MPI

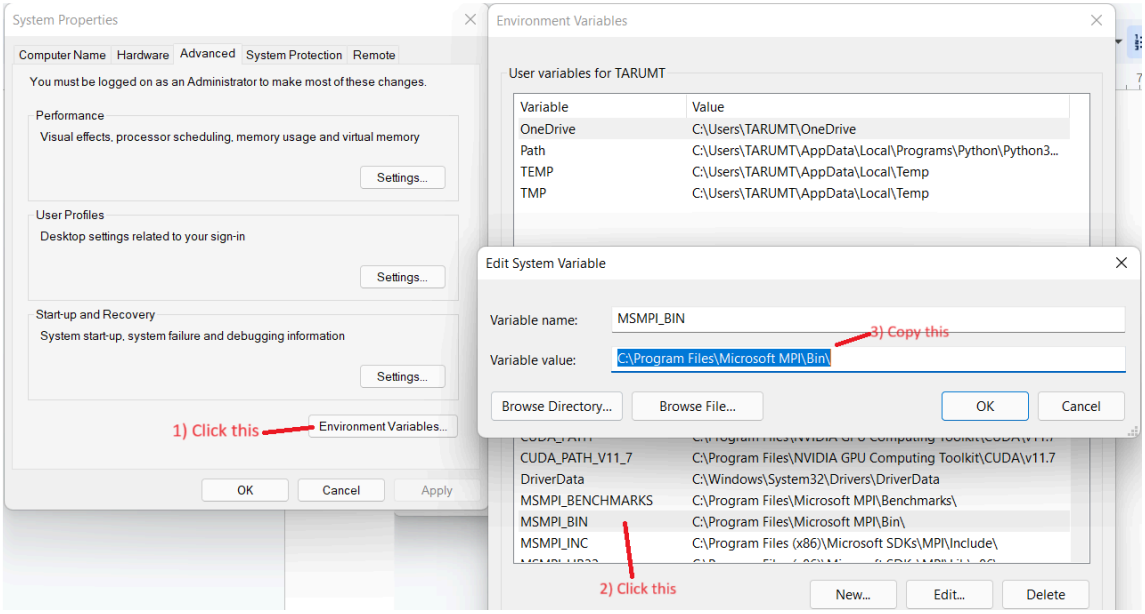
Differences between OpenMP and MPI

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

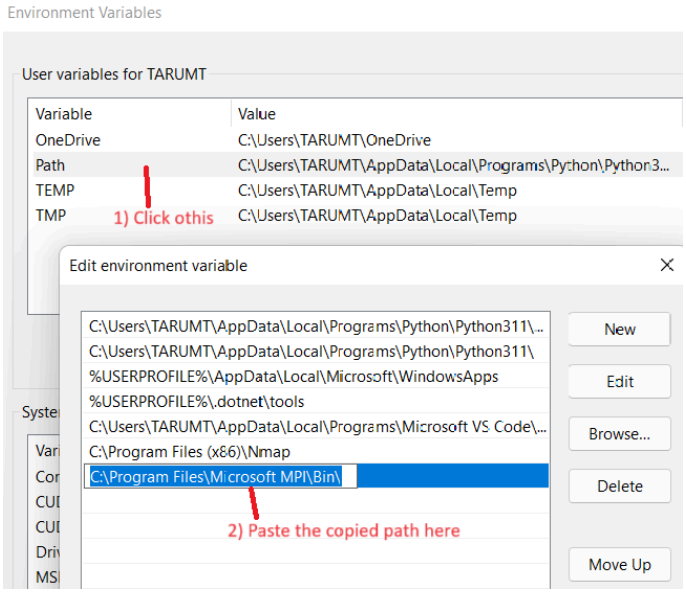
Configuring MPI in Visual Studio

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

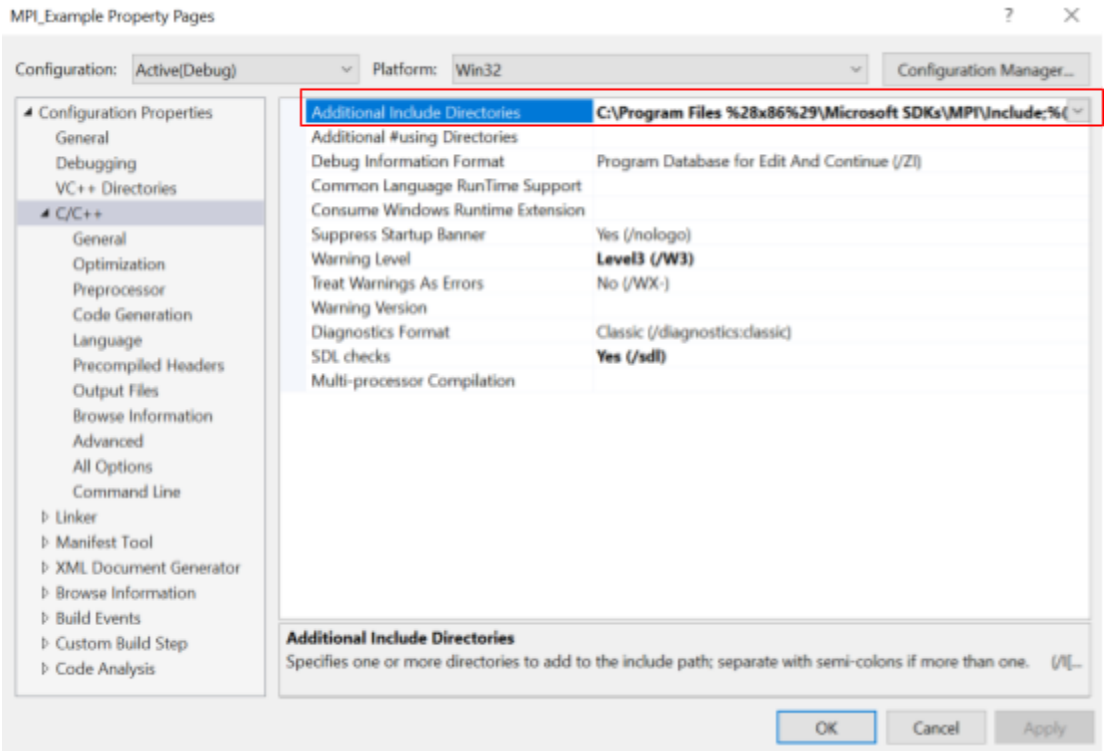
2. In order for the system to find the **mpiexec** program, you need to do the following. Click on the **Windows Start icon** and type **env**. The dialog on the left most will appear. Follow the steps in the image below. [Note: if you don't see **MSMPI_BIN**, it means you have not successfully installed the 2 installation programs in step 1.]



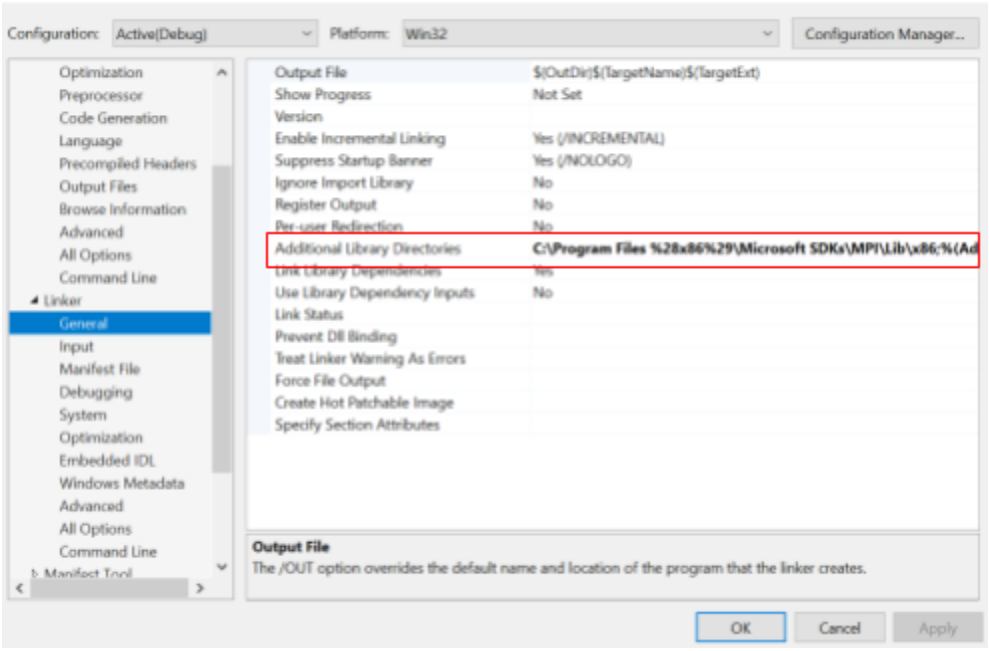
Click on cancel and then follow the steps in the following figure.



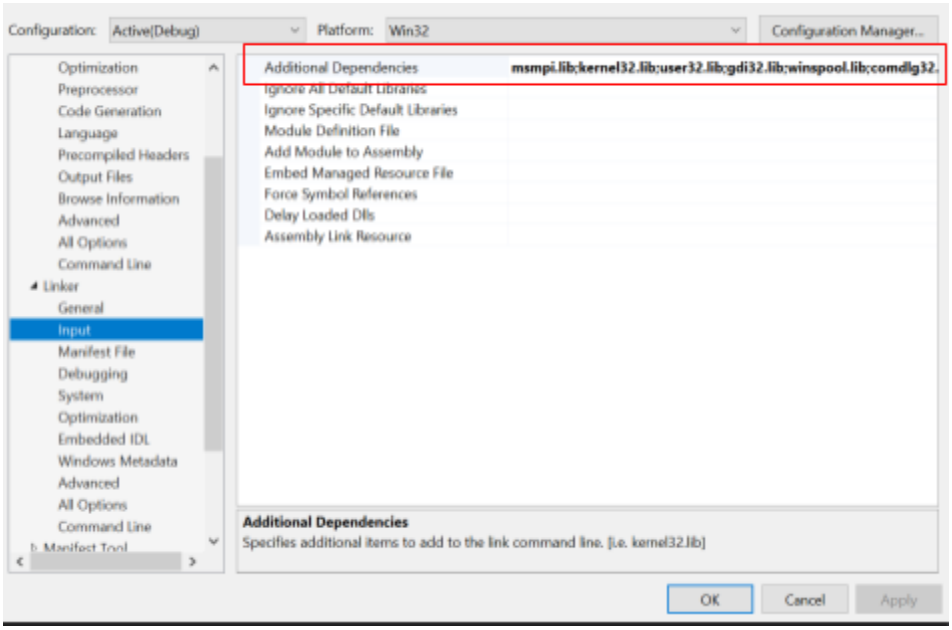
3. Configure project properties → C/C++ → Additional Include Directories (The path is where *mpi.h* is stored, normally is installed at "*C:\Program Files (x86)\Microsoft SDKs\MPI\Include*")

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

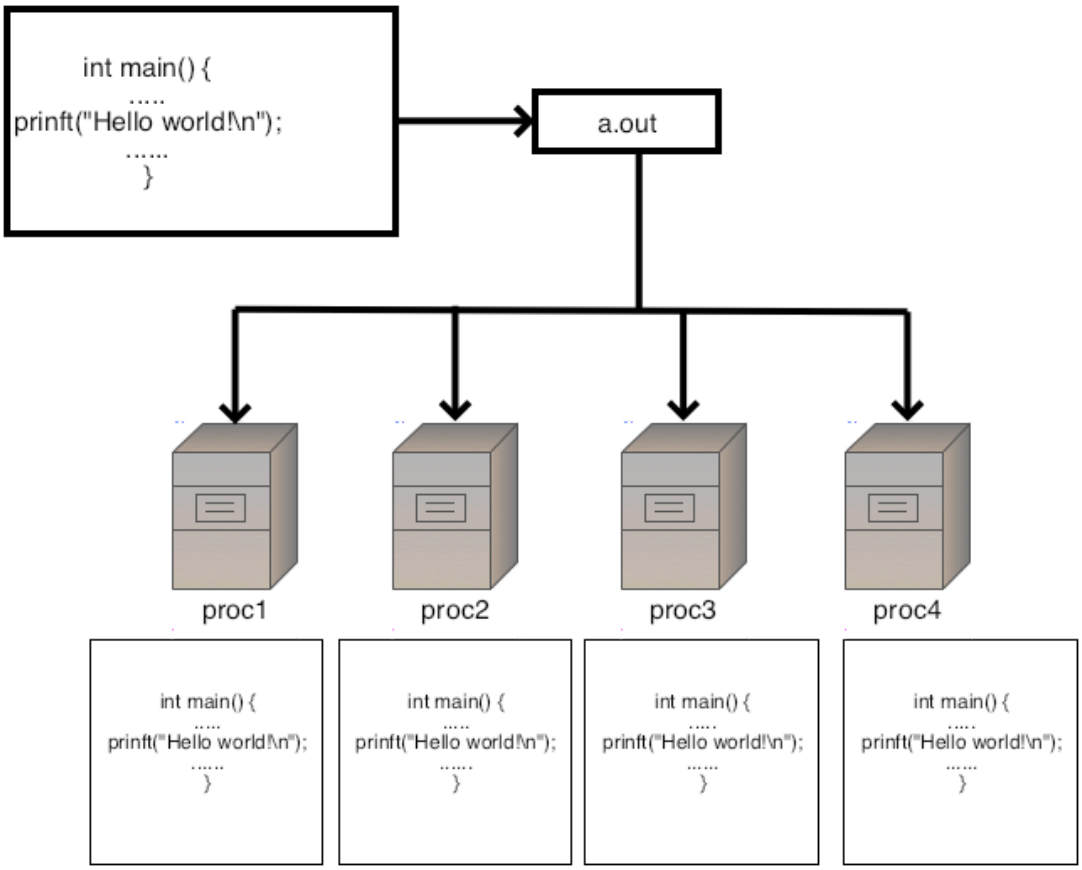


Figure: Running a hello world program in parallel

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

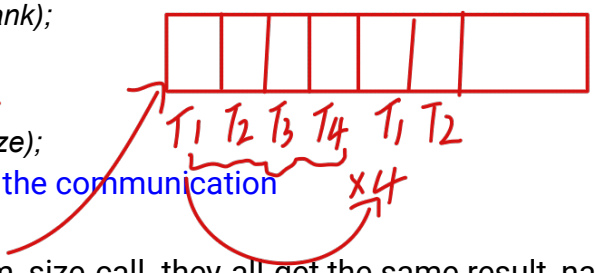
To distinguish between processes in a communicator, MPI provides two calls:

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



Why does we need rank size

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:

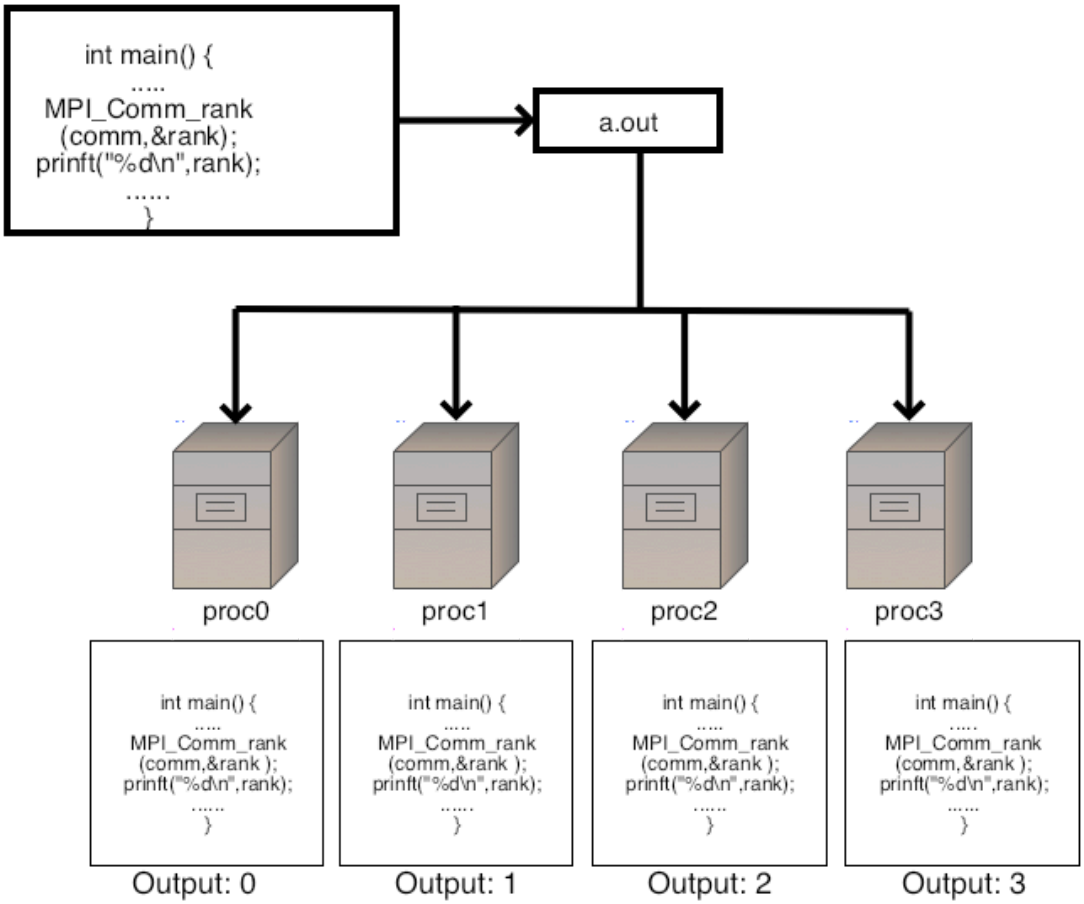
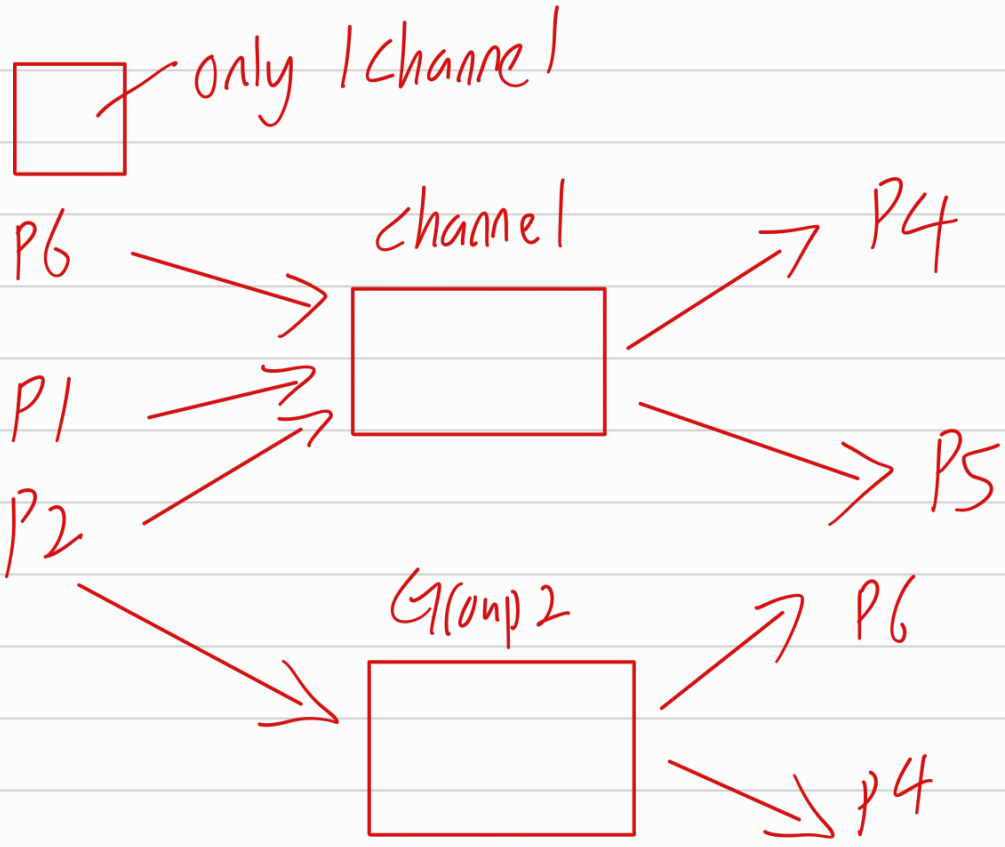


Figure: Parallel program that prints process rank

what is MPI_COMM_WORLD



decide how to join group / who to join

01_Rank_NumReq_GetProcessorName.cpp

```cpp
// run in cmd/powershell with mpiexec -n 4 01_Rank_NumReq_GetProcessorName
#include <iostream>
#include <mpi.h>

using namespace std;

int main(int args, char **argvs)
{
    cout << "Hello World" << endl;
    int rank = 0, numOfProcess = 0;

    MPI_Init(&args, &argvs);
    // The default communicator is called MPI_COMM_WORLD. It basically groups all the
    // processes when the program started. If you take a look at the example below,
    // you see a depiction of a program ran with five processes. Every process is
    // connected and can communicate inside this communicator.
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);         // process id
    MPI_Comm_size(MPI_COMM_WORLD, &numOfProcess); // number of thread/process

    // Single by single process
    if (rank == 0)
    {
        char name[MPI_MAX_PROCESSOR_NAME];
        int len;
        cout << "Hi there, from rank " << rank << endl;
        MPI_Get_processor_name(name, &len);
        cout << "PC name is " << name << endl;
    }

    cout << "Hello World from process rank(number) " << rank << " from " << numOfProcess << endl;
    MPI_Finalize();
    return 0;
}
```

print the processor name

Output:

```
└─ mpirun -np 4 ./01_Rank_NumReq_GetProcessorName
Hello World
Hello World
Hello World
Hello World
Hello World from process rank(number) 2 from 4
Hello World from process rank(number) 1 from 4
Hi there, from rank 0
PC name is fulim
Hello World from process rank(number) 0 from 4
Hello World from process rank(number) 3 from 4
```

4 times

single process

processor name

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:

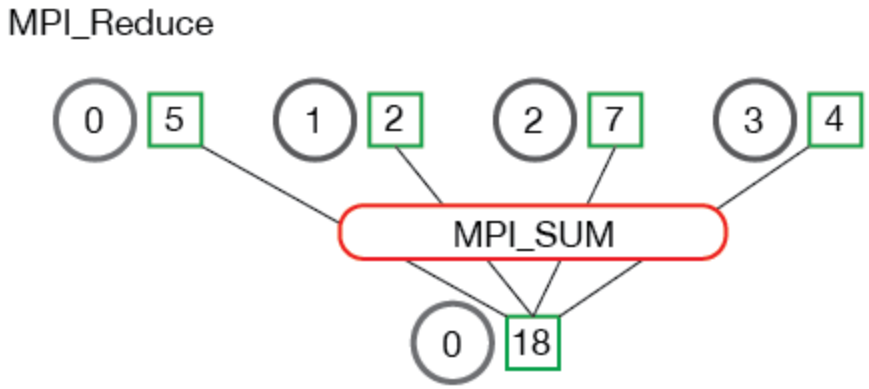
```
MPI_Reduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
   int root, //Recommend to use rank/process 0 because already receive
    MPI_Comm communicator)
```

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.

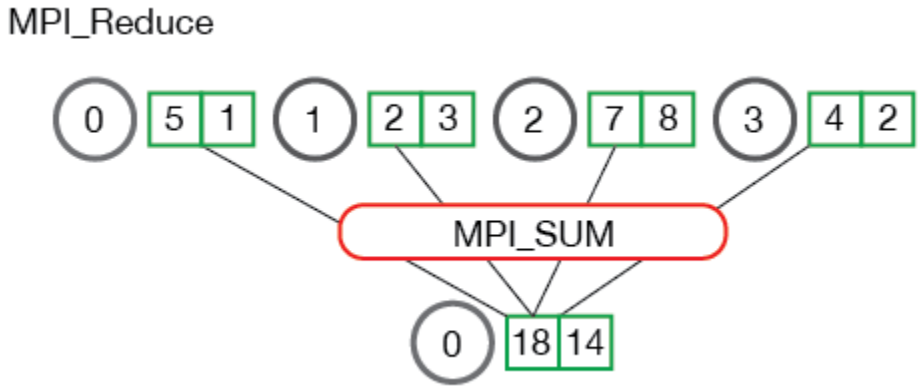
Below is an illustration of the communication pattern of `MPI_Reduce`.

P6 - Distributed Process Management



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.



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 i^{th} element from each array is summed into the i^{th} 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);

02_MPI_Reduce.cpp → Solve Compute PI problem

```cpp
// This program is to caculate PI using MPI
// The algorithm is based on integral representation of PI. If f(x)=4/(1+x^2),
// then PI is the intergral of f(x) from 0 to 1
#include <stdio.h>
#include <mpi.h>
#define N 1E7
#define d (1 / N)
#define d2 (d * d)

int main(int argc, char *argv[]) {
    int rank, size, error, i;
    double pi = 0.0, result = 0.0, sum = 0.0, begin = 0.0, end = 0.0, x2;

    // Init MPI
    MPI_Init(&argc, &argv);

    // Get process ID (rank)
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Get processes Number (size)
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Synchronize all processes and get the begin time
    begin = MPI_Wtime();

    // Each process will caculate a part of the sum
    for (i = rank; i < N; i += size) {
        x2 = d2 * i * i;
        result += 1.0 / (1.0 + x2);
    }

    // Sum up all results
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Reduce(&result, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    // Synchronize all processes and get the end time
    end = MPI_Wtime();

    // Caculate and print PI
    if (rank == 0) {
        pi = 4 * d * sum;
        printf("np=%2d;    Time=%fs;    PI=%.12lf\n", size, end - begin, pi);
    }

    // Finalize MPI
    error = MPI_Finalize();

    return 0;
}
```

Init

get process id

get num of process

Solution

Reduce the result to one sum

single process

close

all in parallel zone

output

```
└─ mpirun -np 4 ./02_MPI_Reduce
np= 4;    Time=0.008421s;    PI=3.141592753590
```

as num of process larger, run faster

```
○  ~/l/Parallel-Computing/Practical6  ma
└─ mpirun -np 8 ./02_MPI_Reduce
np= 8;    Time=0.004555s;    PI=3.141592753590
```

P6 - Distributed Process Management

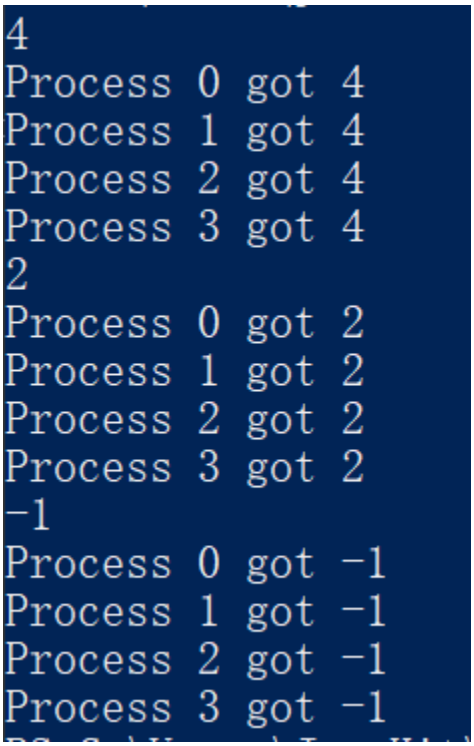
Q3 A common need is for one process to get data from the user, either by reading from the terminal or command line arguments, and then to distribute this information to all other processors [*MPI_Bcast(&value, 1, MPI_INT, 0, MPI_COMM_WORLD*)].

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:



Note:
The list of MPI data types can be found here:
https://rookiehpc.org/mpi/docs/mpi_datatype/index.html

03_Broadcast.cpp

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char **argv)
{
    int rank, value;

    // Init MPI
    MPI_Init(&argc, &argv);

    // Get process ID (rank)
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    do
    {
        if (rank == 0)
        {
            printf("Please give a number (negative number to terminate): ");
            fflush(stdout); // Force immediate printing
            scanf_s("%d", &value);
        }
        // Broadcast the input value to all the processes
        //  Root process usually 0 will send to all receiver, and receiver will receive the value variable value
        //  int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm )
        MPI_Bcast(&value, 1, MPI_INT, 0, MPI_COMM_WORLD); <- broadcast to other process

        // Eveybody (receiver) will print the result
        printf("Process %d received %d\n", rank, value);
        fflush(stdout); // Force immediate printing

        // Synchronize the processes (wait for all the
        // processes to print their values before proceeding
        // to ask for the next value.)
        MPI_Barrier(MPI_COMM_WORLD); <- barrier
        // Without barrier, process will run at different speed,
        // if the thread haven't print out the value before the thread 0, loop again, the prompt will generate firt, then other thread value then print out

    } while (value >= 0);

    printf("Process %d terminated.\n", rank);

    return 0;
}
```

output with barrier:

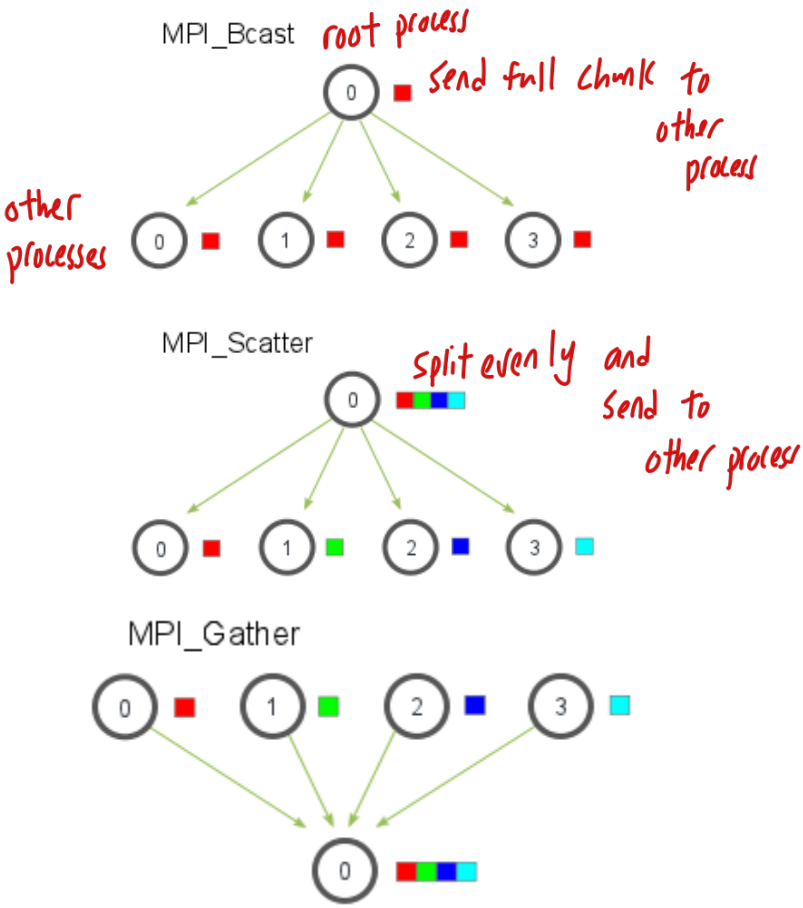
```
Please give a number (negative number to terminate): 1
Process 0 received 1
Process 1 received 1
Process 2 received 1
Process 3 received 1
Process 4 received 1
Process 5 received 1
```

output without barrier

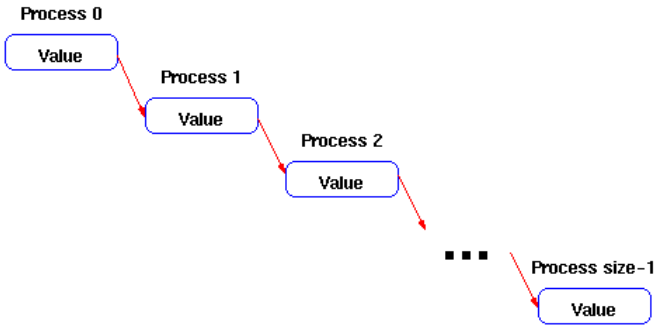
```
Please give a number (negative number to terminate): 1
Process 0 received 1
Please give a number (negative number to terminate): Process 1 received 1
Process 2 received 1
Process 3 received 1
Process 5 received 1
Process 4 received 1
2
Process 0 received 2
Please give a number (negative number to terminate): Process 1 received 2
Process 2 received 2
Process 5 received 2
Process 3 received 2
Process 4 received 2
```

↑
this print befor other thread done

P6 - Distributed Process Management



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:

04_Send_Recv.cpp

```cpp
#include <stdlib.h>
#include <iostream>
#include <mpi.h>

using namespace std;

int main(int argc, char **argv)
{
    int rank, value, size;
    MPI_Status status = {0};

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    do
    {
        if (rank == 0)
        {
            printf("Please give a number: ");
            fflush(stdout);
            scanf("%d", &value);

            // Process 0 send the message
            MPI_Send(&value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
        }
        else
        {
            // Receive from the previous rank/process ID
            MPI_Recv(&value, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, &status);

            if (rank < size - 1)
                // All ranks, except the last, send the value to
                // their immediate neighbours
                MPI_Send(&value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
        }
        printf("Process %d received %d (error code: %d)\n", rank, value,
               status.MPI_ERROR);
        fflush(stdout);
        MPI_Barrier(MPI_COMM_WORLD);
    } while (value >= 0);

    MPI_Finalize();
    return 0;
}
```

Send to next process
process 0 send process 1
receive from previous process
receive
P0 ← P1
send to next process
P1 send P2

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

Output (04_Send_Recv.cpp)

```
└─ mpirun -np 4 ./04_Send_Recv
Please give a number: 2
Process 0 received 2 (error code: 0)
Process 1 received 2 (error code: 0)
Process 2 received 2 (error code: 0)
Process 3 received 2 (error code: 0)
```

Diff between Broadcast and Send & Receive

↓

root send to many other processes

↓

root send to one process, the process send to another process until the end

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
```

https://www.mpich.org/static/docs/v3.1/www3/MPI_Scatter.html

https://www.mpich.org/static/docs/v3.3/www3/MPI_Gather.html

05_Scatter_Gather.cpp

```cpp
// argc - count
// argv - vector
int main(int argc, char **argv)
{

    cout << argc << " inputs\n";
    for (int i = 0; i < argc; i++)
    {
        cout << "[" << i << "] = " << argv[i] << endl;
    }

    if (argc != 2)
    {
        // argv[0] - name of of program EG P6
        fprintf(stderr, "Usage: %s num_elements_per_proc\n", argv[0]);
        exit(1);
    }

    // argv[1] - number EG 6
    cout << "The num_elements_per_proc is (string) " << argv[1] << endl;

    // atoi - ascii to integer
    int num_elements_per_proc = atoi(argv[1]);
    cout << "The num_elements_per_proc is " << num_elements_per_proc << endl;
```

print the argument value

if do not have 2 argument print out usage and exist

before atoi

after atoi (ascii to integer)

output:

4 process

```
└─ mpirun -np 4 ./05_Scatter_Gather 1000
2 inputs   P1
[0] = ./05_Scatter_Gather
[1] = 1000
The num_elements_per_proc is (string) 1000
The num_elements_per_proc is 1000
2 inputs   P2
[0] = ./05_Scatter_Gather
[1] = 1000
The num_elements_per_proc is (string) 1000
The num_elements_per_proc is 1000
2 inputs   P3
[0] = ./05_Scatter_Gather
[1] = 1000
The num_elements_per_proc is (string) 1000
The num_elements_per_proc is 1000
2 inputs   P4
[0] = ./05_Scatter_Gather
[1] = 1000
The num_elements_per_proc is (string) 1000
The num_elements_per_proc is 1000
```

before atoi

after atoi

```
└─ mpirun -np 4 ./05_Scatter_Gather 01000
2 inputs
[0] = ./05_Scatter_Gather
[1] = 01000
The num_elements_per_proc is (string) 01000
The num_elements_per_proc is 1000
2 inputs
[0] = ./05_Scatter_Gather
[1] = 01000
The num_elements_per_proc is (string) 01000
The num_elements_per_proc is 1000
2 inputs
[0] = ./05_Scatter_Gather
[1] = 01000
The num_elements_per_proc is (string) 01000
The num_elements_per_proc is 1000
2 inputs
[0] = ./05_Scatter_Gather
[1] = 01000
The num_elements_per_proc is (string) 01000
The num_elements_per_proc is 1000
```

before atoi

after atoi

```
└─ mpirun -np 4 ./05_Scatter_Gather abc
2 inputs
[0] = ./05_Scatter_Gather
[1] = abc
The num_elements_per_proc is (string) abc
The num_elements_per_proc is 0
2 inputs
[0] = ./05_Scatter_Gather
[1] = abc
The num_elements_per_proc is (string) abc
The num_elements_per_proc is 0
2 inputs
[0] = ./05_Scatter_Gather
[1] = abc
The num_elements_per_proc is (string) abc
The num_elements_per_proc is 0
2 inputs
[0] = ./05_Scatter_Gather
[1] = abc
The num_elements_per_proc is (string) abc
The num_elements_per_proc is 0
```

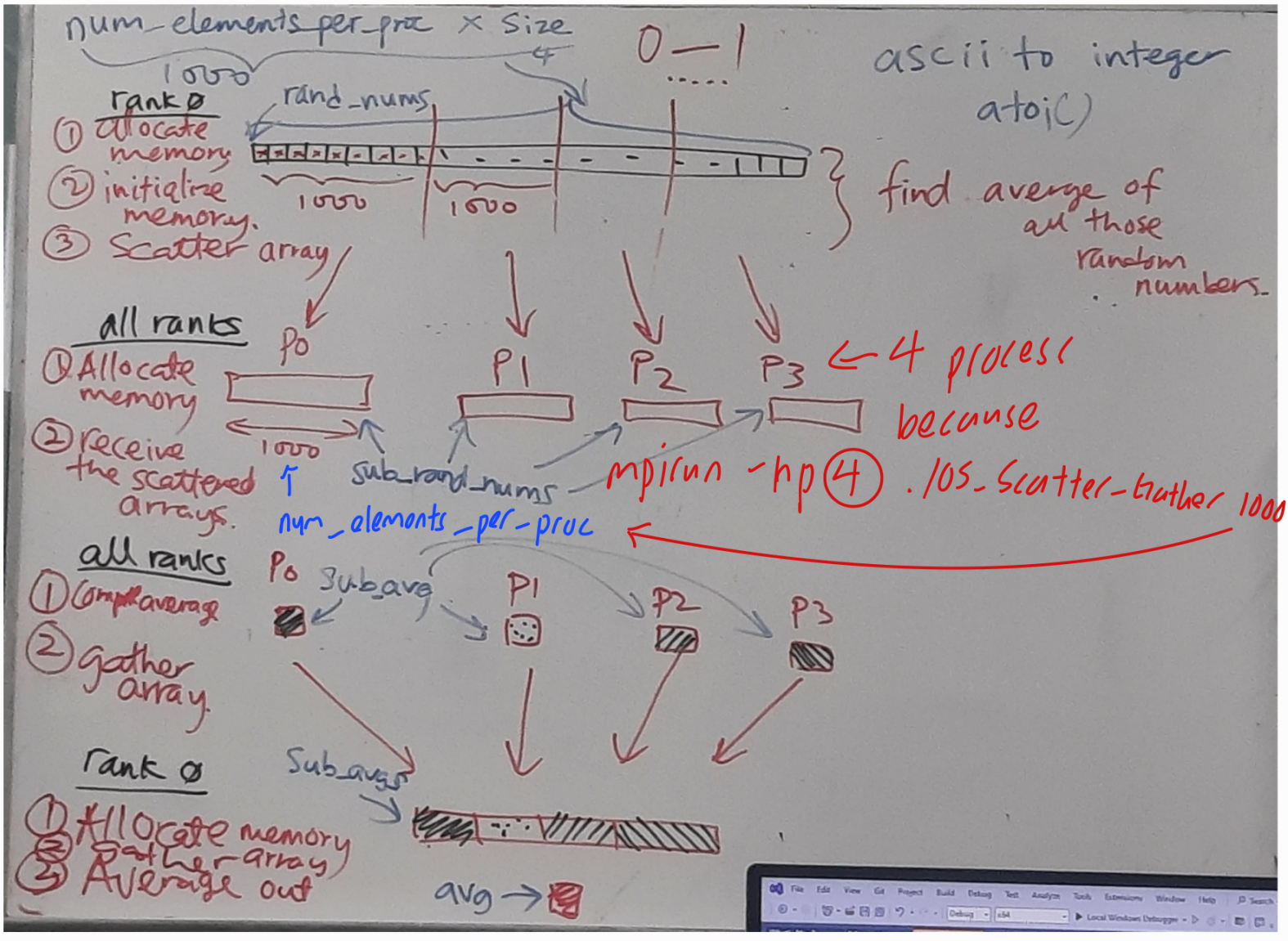
before atoi

after atoi

05_Scatter_Gather.cpp

```cpp
// Seed the random number generator to get different results each time
srand((unsigned int)time(NULL));

// cout << "Done \n";
// exit(0);

int world_rank, world_size;
MPI_Status status = {0};

MPI_Init(&argc, &argv);

// Create "world_rank" and set it to the process ID
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

// Create "world_size" and set it to number of MPI processes
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

// Create a random array of elements on the root process. Its total
// size will be the number of elements per process times the number
// of processes
double *rand_nums = NULL;
if (world_rank == 0)
{
    rand_nums = create_rand_nums(num_elements_per_proc * world_size);
}

// For each process, create a buffer that will hold a subset of the entire
// array
double *sub_rand_nums = (double *)malloc(sizeof(double) * num_elements_per_proc);
assert(sub_rand_nums != NULL); // Enough Memory or not

// Scatter the random numbers from the root process to all processes in
// the MPI world
MPI_Scatter(rand_nums, num_elements_per_proc, MPI_DOUBLE, sub_rand_nums, num_elements_per_proc, MPI_DOUBLE, 0, MPI_COMM_WORLD);

// Compute the average of your subset
double sub_avg = compute_avg(sub_rand_nums, num_elements_per_proc);

// Gather all partial averages down to the root process
double *sub_avgs = NULL;
if (world_rank == 0)
{
    sub_avgs = (double *)malloc(sizeof(double) * world_size);
    assert(sub_avgs != NULL);
}
MPI_Gather(&sub_avg, 1, MPI_DOUBLE, sub_avgs, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

// Now that we have all of the partial averages on the root, compute the
// total average of all numbers. Since we are assuming each process computed
// an average across an equal amount of elements, this computation will
// produce the correct answer.
if (world_rank == 0)
{
    double avg = compute_avg(sub_avgs, world_size);
    printf("Avg of all elements is %f\n", avg);
    // Compute the average across the original data for *comparison*
    double original_data_avg =
        compute_avg(rand_nums, num_elements_per_proc * world_size);
    printf("Avg computed across original data is %f\n", original_data_avg);
}

// Clean up
if (world_rank == 0)
{
    free(rand_nums);
    free(sub_avgs);
}
free(sub_rand_nums);

MPI_Finalize();

return 0;
```

Explanation



output:

```
Avg of all elements is 0.496010
Avg computed across original data is 0.496010
```