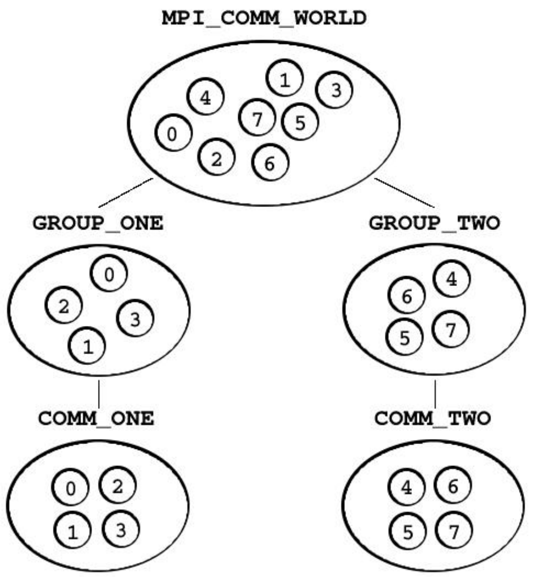


Figure 1: Communicators

All processes within a given communicator of size N have their unique rank (id number) from 0 to $(N-1)$. To determine the size of a particular communicator the following commands is passed:

```
call MPI_COMM_SIZE(communicator, mpisize, mpierror)
(https://www.open-mpi.org/doc/v4.0/man3/MPI_Comm_size.3.php)
```

Where the input is the communicator in question and the output 'mpisize', is of type INTEGER and is the number of processes.

The rank of a particular process is determined using the command:

```
call MPI_COMM_RANK(communicator, mpirank, mpierror)
(https://www.open-mpi.org/doc/v4.0/man3/MPI_Comm_rank.3.php)
```

Here again the input is the communicator and the output is the INTEGER rank of the individual process within that communicator. The rank of each processor (it's id) is an important input parameter for most MPI communication routines.

Compiling and Running MPI Programs

Compiling and executing MPI Fortran programs is very similar to executing serial Fortran programs. The two key differences are the compiler name and specifying the number of processes within the program.

To compile a program into an executable we use the 'mpif90' compiler in the command line which is analogous to 'gfortran' for serial programs. For example to compile the code 'mpitest.f90' into an executable 'mpitest.exe' we:

```
mpif90 mpitest.f90 -o mpitest.exe
```

To run the program we use the command 'mpirun' and must specify the number of processes for the program with the '-np' flag. To launch the program across 6 processes:

```
mpirun -np 6 mpitest.exe
```

If the number of processes exceeds the number of cores available in the hardware, the code will not run efficiently.

Practise Problem 1

Write a simple MPI program that prints the following to the screen:

- Size of Global Communicator (Number of Processes)
- Rank of the Individual Process (PID)
- "Hello world"

Download and open 'mpitute1.f90' again as a template to begin with.

Make sure you declare the variables for both RANK and SIZE before using the associated routines.

Try running your program with varying number of processes. You should get as many lines of output as you have processes.

MPI Send/Receive

When processes need to access memory allocated on neighbouring processes, the two processes must communicate this using the message passing interface (MPI). The most simple MPI communication is the SEND/RECV communication between 2 nodes which is done by executing the following commands:

```
call MPI_SEND (send-data,send-count,send_type,destination-ID,tag,communicator,mpierror)
```

`https://www.open-mpi.org/doc/v3.0/man3/MPI_Send.3.php`

```
call MPI_RECV (recv-data,recv-count,recv_type,sender-ID,tag,communicator,status,mpierror)
```

`https://www.open-mpi.org/doc/v3.0/man3/MPI_Recv.3.php`
Lets examine the corresponding terms in each routine.

The 'send-data' and 'recv-data' variables correspond to the data being transferred and received accordingly. If the data is an array adequate space must be allocated in the receiving array to accommodate the incoming data. The data of both is to be of the same type and explicitly specified later in the routine.

The 'send-count' and 'recv-count' is the INTEGER value of the number of data elements to be sent or received and must correspond with the size of the sending/receiving data.

'Send-type' and 'recv-type' are INTEGER values that specify the type of data being sent such as REAL, INTEGER, COMPLEX etc. The types are specified using an MPI prefix such as:

```
MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION,  MPI_CHAR
```

Alternatively the precision of REAL values may be specified such as:

```
MPI_REAL4, MPI_REAL8, MPI_REAL16
```

The 'destination-ID' and 'sender-ID' correspond to the INTEGER values that represent the individual ranks of the processes within the particular communicator and must match for a successful data exchange.

The 'tag' input is a unique INTEGER value that must match for the send to be successful. When running programs with many simultaneous data exchanges it is important to establish a precise system for the 'tag' values to ensure no mismatch between communications.

Again the 'communicator' refers to the communicator which will be used for this communication and hence provides a reference for the ID ranks of the processes.

Both commands output an 'mpierror' as all MPI routines as well as the RECV command outputing a 'status' value that provides feedback on the data transfer.

Further the SEND/RECV commands are blocking commands that will stop the process from executing any further lines of code until the matching command is executed on the target node and data transfer is complete. Thus the command serves as a way to synchronize your program to ensure no single process is ahead of the others. However when using blocking communications it is essential to ensure that your order the SEND/RECV commands appropriately to ensure no grid-lock in the program where all processes are awaiting a corresponding call that will never arrive.

To illustrate the idea of simple MPI SEND/RECV communication consider the following program:

```
! Simple program to swap some data between 2 processes
PROGRAM MPISENDRECV
    IMPLICIT NONE
    INCLUDE 'mpif.h'

    ! Variable Declaration
    INTEGER, DIMENSION(10) :: a,b,c
    INTEGER :: i, ierr, ndata, tag, stat(MPI_STATUS_SIZE), pid, comm_size

    CALL MPI_INIT(ierr) ! Initialize MPI Program

    CALL MPI_COMM_SIZE(MPI_COMM_WORLD,comm_size,ierr) ! Get Size of Communicator
    CALL MPI_COMM_RANK(MPI_COMM_WORLD,pid,ierr) ! Get Size of Communicator

    DO i = 1,10
        a(i) = pid*10 + i ! Fill in Data Array indidivual to each process
    ENDDO

    ndata = 10 ! Amount of data to send/recv
    tag = 101  ! Individual communication tag

    IF(PID.eq.0) then
        ! Send from ID 0 --> ID 1
        CALL MPI_SEND(a,ndata,MPI_INTEGER,pid+1,tag,MPI_COMM_WORLD,ierr)
        ! Recv from ID 1 --> ID 0
        CALL MPI_RECV(b,ndata,MPI_INTEGER,pid+1,tag,MPI_COMM_WORLD,stat,ierr)
    ELSE
        ! Recv from ID 0 --> ID 1
        CALL MPI_RECV(b,ndata,MPI_INTEGER,pid-1,tag,MPI_COMM_WORLD,stat,ierr)
      ! Send from ID 1 --> ID 0
        CALL MPI_SEND(a,ndata,MPI_INTEGER,pid-1,tag,MPI_COMM_WORLD,ierr)
    ENDIF
    c = a + b    ! Add together
    WRITE(*,*) "PID", pid, "c", c ! Write out to screen

    CALL MPI_FINALIZE(ierr) ! Terminate MPI Program
END PROGRAM MPISENDRECV
```

The program initializes the MPI libraries as discussed prior and declares all variables to be used. Note that the 'stat' variable is given the extra definition of MPI_STATUS_SIZE to accommodate for varying size of array data.

The 'a' array is filled with data unique to each of the processes and the total amount of data and data tag are defined.

The data is first sent from ID > ID 1 and stored in the 'b' array. The opposite is then done with data being sent from ID 1 > ID 0. Note that by virtue of being a blocking communication that the second data exchange will not occur until the first is complete.

Finally the 'a' and 'b' array are added together into the 'c' array and the total data is recovered on both processes.

Note that this is a very simplified example that only works on 2 processes. It is critical to ensure your code should work in the generic case of any number of processes and therefore be very careful and precise in your order of SEND/RECV, ID and data tags.

Master/Slave Concept

A useful and simple concept in MPI programming is to declare one process the master process (often pid = 0) and use this to organise read-in or write-out.

Practise Problem 2

Copy the above program 'mpisendrecv.f90' into your text editor of choice.

Try swapping the order of SEND and RECV on node two. What happens?

Can you change the code to send a DOUBLE_PRECISION array instead?

Can you change the code to send a single value instead of an array?

Practise Problem 3

Write a simple MPI program that transfers data 'x' sequentially from PID 0 to PID $N-1$ as illustrated in the figure below.

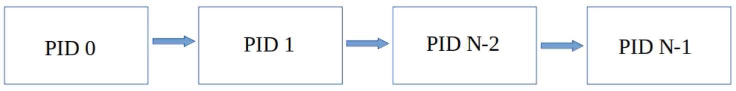
You must make your program work for any arbitrary number of processes. Try sending both a single INTEGER value or a 1D REAL array.

After each successful transfer have both the sending and receiving process print to the screen:

- Whether the process is sending or receiving
- The process PID
- PID of the sending/receiving process
- the value 'x'
- Number of sends completed
- Number of sends to go

An example of two iterations would be:

```
Send/Recv    PID    PID(send/recv)   'x'    Sends Completed   Sends to go
"Sending"    0      1                1234   1                 5
"Receiving"  1      0                1234   1                 5
"Sending"    1      2                1234   2                 4
"Receiving"  2      1                1234   2                 4
```



BONUS: Practise Problem 4

If time allows, write an MPI program that integrates:

$$x = sin(\theta), \;\; \theta \in (0, 2\pi) \tag{1}$$

Divide the integral into sections and have each process calculate its contribution and print the result and limits of integration to the screen. Gather all the data to a master node and print the total integral to the screen for confirmation.

Again ensure your program works for an arbitrary number of processes.

Monitor the performance of your program by either using the MPI_WTIME routine.
`https://www.open-mpi.org/doc/v3.0/man3/MPI_Wtime.3.php`

The final output should be similar to:

```
PID  Theta_Min  Theta_max   Integral
0    0          PI/6        1.366
(Repeat for all PID)

FINAL INTEGRAL = 0.0
TIME TAKEN     = 2.0
```

(Hint: Use a DO loop to efficiently gather the data to the master node.)

Lab 2 Mini-Assignment

Due Date: 23/08/2019 - 2pm

Consider the 1D Heated Rod problem from the mini-assignments in Week 1.

Build upon your code from the previous week to solve the problem using MPI to decompose the domain across multiple processes and solve the solution at t=2s in parallel.

The problem has the initial conditions, dimension and properties listed below:

The rod has a length $L = 1m$. Heat transfer through the rod is described by:

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial t^2}, \quad \alpha = 0.1\ m^2/s$$

The initial temperature distribution is prescribed by:

$$T = e^{-0.5((x-0.5L)/0.05)^2}$$

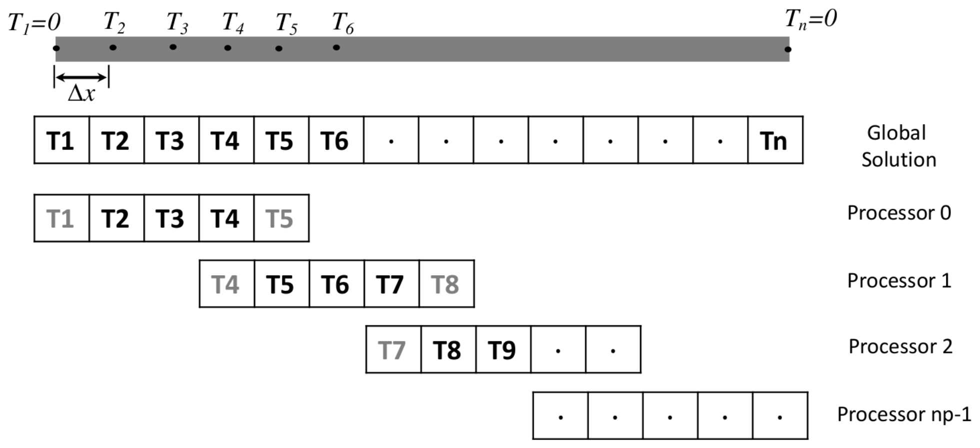
And with constant boundary conditions:

$$T(0) = T(L) = 0°\mathrm{C}$$

Begin by considering how you would decompose your domain across the multiple processes including the overlapping nodes that will need to be transferred across to neighbours. A visual representation of the solution field is shown below.

(Hint: During the initialization use global indices to describe the solution field across each process.)

(Hint: Clearly define the neighbour node ID for both the left and right swap during initialization.)



T_1=0
T_2
T_3
T_4
T_5
T_6
T_n=0
Δx
T1 T2 T3 T4 T5 T6 · · · · · · · Tn
Global Solution
T1 T2 T3 T4 T5
Processor 0
T4 T5 T6 T7 T8
Processor 1
T7 T8 T9 · ·
Processor 2
· · · · ·
Processor np-1