# MPI

Differences between OpenMP and MPI

| OpenMP	MPI
Add pragmas to existing program	Must rewrite program to describe how single process should operate on its data and communicate with other processes
Compiler + runtime system arrange for parallel execution	Explicit data movement: programmer must say exactly what data goes where and when
Rely on shared memory for communication	Advantage: Can operate on systems that don't have shared memory

Configuring MPI in Visual Studio

1. Download Microsoft MPI here and run the 2 installation programs.

2. In order for the system to find the **mpiexec** program, you need to do the following. Click on the **Windows Start icon** and type **env**. The dialog on the left most will appear. Follow the steps in the image below. [Note: if you don't see **MSMPI_BIN**, it means you have not successfully installed the 2 installation programs in step 1.]
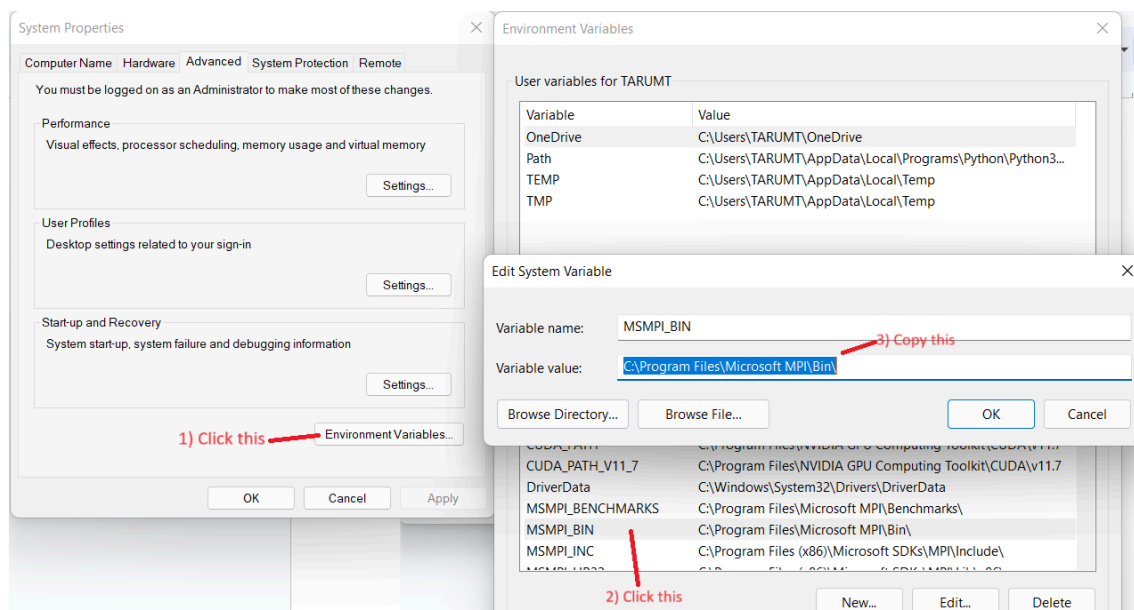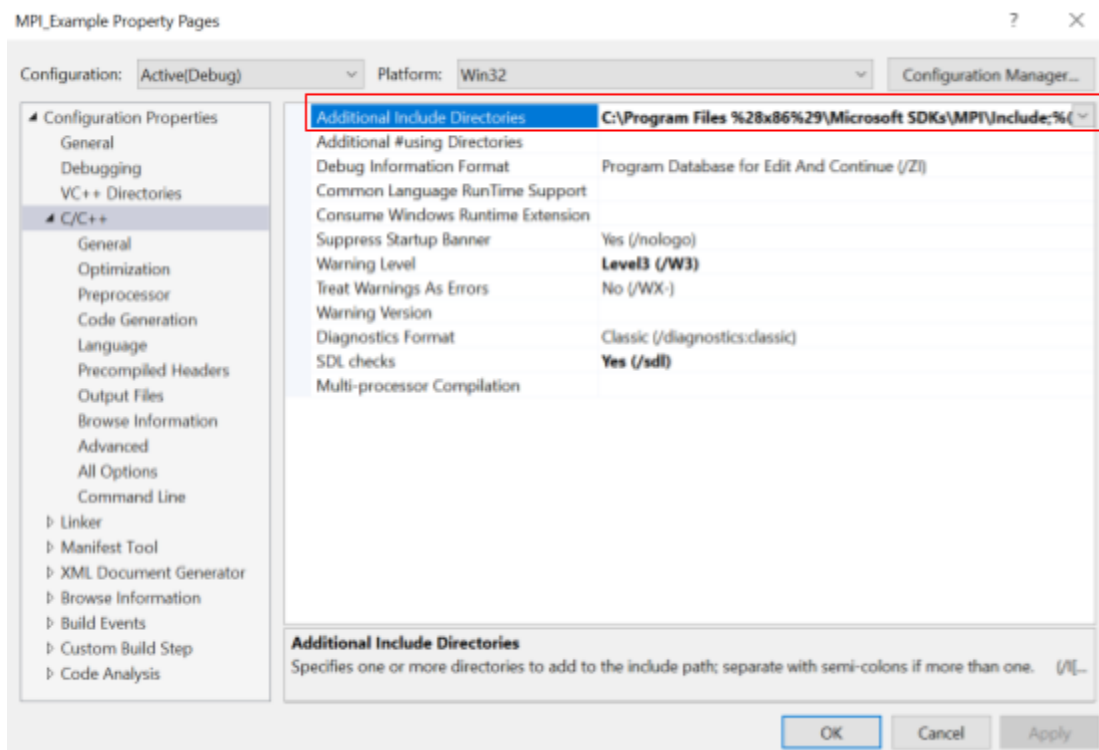
Click on cancel and then follow the steps in the following figure.
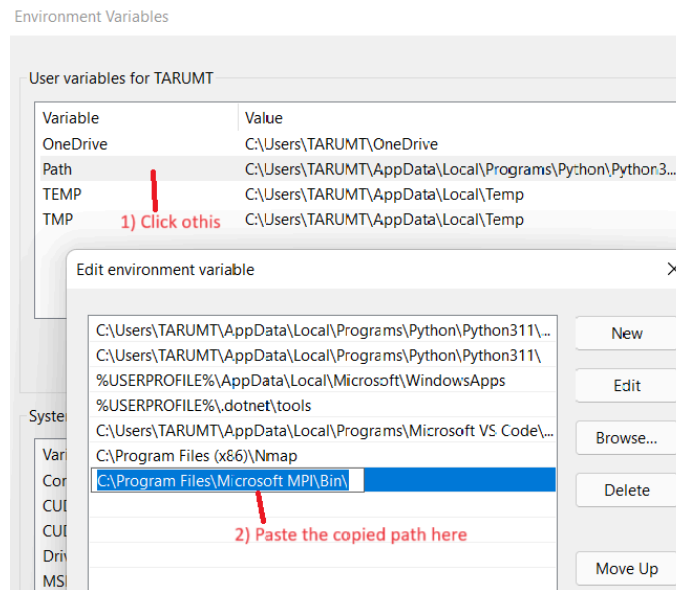
3. Configure project properties → C/C++ → Additional Include Directories (The path is where *mpi.h* is stored, normally is installed at "*C:\Program Files (x86)\Microsoft SDKs\MPI\Include*")
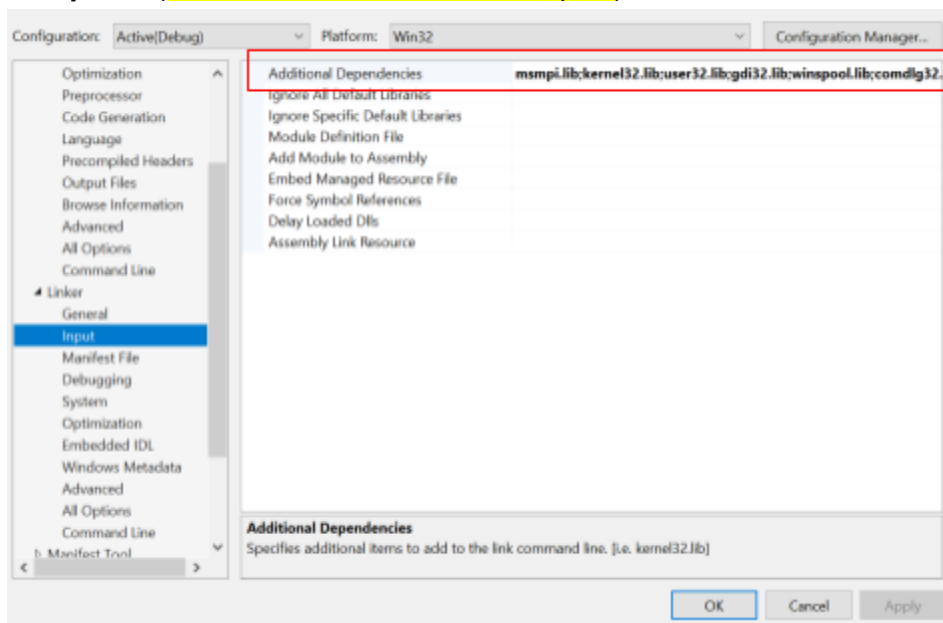

4.

5. Configure project properties Linker → General → Additional library Directories (The path is where msmpi.lib is stored, normally is installed at "*C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x64*")
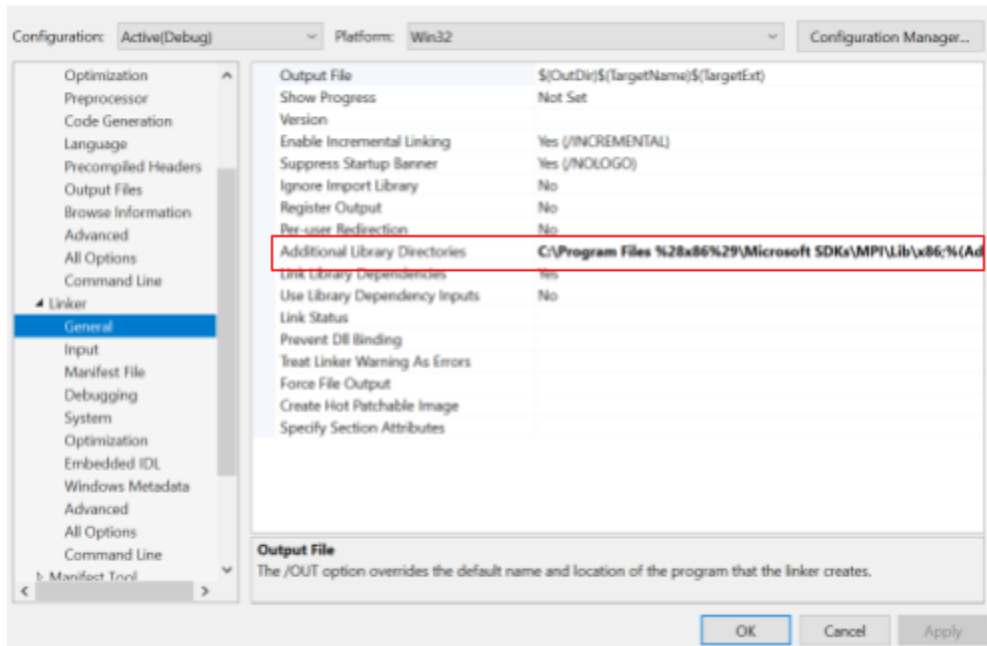
6. Configure project properties Linker → Input → Additional Dependencies, add in **msmpi.lib;** *(make sure the semicolon is copied)*

7. Add a C++ file with code downloaded here. Then run the exe file in Powershell with the command below:

```
mpiexec -n 4 filename.exe
```

The command **-n <np>** specifies the number of processes to be used

Reference: **[mpiexec command](#)**

Note:
* To successfully run *mpiexec* in PowerShell, we may need to include the following path in the system environment. To do that, press Windows+R keys combination, then type *sysdm.cpl* and then press Enter key.

In the pop up window, follow the following trails

System Properties > Advanced > Environment Variable> System Variable > Path > Edit > New

and enter the following path:

C:\Program Files\Microsoft MPI\Bin

In the PowerShell, change the directory to **x64\Debug**

> cd x64\Debug

and then run the program (the following command presumes that the program is called p6.exe)

> mpiexec -n 4 p6.exe

Example of the output after running the program:

```
PS C:\Users\X> mpiexec -n 4 "D:\Dr Tew\BMCS3003\Project1\Debug\Project1.exe"
Hello World
Hello World
Hello World
Hello World
Hello World from process rank(number) 0 from 4
Hello World from process rank(number) 3 from 4
Hello World from process rank(number) 2 from 4
Hello World from process rank(number) 1 from 4
PS C:\Users\X>
```

Starting And Running MPI Processes

The **Single Program, Multiple Data (SPMD)** model is only a single source, compiled into a single executable, the parallel run comprises a number of independently started MPI processes.

The example above is designed to give you an intuition for this one-source-many-processes setup.

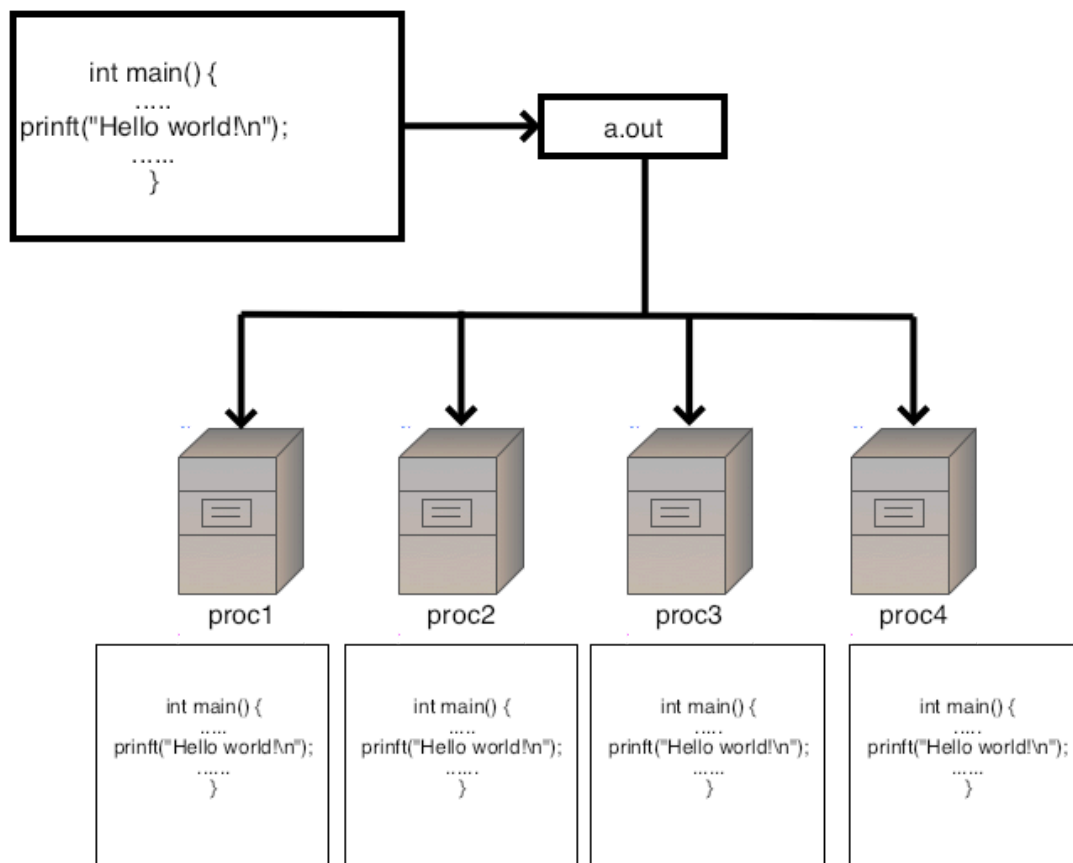For more information about Microsoft MPI functionality, please refer to:

https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi

Figure: Running a hello world program in parallel

Exercise

Q1   Now use the command **MPI_Get_processor_name** in between the init and finalize statement, and print out on what processor your process runs. Confirm that you are able to run the program in parallel with the corresponding number of processes. The character buffer needs to be allocated by you, it is not created by MPI, with size at least **MPI_MAX_PROCESSOR_NAME**.

Process And Communicator Properties: Rank And Size

To distinguish between processes in a communicator, MPI provides two calls:
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
states the process ID in rank

MPI_Comm_size(MPI_COMM_WORLD, &size);
reports how many processes there are in the communication

If every process executes the MPI_Comm_size call, they all get the same result, namely the total number of processes in your run. On the other hand, if every process executes MPI_Comm_rank , they all get a different result, namely their own unique number, an integer in the range from zero to the number of processes minus 1. See Figure below:
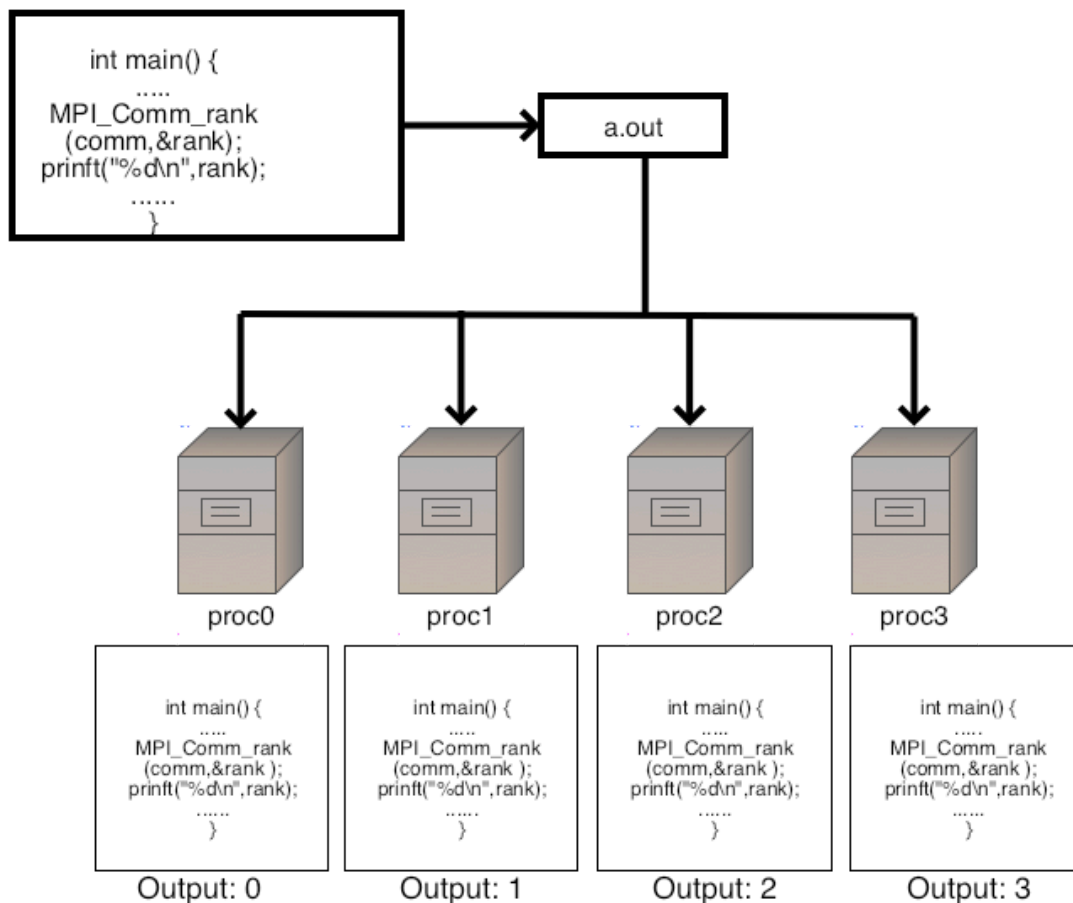

Figure: Parallel program that prints process rank

MPI_Reduce

Similar to `MPI_Gather`, `MPI_Reduce` takes an array of input elements on each process and returns an array of output elements to the root process. The output elements contain the reduced result. The prototype for `MPI_Reduce` looks like this:

```
MPI_Reduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    int root, //Recommend to use rank/process 0 because already receive
    MPI_Comm communicator)
```

The `send_data` parameter is an array of elements of type `datatype` that each process wants to reduce. The `recv_data` is only relevant to the process with a rank of `root`. The `recv_data` array contains the reduced result and has a size of `sizeof(datatype) * count`. The `op` parameter is the operation that you wish to apply to your data. MPI contains a set of common reduction operations that can be used. Although custom reduction operations can be defined, it is beyond the scope of this lesson. The reduction operations defined by MPI include:

- `MPI_MAX` - Returns the maximum element.
- `MPI_MIN` - Returns the minimum element.
- `MPI_SUM` - Sums the elements.
- `MPI_PROD` - Multiplies all elements.
- `MPI_LAND` - Performs a logical *AND* across the elements.
- `MPI_LOR` - Performs a logical *OR* across the elements.
- `MPI_BAND` - Performs a bitwise *AND* across the bits of the elements.
- `MPI_BOR` - Performs a bitwise *OR* across the bits of the elements.
- `MPI_MAXLOC` - Returns the maximum value and the rank of the process that owns it.
- `MPI_MINLOC` - Returns the minimum value and the rank of the process that owns it.

Below is an illustration of the communication pattern of `MPI_Reduce`.

P6 - Distributed Process Management

MPI_Reduce
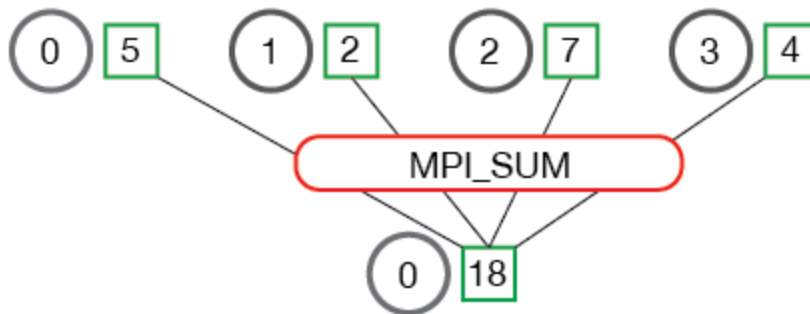
In the above, each process contains one integer. `MPI_Reduce` is called with a root process of 0 and using `MPI_SUM` as the reduction operation. The four numbers are summed to the result and stored on the root process.

It is also useful to see what happens when processes contain multiple elements. The illustration below shows reduction of multiple numbers per process.

MPI_Reduce
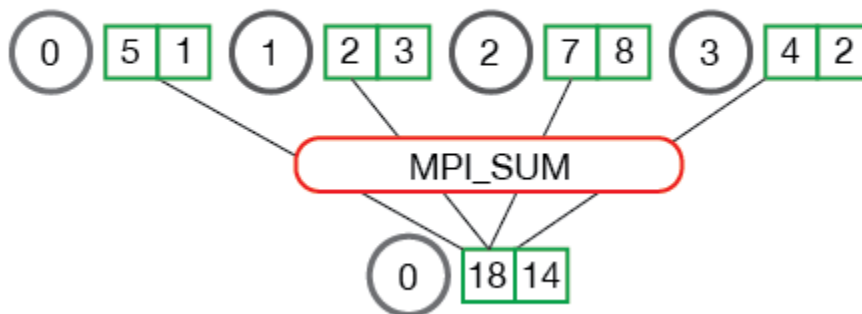
The processes from the above illustration each have two elements. The resulting summation happens on a per-element basis. In other words, instead of summing all of the elements from all the arrays into one element, the ith element from each array is summed into the ith element in the result array of process 0.

Q2 $f(x)=4/(1+x^2)$, so PI is the integral of $f(x)$ from 0 to 1. Then PI can be easily calculated by using the trapezoidal rule. Download the code here. Besides using MPI_Comm_rank and MPI_Comm_size above, initiate and terminate MPI execution environments as well. Also, synchronize the parallel processes by adding the code below at the appropriate segment.
MPI_Barrier(MPI_COMM_WORLD);
MPI_Reduce(&result, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

P6 - Distributed Process Management

Q3   A common need is for one process to get data from the user, either by reading from the terminal or command line arguments, and then to distribute this information to all other processors [*MPI_Bcast*(*&value, 1, MPI_INT, 0, MPI_COMM_WORLD*)].
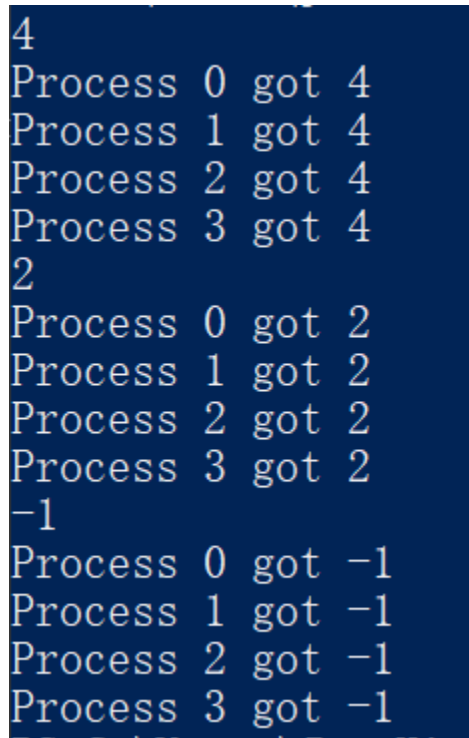
Write a program that reads an integer value from the terminal and distributes the value to all of the MPI processes. Each process should print out its rank and the value it received. Values should be read until a negative integer is given as input.

You may want to use these MPI routines in your solution:
MPI_Init, MPI_Comm_rank, MPI_Bcast, MPI_Finalize

Download the code here. You may also make use of fflush(stdout) to display the output after every input.
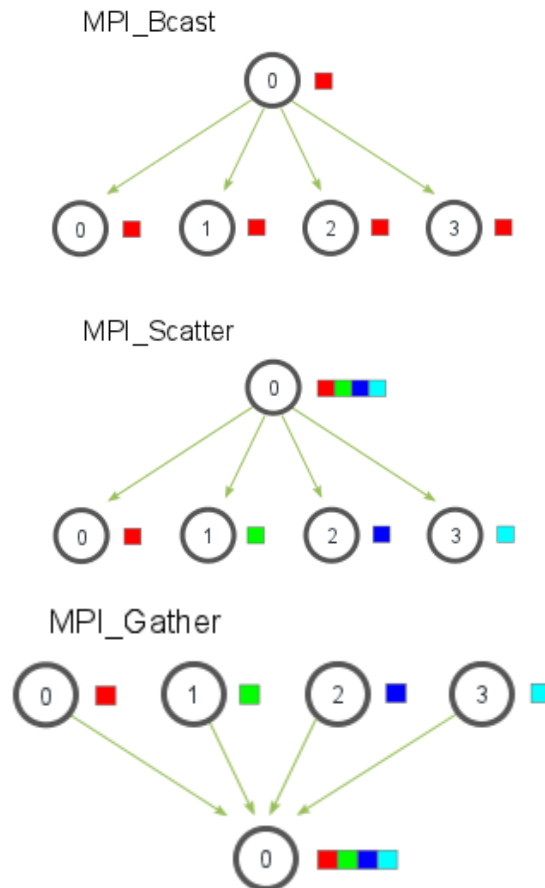
Output:

```
4
Process 0 got 4
Process 1 got 4
Process 2 got 4
Process 3 got 4
2
Process 0 got 2
Process 1 got 2
Process 2 got 2
Process 3 got 2
-1
Process 0 got -1
Process 1 got -1
Process 2 got -1
Process 3 got -1
```
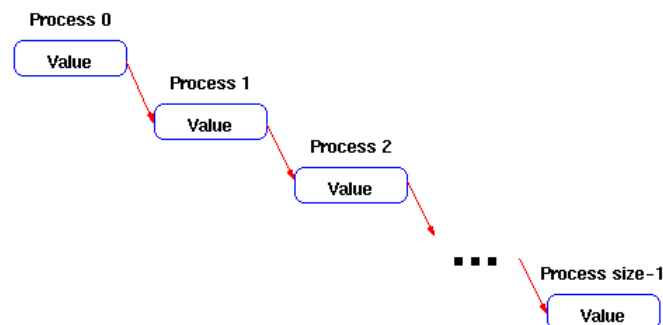
Note:
The list of MPI data types can be found here:
https://rookiehpc.org/mpi/docs/mpi_datatype/index.html

MPI_Bcast

MPI_Scatter

MPI_Gather

Q4   Write a program that takes data from process zero and sends it to all of the other processes by sending it in a ring. That is, process i should receive the data and send it to process i+1, until the last process is reached.

Assume that the data consists of a single integer. Process zero reads the data from the user.

You may want to use these MPI routines in your solution:

   i.   MPI_Send(&value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
   ii.   MPI_Recv(&value, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, &status);

Note:

P6 - Distributed Process Management

You will need this function to find the size of the threads:
MPI_Comm_size(MPI_COMM_WORLD, &size);

Download the code here.

Q5   Computing average numbers with MPI_Scatter and MPI_Gather.
The program takes the following steps:
1. Generate a random array of numbers on the root process (process 0).
2. Scatter the numbers to all processes, giving each process an equal amount of numbers.
3. Each process computes the average of their subset of the numbers.
4. Gather all averages to the root process. The root process then computes the average of these numbers to get the final average.

You may want to use these MPI routines in your solution:

i.   *MPI_Scatter(rand_nums, num_elements_per_proc, MPI_DOUBLE, sub_rand_nums, num_elements_per_proc, MPI_DOUBLE, 0, MPI_COMM_WORLD);*

ii.   *MPI_Gather(&sub_avg, 1, MPI_DOUBLE, sub_avgs, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);*

Download the code here.

```
Avg of all elements is 0.479612
Avg computed across original data is 0.479612
```