Lab 4: Parallel Programming

Introduction

In this lab, you will learn parallel programming using MPI (message-passing interface) and OpenMP (open multi-processing). MPI is a message-passing portable standard or a distributed-memory parallel computing platform while OpenMP is a shared-memory parallel computing platform. To make things more convenient, we will be using Google Colaboratory (Colab) which already has the needed installation prerequisites to compile and execute MPI and OpenMPI programs. So to make use of Colab for our experiments, first, you will need to write or modify the C-code in a text file in your machine. Then, upload the C-code to Colab and compile the program. After, that you can execute the program and see the results in Colab. In the first part of this lab, you will be introduced to MPI programming and on the second part we will look at implementations on OpenMP. After completing this lab, you are required to do the designated homework.

Prerequisites

1. If you finished Lab 3 (Cache Simulation), we will assume that you are already familiar on how to start and set-up Colab. If not, please go to section 2A-1 to 2A-3 of Lab 3 for the simple guide on Colab.

2. (THIS IS OPTIONAL) If you can afford to install an Ubuntu virtual machine on your computer (*if your computer has a large enough memory,16GB or more, and sufficient logical processor count, 6 or more*) go to the following sites then download and install the following. Follow the necessary procedures and instructions.

- **Virtual Machine Installer** (*use free trial if prompted*) **VMWare Workstation**: https://www.vmware.com/products/workstation-pro/workstation-pro-evaluation.html **or you can also use Oracle VirtualBox:** https://www.virtualbox.org/
- **Ubuntu** (Linux Distribution): https://ubuntu.com/download/desktop
- **MPI Installation** (tutorial and download guide): https://mpitutorial.com/tutorials/installing-mpich2/
- **OpenMP** is automatically installed in the operating system.

1. Introduction to MPI

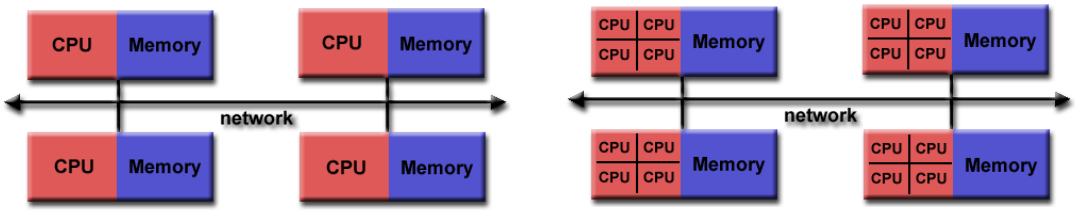
Objectives

In this part you will learn how to code, compile, and execute MPI in C-programming language. Moreover, you will learn basic MPI functions and especially how to structure your code for distributed-memory parallel programming.

A. Basic MPI Routine Functions

1. MPI (message passing interface) is a distributed memory parallel computing platform. It means that for each process or task it has its own memory space which other processes cannot directly access. Memory is not shared among processes but instead needs a communication routine such as message-passing to send or receive data from other processes. (Please take note that in this lab guide the terms process and task are used interchangeably but process can also refer to the main process that handles child processes)

Now, as architecture trends changed, shared memory SMPs were combined over networks creating hybrid distributed memory / shared memory systems , and MPI supports them all.

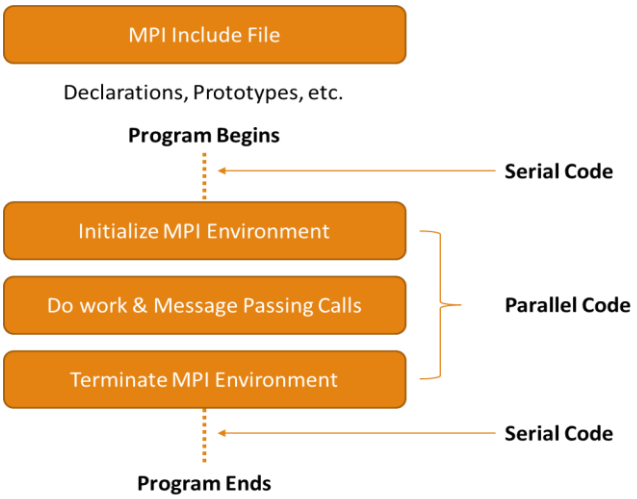


Distributed-Memory Illustration
From: https://computing.llnl.gov/tutorials/mpi/

2. MPI code structure:

For MPI, serial code means headers and initializations that are not executed in parallel. Serial set-ups and post set-ups are usually done by the master process.

Parallel Code means the main body of the parallel process using MPI functions.



3. The routine functions in MPI are used to initialized the distributed memory environment, set-up each process and communicators, and finally close the environment.

MPI Routines

- MPI_Init(&argc, &argv) – Initialize MPI environment.
- MPI_Comm_size(comm, &size) – Initialize communicator and get total number of processes.
- MPI_Comm_rank(comm, &rank) – Initialize communicator for each process and get process ID.
- MPI_Get_processor_name(&name, &length) – Get the processor assigned for each process.

- MPI_Finalize() – Close the MPI environment.

4. Let's do your first MPI program, **"**Hello World" in MPI. **Please refer to mpi_hello.c** file as attached to this Lab Guide.

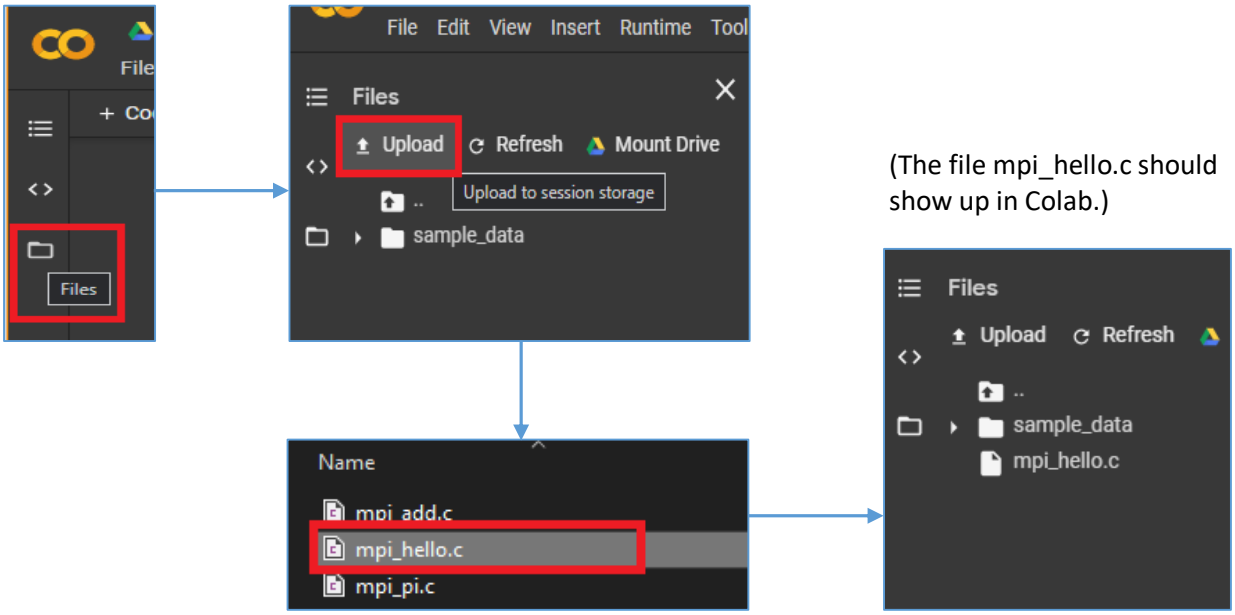
"Hello World" in MPI

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

int main(int argc, char *argv[]) {

    // Initialize MPI environment
    MPI_Init(&argc, &argv);

    // Get the number of tasks
    int numtasks;
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

    // Get the task id
    int taskid;
    MPI_Comm_rank(MPI_COMM_WORLD, &taskid);

    // Get the processor name for each task
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    printf ("Hello from processor %s, task id %d out of %d tasks\n",
            processor_name, taskid, numtasks);

    // Finalize MPI environment
    MPI_Finalize();
}
```

This code demonstrates the basic MPI routine functions.

During MPI execution using mpiexec, the code is copied to all processes where each process executes a copy of the code independently from other processes.

>> Create a new Colab notebook and upload the mpi_hello.c.



(The file mpi_hello.c should show up in Colab.)

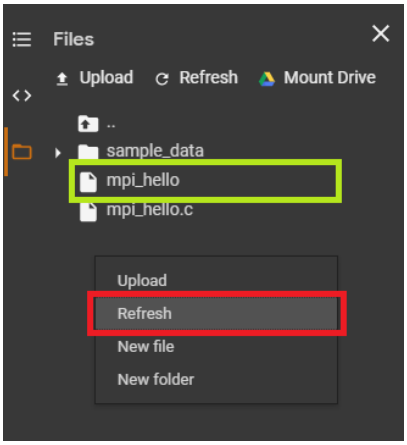
>> Compile the uploaded mpi_hello.c:

(Copy the command)

```
!mpicc -o mpi_hello mpi_hello.c
```

Successful compilation in Colab

```
[1]  1  !mpicc -o mpi_hello mpi_hello.c
```



**If there are no compilation errors no output should show-up below the cell and the output executable code should show up at the right side Files tab (if it does not show up, right click the tab and refresh).

Example of compilation error in Colab

```
[2]  1  !mpicc -o example_error example_error.c

example_error.c: In function 'main':
example_error.c:12:2: error: expected '=', ',', ';', 'asm' or '__attribute__' before 'MPI_Comm_size'
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  ^~~~~~~~~~~~~
example_error.c:12:33: error: 'numtasks' undeclared (first use in this function)
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
                                ^~~~~~~~
example_error.c:12:33: note: each undeclared identifier is reported only once for each function it appears in
```

If there are compilation errors then the output would point to the source of the errors and usually will present options to correct it. **You should make the necessary corrections in the C-code (*open in text file and edit*) and re-upload it.

Execute the code in Colab using mpiexec

(Copy the command) **[Execute in 4 processes using -n 4 option, -n <number of process>]**

```
!mpiexec --allow-run-as-root -n 4 ./mpi_hello
```

Execution output:

```
[3]  1  !mpiexec --allow-run-as-root -n 4 ./mpi_hello

Hello from processor 6c7c275d4dd7, task id 2 out of 4 tasks
Hello from processor 6c7c275d4dd7, task id 0 out of 4 tasks
Hello from processor 6c7c275d4dd7, task id 1 out of 4 tasks
Hello from processor 6c7c275d4dd7, task id 3 out of 4 tasks
```

**Note: to execute mpiexec in Colab you need --allow-run-as-root option but if you execute it in the virtual machine installed in your computer, there is no need for it.

B. MPI Send and Receive Functions

1. The Send and Receive functions are point-to-point functions that is used to pass data to other processes.

MPI_Send – Performs a blocking send

Synopsis:
int **MPI_Send**(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

Input Parameters:
buf - initial address of send buffer (choice)
count - number of elements in send buffer (nonnegative integer)
datatype - datatype of each send buffer element (handle)
dest - rank or id of destination (integer)
tag - message tag (integer)
comm - communicator (handle)

MPI_Recv – Blocking receive for a message

Synopsis:
int **MPI_Recv**(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status * status)

Output Parameters:
buf - initial address of receive buffer (choice)
status - status object (Status)

Input Parameters:
count - maximum number of elements in receive buffer (integer)
datatype - datatype of each receive buffer element (handle)
source – rank or id of source (integer)
tag- message tag (integer)
comm -communicator (handle)

From: https://www.mpich.org/static/docs/

2. During send or receive of data, the data type must be specified to allocate the needed memory space. The following are some of the elementary data types in MPI:

MPI Datatype	C Equivalent
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

3. Let's do an example of send-receive message-passing example. **Please refer to mpi_add.c** file as attached to this Lab Guide. The code mpi_add.c demonstrates addition of all the elements in a large array. It also demonstrates comparison of serial computation time and parallel computation time.

Pseudocode of parallel solution to addition of elements in array

```
find out if I am MASTER or WORKER

if I am MASTER

  initialize the array
  send each WORKER info on part of array it owns
  send each WORKER its portion of initial array

  # calculate my portion of array
  do j = my first column,my last column
    do i = 1,n
      a(i,j) = fcn(i,j)
    end do
  end do

  receive from each WORKER results

else if I am WORKER
  receive from MASTER info on part of array I own
  receive from MASTER my portion of initial array

  # calculate my portion of array
  do j = my first column,my last column
    do i = 1,n
      a(i,j) = fcn(i,j)
    end do
  end do

  send MASTER results

endif
```

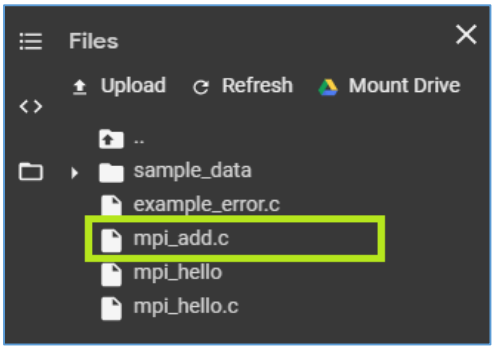
From: https://computing.llnl.gov/tutorials/parallel_comp/#ExamplesArray

**Note: In the implementation of mpi_add.c, serial computation is also done for execution time comparison. The serial execution is done by the MASTER after initializing the array.

**Measuring the execution time: To properly measure the parallel execution time, put the start_time() function only after all serial computation or serial overhead is done and the end_time() function before any proceeding serial executions. Similarly in serial timing, measure only the execution time of computation block of the serial process.

****Refer to how the mpi_add.c code is structured.**

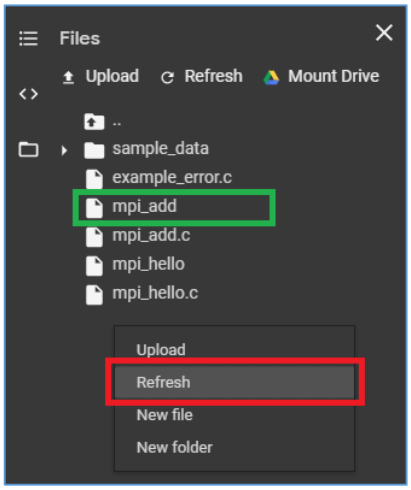
4. Upload mpi_add.c to Colab. Make sure that you can see the file at the right side Files tab.



>> Then **compile** the uploaded **mpi_add.c**:
(Copy the command)

```
!mpicc -o mpi_add mpi_add.c
```

Successful compilation




```
[5]  1  !mpicc -o mpi_add mpi_add.c
```

**Make sure that compilation is successful. There should be no output for the compilation cell and you can see the mpi_add output file added to the right side Files tab (*refresh when not seen immediately*).

5. Now let us **execute mpi_add with 1, 2, 4, and 8 processes**. Please take note that Colab has at most 2 virtual cores which means that parallel speed-up (Serial wall time / Parallel wall time) should not deviate so much to 2.0 (*sometimes it will exceed by a small percentage because of deviations in the internal timing calculations*). (You can see how many cores are in Colab server by giving "*!cat /proc/cpuinfo"* in notebook.)

(Copy the command) **[Execute with 1 process for parallel code]**

```
!mpiexec --allow-run-as-root -n 1 ./mpi_add
```

Execution output:

```
[6]  1  !mpiexec --allow-run-as-root -n 1 ./mpi_add

MPI task 0 has started...

*** Serial sum = 5.000000e+15

>> Serial time = 0.348552 seconds

Task 0 sum = 5.000000e+15

*** Parallel sum = 5.000000e+15

>> Parallel time = 0.354404 seconds
```

Speed-up = (Serial wall time / Parallel wall time)
Speed-up = 0.9835 ≈ 1
**No speed-up

(Copy the command) **[Execute with 2 processes for parallel code]**

```
!mpiexec --allow-run-as-root -n 2 ./mpi_add
```

Execution output:

```
[7]  1  !mpiexec --allow-run-as-root -n 2 ./mpi_add

MPI task 0 has started...
MPI task 1 has started...

*** Serial sum = 5.000000e+15

>> Serial time = 0.390911 seconds

Task 1 sum = 3.750000e+15
Task 0 sum = 1.250000e+15

*** Parallel sum = 5.000000e+15

>> Parallel time = 1.752518 seconds
```

Speed-up = (Serial wall time / Parallel wall time)
Speed-up = 0.2231
**No speed-up

(Copy the command) **[Execute with 4 processes for parallel code]**

```
!mpiexec --allow-run-as-root -n 4 ./mpi_add
```

Execution output:

```
[8]  1  !mpiexec --allow-run-as-root -n 4 ./mpi_add

MPI task 0 has started...
MPI task 3 has started...
MPI task 2 has started...
MPI task 1 has started...

*** Serial sum = 5.000000e+15

>> Serial time = 0.386858 seconds

Task 1 sum = 9.375000e+14
Task 2 sum = 1.562500e+15
Task 0 sum = 3.125000e+14
Task 3 sum = 2.187500e+15

*** Parallel sum = 5.000000e+15

>> Parallel time = 3.149228 seconds
```

Speed-up = (Serial wall time / Parallel wall time)
Speed-up = 0.1228
**No speed-up

(Copy the command) **[Execute with 8 processes for parallel code]**

```
!mpiexec --allow-run-as-root -n 8 ./mpi_add
```

Execution output:

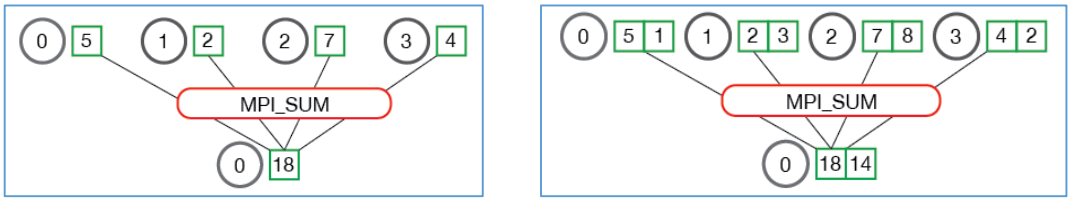
```
[9]  1  !mpiexec --allow-run-as-root -n 8 ./mpi_add

MPI task 4 has started...
MPI task 7 has started...
MPI task 6 has started...
MPI task 5 has started...
MPI task 3 has started...
MPI task 0 has started...
MPI task 1 has started...
MPI task 2 has started...

*** Serial sum = 5.000000e+15

>> Serial time = 0.388608 seconds

Task 1 sum = 2.343750e+14
Task 2 sum = 3.906250e+14
Task 3 sum = 5.468750e+14
Task 4 sum = 7.031250e+14
Task 5 sum = 8.593750e+14
Task 6 sum = 1.015625e+15
Task 7 sum = 1.171875e+15
Task 0 sum = 7.812499e+13

*** Parallel sum = 5.000000e+15

>> Parallel time = 6.029414 seconds
```

Speed-up = (Serial wall time / Parallel wall time)
Speed-up = 0.0645
**No speed-up

6. DISCUSSION – When we execute the code by increasing number of processes (1, 2, 4, and 8), we can observe that there is no speed-up (speed-up < 1.0) for any number of process. This is because the send-receive overhead for each process takes a lot of time than the actual addition operation. This means that the nature of the problem can be efficiently done in serial computation and it is not a practical problem for distributed-memory parallel computing.

C. MPI Reduce and Barrier Functions

1. MPI_Reduce is a collective communication scheme that performs an operation, e.g. Sum, Multiply, across all the common data containers for each process and then sends it to 1 destination process.



MPI_Reduce Operation Illustration
From: http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

MPI_Reduce – Reduces values on all processes to a single value

Synopsis:
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
MPI_Op op, int root, MPI_Comm comm)
Input Parameters:
sendbuf - address of send buffer (choice)
count - number of elements in send buffer (integer)
datatype - data type of elements of send buffer (handle)
op - reduce operation (handle)
root - rank of root process (integer)
comm - communicator (handle)

Output Parameters:
recvbuf - address of receive buffer (choice, significant only at root)

From: https://www.mpich.org/static/docs/

MPI Predefined Reduce Operations
[MPI_MAX] maximum
[MPI_MIN] minimum
[MPI_SUM] sum
[MPI_PROD] product
[MPI_LAND] logical and
[MPI_BAND] bit-wise and
[MPI_LOR] logical or
[MPI_BOR] bit-wise or
[MPI_LXOR] logical xor
[MPI_BXOR] bit-wise xor
[MPI_MAXLOC] max value and location
[MPI_MINLOC] min value and location

2. MPI_Barrier is a "fence" operation that blocks all processes from entering the other side of the "fence" until all processes reached the same part of the execution

MPI_Barrier – Blocks until all processes in the communicator have reached this routine.

Synopsis:
int MPI_Barrier(MPI_Comm comm)

Input Parameters:
comm - communicator (handle)

3. Let's do an example that uses MPI_Reduce and MPI_Barrier example. **Please refer to mpi_pi.c** file as attached to this Lab Guide. The code mpi_pi.c demonstrates calculation of the constant Pi using Monte Carlo approximation method. In the parallel solution, MPI_Reduce is used to sum all Pi estimates from all processes then sends the result to the MASTER.

General Algorithm:

PI Calculation

- The value of PI can be calculated in various ways. Consider the Monte Carlo method of approximating PI:
  - Inscribe a circle with radius **r** in a square with side length of **2*r***
  - The area of the circle is Πr^2 and the area of the square is $4r^2$
  - The ratio of the area of the circle to the area of the square is:
    $\Pi r^2 / 4r^2 = \Pi / 4$
  - If you randomly generate **N** points inside the square, approximately
    N * **Π** / **4** of those points (**M**) should fall inside the circle.
  - **Π** is then approximated as:
    N * **Π** / **4** = **M**
    Π / **4** = **M** / **N**
    Π = **4** * **M** / **N**
  - Note that increasing the number of points generated improves the approximation.

From: https://computing.llnl.gov/tutorials/parallel_comp/#ExamplesPI

Pseudocode of parallel solution for Calculating Pi

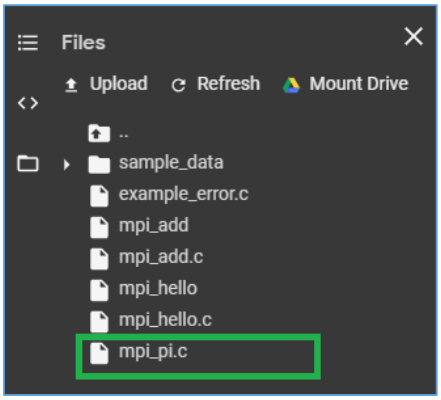
```
npoints = more than 100000
circle_count = 0

p = number of tasks
num = npoints/p

find out if I am MASTER or WORKER

do j = 1,num
  generate 2 random numbers between 0 and 1
  xcoordinate = random1
  ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

if I am MASTER

  receive from WORKERS their circle_counts
  compute PI (use MASTER and WORKER calculations)

else if I am WORKER

  send to MASTER circle_count

endif
```

**Note: In the implementation of mpi_pi.c, serial computation is also done for execution time comparison. The serial execution is done by the MASTER before executing parallel operation.

**MPI_Barrier is used to synchronized the MASTER and WORKERs during parallel execution. This is because when the MASTER is executing the serial process, the WORKERs can start sampling without Barrier function (there is no data dependency in random sampling). Remember that each process runs the same lines of code respectively to their IDs.

****Refer to how the mpi_pi.c code is structured.**

4. Upload mpi_pi.c to Colab. Make sure that you can see the file at the right side Files tab.



>> Then **compile** the uploaded **mpi_pi.c**:

(Copy the command)

```
!mpicc -o mpi_pi mpi_pi.c
```

>> After successful compilation let us now **execute mpi_pi with 1, 2, 4, and 8 processes**. In the implementation of mpi_pi.c, the number of processes directly corresponds to the total number of points (total_points = num_process * num_points_per_process). So the serial process has to loop over the total_points. In the Monte Carlo approximation method of calculating Pi, the more number of points, the closer to the estimate to the real value (*in mpi_pi.c -> num_points_per_process = 200000*).

(Copy the command) **[Execute with 1 process]**

```
!mpiexec --allow-run-as-root -n 1 ./mpi_pi
```

Execution output:

```
[17]  1  !mpiexec --allow-run-as-root -n 1 ./mpi_pi

MPI task 0 has started...

***Start serial process...
   After  20000000 points, average value of pi = 3.14251000
   After  40000000 points, average value of pi = 3.14240160
   After  60000000 points, average value of pi = 3.14202427
   After  80000000 points, average value of pi = 3.14185760
   After 100000000 points, average value of pi = 3.14174484
   After 120000000 points, average value of pi = 3.14166450
   After 140000000 points, average value of pi = 3.14167454
   After 160000000 points, average value of pi = 3.14167757
   After 180000000 points, average value of pi = 3.14171062
   After 200000000 points, average value of pi = 3.14168640

>> Serial time = 8.189066 seconds

***Start parallel process...
   After  20000000 points, average value of pi = 3.14156900
   After  40000000 points, average value of pi = 3.14155020
   After  60000000 points, average value of pi = 3.14157567
   After  80000000 points, average value of pi = 3.14160155
   After 100000000 points, average value of pi = 3.14168984
   After 120000000 points, average value of pi = 3.14172503
   After 140000000 points, average value of pi = 3.14162454
   After 160000000 points, average value of pi = 3.14158708
   After 180000000 points, average value of pi = 3.14155322
   After 200000000 points, average value of pi = 3.14159552

>> Parallel time = 8.167368 seconds

Real value of PI: 3.1415926535897
```

Speed-up = (Serial wall time / Parallel wall time)
Speed-up = 1.0
**No speed-up

(Copy the command) **[Execute with 2 processes]**

```
!mpiexec --allow-run-as-root -n 2 ./mpi_pi
```

Execution output:

```
[18]  1  !mpiexec --allow-run-as-root -n 2 ./mpi_pi

MPI task 0 has started...
MPI task 1 has started...

***Start serial process...
   After  40000000 points, average value of pi = 3.14240160
   After  80000000 points, average value of pi = 3.14185760
   After 120000000 points, average value of pi = 3.14166450
   After 160000000 points, average value of pi = 3.14167758
   After 200000000 points, average value of pi = 3.14168640
   After 240000000 points, average value of pi = 3.14166370
   After 280000000 points, average value of pi = 3.14166216
   After 320000000 points, average value of pi = 3.14170089
   After 360000000 points, average value of pi = 3.14164226
   After 400000000 points, average value of pi = 3.14164096

>> Serial time = 20.799177 seconds

***Start parallel process...
   After  40000000 points, average value of pi = 3.14158640
   After  80000000 points, average value of pi = 3.14153695
   After 120000000 points, average value of pi = 3.14142287
   After 160000000 points, average value of pi = 3.14138850
   After 200000000 points, average value of pi = 3.14147908
   After 240000000 points, average value of pi = 3.14146203
   After 280000000 points, average value of pi = 3.14149110
   After 320000000 points, average value of pi = 3.14152935
   After 360000000 points, average value of pi = 3.14155693
   After 400000000 points, average value of pi = 3.14152263

>> Parallel time = 12.767032 seconds

Real value of PI: 3.1415926535897
```

Speed-up = (Serial wall time / Parallel wall time)
Speed-up = 1.6291
**There is speed-up below 2.0x

(Copy the command) **[Execute with 4 processes]**

```
!mpiexec --allow-run-as-root -n 4 ./mpi_pi
```

Execution output:

```
[19]  1  !mpiexec --allow-run-as-root -n 4 ./mpi_pi

MPI task 1 has started...
MPI task 2 has started...
MPI task 3 has started...
MPI task 0 has started...

***Start serial process...
   After  80000000 points, average value of pi = 3.14185760
   After 160000000 points, average value of pi = 3.14167758
   After 240000000 points, average value of pi = 3.14166370
   After 320000000 points, average value of pi = 3.14170089
   After 400000000 points, average value of pi = 3.14164096
   After 480000000 points, average value of pi = 3.14152070
   After 560000000 points, average value of pi = 3.14156672
   After 640000000 points, average value of pi = 3.14153928
   After 720000000 points, average value of pi = 3.14156556
   After 800000000 points, average value of pi = 3.14156810

>> Serial time = 41.345524 seconds

***Start parallel process...
   After  80000000 points, average value of pi = 3.14160930
   After 160000000 points, average value of pi = 3.14186647
   After 240000000 points, average value of pi = 3.14173912
   After 320000000 points, average value of pi = 3.14171441
   After 400000000 points, average value of pi = 3.14164677
   After 480000000 points, average value of pi = 3.14158933
   After 560000000 points, average value of pi = 3.14162116
   After 640000000 points, average value of pi = 3.14164055
   After 720000000 points, average value of pi = 3.14164307
   After 800000000 points, average value of pi = 3.14162965

>> Parallel time = 26.084398 seconds

Real value of PI: 3.1415926535897
```

Speed-up = (Serial wall time / Parallel wall time)
Speed-up = 1.5851
**There is speed-up below 2.0x

(Copy the command) **[Execute with 8 processes]**

```
!mpiexec --allow-run-as-root -n 8 ./mpi_pi
```

Execution output:

```
[20]  1  !mpiexec --allow-run-as-root -n 8 ./mpi_pi

MPI task 2 has started...
MPI task 4 has started...
MPI task 6 has started...
MPI task 3 has started...
MPI task 7 has started...
MPI task 1 has started...
MPI task 5 has started...
MPI task 0 has started...

***Start serial process...
   After 160000000 points, average value of pi = 3.14167758
   After 320000000 points, average value of pi = 3.14170089
   After 480000000 points, average value of pi = 3.14152070
   After 640000000 points, average value of pi = 3.14153928
   After 800000000 points, average value of pi = 3.14156810
   After 960000000 points, average value of pi = 3.14158770
   After 1120000000 points, average value of pi = 3.14159036
   After 1280000000 points, average value of pi = 3.14159242
   After 1440000000 points, average value of pi = 3.14156584
   After 1600000000 points, average value of pi = 3.14154664

>> Serial time = 86.763237 seconds

***Start parallel process...
   After 160000000 points, average value of pi = 3.14154018
   After 320000000 points, average value of pi = 3.14161297
   After 480000000 points, average value of pi = 3.14158497
   After 640000000 points, average value of pi = 3.14159474
   After 800000000 points, average value of pi = 3.14156672
   After 960000000 points, average value of pi = 3.14157282
   After 1120000000 points, average value of pi = 3.14156934
   After 1280000000 points, average value of pi = 3.14155989
   After 1440000000 points, average value of pi = 3.14159207
   After 1600000000 points, average value of pi = 3.14160690

>> Parallel time = 55.608028 seconds

Real value of PI: 3.1415926535897
```

Speed-up = (Serial wall time / Parallel wall time)
Speed-up = 1.5602
**There is speed-up below 2.0x

EXAMPLE OUTPUT FROM UBUNTU VIRTUAL MACHINE WITH 8 LOGICAL CORES

- Executing with 4 Processes

```
(base) arczer0@VinceVirtual:~/Documents/MPI$ mpiexec -n 4 ./mpi_pi
MPI task 1 has started...
MPI task 2 has started...
MPI task 3 has started...
MPI task 0 has started...

***Start serial process...
   After  80000000 points, average value of pi = 3.14185760
   After 160000000 points, average value of pi = 3.14167758
   After 240000000 points, average value of pi = 3.14166370
   After 320000000 points, average value of pi = 3.14170089
   After 400000000 points, average value of pi = 3.14164096
   After 480000000 points, average value of pi = 3.14152070
   After 560000000 points, average value of pi = 3.14156672
   After 640000000 points, average value of pi = 3.14153928
   After 720000000 points, average value of pi = 3.14156556
   After 800000000 points, average value of pi = 3.14156810

>> Serial time = 18.825493 seconds

***Start parallel process...
   After  80000000 points, average value of pi = 3.14160930
   After 160000000 points, average value of pi = 3.14186647
   After 240000000 points, average value of pi = 3.14173912
   After 320000000 points, average value of pi = 3.14171441
   After 400000000 points, average value of pi = 3.14164677
   After 480000000 points, average value of pi = 3.14158933
   After 560000000 points, average value of pi = 3.14162116
   After 640000000 points, average value of pi = 3.14164055
   After 720000000 points, average value of pi = 3.14164307
   After 800000000 points, average value of pi = 3.14162965

>> Parallel time = 4.694302 seconds
```

- Speed-up = 4.0x

- Executing with 8 Processes

```
(base) arczer0@VinceVirtual:~/Documents/MPI$ mpiexec -n 8 ./mpi_pi
MPI task 0 has started...
MPI task 3 has started...
MPI task 6 has started...
MPI task 7 has started...
MPI task 5 has started...
MPI task 2 has started...
MPI task 1 has started...
MPI task 4 has started...

***Start serial process...
   After 160000000 points, average value of pi = 3.14167758
   After 320000000 points, average value of pi = 3.14170089
   After 480000000 points, average value of pi = 3.14152070
   After 640000000 points, average value of pi = 3.14153928
   After 800000000 points, average value of pi = 3.14156810
   After 960000000 points, average value of pi = 3.14158770
   After 1120000000 points, average value of pi = 3.14159036
   After 1280000000 points, average value of pi = 3.14159242
   After 1440000000 points, average value of pi = 3.14156584
   After 1600000000 points, average value of pi = 3.14154664

>> Serial time = 42.579639 seconds

***Start parallel process...
   After 160000000 points, average value of pi = 3.14154018
   After 320000000 points, average value of pi = 3.14161297
   After 480000000 points, average value of pi = 3.14158497
   After 640000000 points, average value of pi = 3.14159474
   After 800000000 points, average value of pi = 3.14156672
   After 960000000 points, average value of pi = 3.14157282
   After 1120000000 points, average value of pi = 3.14156934
   After 1280000000 points, average value of pi = 3.14155989
   After 1440000000 points, average value of pi = 3.14159207
   After 1600000000 points, average value of pi = 3.14160690

>> Parallel time = 5.686698 seconds
```

- Speed-up = 7.49x

5. DISCUSSION – Executing this computation method with more than 1 process yields a speed-up greater than 1.0 but less than 2.0. This is exactly predictable since the number of virtual processors in Colab is only 2. The same line of observation can be seen for the examples executed in 8-logical core virtual machine that the maximum theoretical speed-up is equal to the number of logical cores. Thus, the speed-up should not be greater than the number of processors. In Colab, when we increase the processes to more than the number of virtual processors, we can observe a slight decrease in speed-up. This is because if there are more processes than processors, 2 or more processes can be scheduled to run in a single processor which decreases the computation efficiency.

2. Introduction to OpenMP

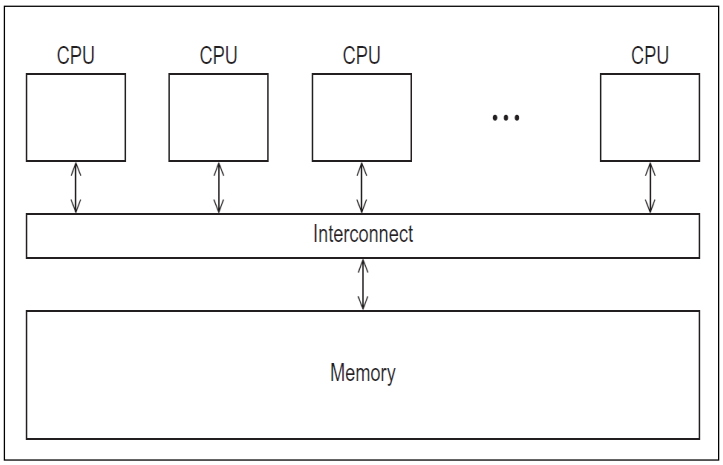
Objectives

In this part you will learn how to code, compile, and execute OpenMP in C-programming language. Moreover, you will learn basic OpenMP directives for shared-memory parallel programming and how to structure your code for shared-memory parallel programming.

A. Hello World in OpenMP

1. OpenMP is a shared-memory parallel computing platform. It means that each thread can access the same memory locations unlike distributed-memory in which processes only communicate via message-passing. In OpenMP, the programmer simply states that a block of code should be executed in parallel. The precise determination of the tasks and which thread to be executed is left to the compiler and the run-time system.

In order to state that the block of code should be executed in parallel, OpenMP uses a "directives-based" specification in its API. In C-language Pragma directive (#pragma) is a special preprocessor instruction that helps us to specify behaviors that aren't part of the basic C language. In which case, helps us specify which block of code to be executed in parallel and the specifications of its parallelization.



Shared-Memory Illustration

2. Let's do your first OpenMP program, **"**Hello World" in OpenMP. **Please refer to omp_hello.c** file as attached to this Lab Guide.

"Hello World" in OpenMP

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

// Function prototype
void hello(void);

int main(int argc, char *argv[]) {

    // Get number of threads from the command line
    int thread_count = strtol(argv[1], NULL, 10);

    // OpenMP parallel threads using pragma directive
    #pragma omp parallel num_threads(thread_count)
    hello();

    return 0;
}

void hello(void) {
    // Get parameters from OpenMP function calls
    int thread_id = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("Hello from thread %d of %d\n", thread_id, thread_count);
}
```

The **hello()** function is specified to be executed in parallel by the #pragma directive for omp parallel threads with "thread_count" number of threads.

Syntax for OpenMP parallel thread directive:
#pragma omp parallel num_threads(<*specify number of threads*>)

3. Upload omp_hello.c to Colab and then **compile it**.

(Copy the command)

```
!gcc -fopenmp -o omp_hello omp_hello.c
```

>> Make sure that compilation is successful then **execute omp_hello.**

(Copy the command) **[Execute with 4 threads]**

```
!./omp_hello 4
```

Execution output:

```
[2]   1   !./omp_hello 4

      Hello from thread 2 of 4
      Hello from thread 3 of 4
      Hello from thread 0 of 4
      Hello from thread 1 of 4
```

B. Summing Elements of Array in OpenMP

1. In the MPI section 1-B-3, parallelization of summing elements or array was demonstrated using send and receive message-passing method. As we have experimented, this method of distributed-memory parallel computing is not efficient for the given problem because of the headers associated to distribute and gather the data. But, in shared-memory parallel computing, there is no such need to distribute the

data because all the parallel threads can readily access it without any additional headers involved (also no critical section).

For OpenMP, instead we will use a parallel for loop with reduction. The parallel threads will work on equal amount of data. The compiler will be the one to assign the starting index and ending index for each thread. Then after the parallel addition for each thread, the sum of each thread will be reduced by the sum operator and will be stored in the designated variable

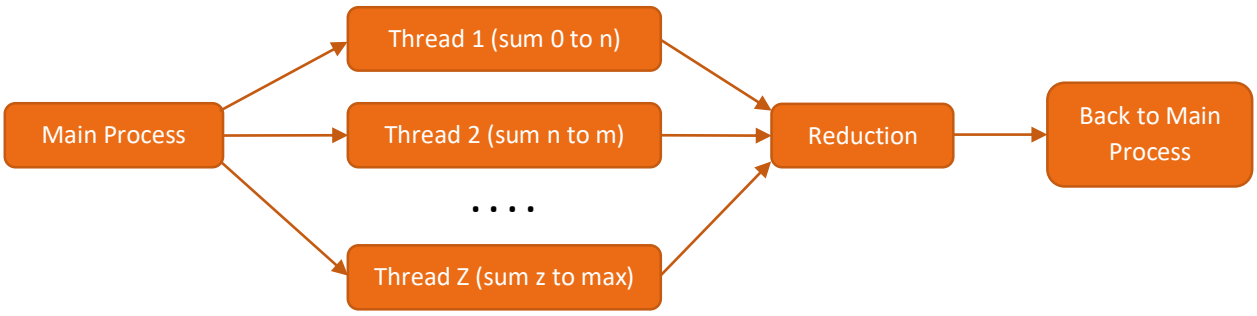


Illustration of Thread Distribution then Reduction

OpenMPof parallel solution to addition of elements in array

```
// Execute parallel addition and reduction using OpenMP pragma directive
double parallel_sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
    reduction(+: parallel_sum) shared(data)
for (int i = 0; i < ARRAYSIZE; i++) {
    parallel_sum = parallel_sum + data[i];
}
```

Syntax for OpenMP parallel FOR directive with reduction operation and shared parameters:
#pragma omp parallel for num_threads(<*specify number of threads*>) \
Reduction(<*operation*> : <*destination*>) shared(<*parameters*>)

- The FOR loop is specified to be executed in "thread_count" number of threads.
- Reduction uses addition operation (+) then the result is stored in parallel_sum variable.
- The parameter "data" is shared, that means all threads has common copy of it.

2. Upload omp_add.c to Colab and then **compile it**.

(Copy the command)

```
!gcc -fopenmp -o omp_hello omp_hello.c
```

>> After successful compilation let us now **execute omp_add with 1, 2, 4, and 8 threads**.
(Copy the command) **[Execute with 1 thread]**

```
!./omp_add 1
```

Execution output:

```
[4]  1  !./omp_add 1

*** Serial sum = 5.000000e+15
>> Serial time = 0.343899 seconds

*** Parallel sum = 5.000000e+15
>> Parallel time = 0.348209 seconds
```

Speed-up = (Serial wall time / Parallel wall time)
Speed-up = 0.9876 ≈ 1.0
**No speed-up

(Copy the command) **[Execute with 2 threads]**

```
!./omp_add 2
```

Execution output:

```
[5]  1  !./omp_add 2

*** Serial sum = 5.000000e+15
>> Serial time = 0.345283 seconds

*** Parallel sum = 5.000000e+15
>> Parallel time = 0.188840 seconds
```

Speed-up = (Serial wall time / Parallel wall time)
Speed-up = 1.8284
**There is speed-up below 2.0x

(Copy the command) **[Execute with 4 threads]**

```
!./omp_add 4
```

Execution output:

```
[6]  1  !./omp_add 4

*** Serial sum = 5.000000e+15
>> Serial time = 0.347458 seconds

*** Parallel sum = 5.000000e+15
>> Parallel time = 0.210519 seconds
```

Speed-up = (Serial wall time / Parallel wall time)
Speed-up = 1.6505
**There is speed-up below 2.0x

(Copy the command) **[Execute with 4 threads]**

```
!./omp_add 4
```

Execution output:

```
[7]  1  !./omp_add 8

*** Serial sum = 5.000000e+15
>> Serial time = 0.349675 seconds

*** Parallel sum = 5.000000e+15
>> Parallel time = 0.220906 seconds
```

Speed-up = (Serial wall time / Parallel wall time)
Speed-up = 1.5829
**There is speed-up below 2.0x

3. DISCUSSION – In the MPI approach in section 1-B-5, we observed that because of the headers involved in message-passing there is no speed-up. In OpenMP shared-memory approach, we can observe that there is a speed-up for 2, 4, and 8 threads. Also because Colab has only 2 virtual processors, its maximum speed-up can be attained for 2 threads. Executing it in more than 2 threads will result in gradually decreasing speed-up.

C. Parallel Calculation of Pi in OpenMP

1. In the MPI section 1-C-3, parallelization of the Monte Carlo approximation method of calculating Pi is done in a distributed-memory scheme. As we have experimented, distributed-memory is well suited for that problem because each sampling method does not depend on a shared data. In OpenMP, the same strategy will used to parallelized the Pi calculation process, that sampling will be done for each thread. We will parallelize the FOR loop involving the sampling method:

OpenMP parallel solution for Calculating Pi

```
ave_pi = 0.0; // Re-initialize variable
for (int i = 0; i < ROUNDS; i++) {

    // Execute parallel function call and reduction using OpenMP pragma directive
    sum_pi = 0.0;
    # pragma omp parallel for num_threads(thread_count) \
        reduction(+: sum_pi) private(proc_pi)
    for (int task = 0; task < thread_count; task++) {
        // Compute Pi per task distribution using the Monte Carlo estimation
        proc_pi = compute_pi(POINTS);
        sum_pi = sum_pi + proc_pi;
    }

    pi = sum_pi / thread_count;
    ave_pi = ((ave_pi * i) + pi) / (i + 1); // Compute moving average
    if ((i + 1) % 100 == 0) // Show result per 100 iteration
        printf("   After %9d points, average value of pi = %10.8f\n",
            (POINTS * thread_count * (i + 1)), ave_pi);
}
```

In addition of array elements in OpenMP section 2-B-1, we used a parallel for directive with reduction for shared parameters. For the solution of calculating Pi in parallel, we will use the same parallel FOR with reduction but with private parameters.

Syntax for OpenMP parallel FOR directive with reduction operation and private parameters:
#pragma omp parallel for num_threads(<*specify number of threads*>) \
Reduction(<*operation*> : <*destination*>) private(<*parameters*>)

- Private parameter means that each thread will have its own independent copy of the specified parameters

It is also possible that a computation has both shared and private parameters.
Syntax for OpenMP parallel FOR directive with reduction operation and with both shared and private parameters.
#pragma omp parallel for num_threads(<*specify number of threads*>) \
Reduction(<*operation*> : <*destination*>) shared(<*parameters*>) private(<*parameters*>)

3. Upload omp_pi.c to Colab and then **compile it**.

(Copy the command)

```
!gcc -fopenmp -o omp_pi omp_pi.c
```

>> After successful compilation let us now **execute omp_pi with 1, 2, 4, and 8 threads**.

(Copy the command) **[Execute with 1 thread]**

```
!./omp_pi 1
```

Execution output:

```
[9]  1  !./omp_pi 1

***Start serial process...
   After  20000000 points, average value of pi = 3.14128800
   After  40000000 points, average value of pi = 3.14169580
   After  60000000 points, average value of pi = 3.14161840
   After  80000000 points, average value of pi = 3.14175115
   After 100000000 points, average value of pi = 3.14172628
   After 120000000 points, average value of pi = 3.14162623
   After 140000000 points, average value of pi = 3.14165386
   After 160000000 points, average value of pi = 3.14160998
   After 180000000 points, average value of pi = 3.14166567
   After 200000000 points, average value of pi = 3.14168416
>> Serial time = 6.799687 seconds

***Start parallel process...
   After  20000000 points, average value of pi = 3.14100340
   After  40000000 points, average value of pi = 3.14175910
   After  60000000 points, average value of pi = 3.14172467
   After  80000000 points, average value of pi = 3.14164670
   After 100000000 points, average value of pi = 3.14156964
   After 120000000 points, average value of pi = 3.14145400
   After 140000000 points, average value of pi = 3.14152126
   After 160000000 points, average value of pi = 3.14158892
   After 180000000 points, average value of pi = 3.14155911
   After 200000000 points, average value of pi = 3.14153940
>> Parallel time = 6.756147 seconds

Real value of PI: 3.1415926535897
```

Speed-up = (Serial wall time / Parallel wall time)
Speed-up = 1.0
**No speed-up

(Copy the command) **[Execute with 2 threads]**

```
!./omp_pi 2
```

Execution output:

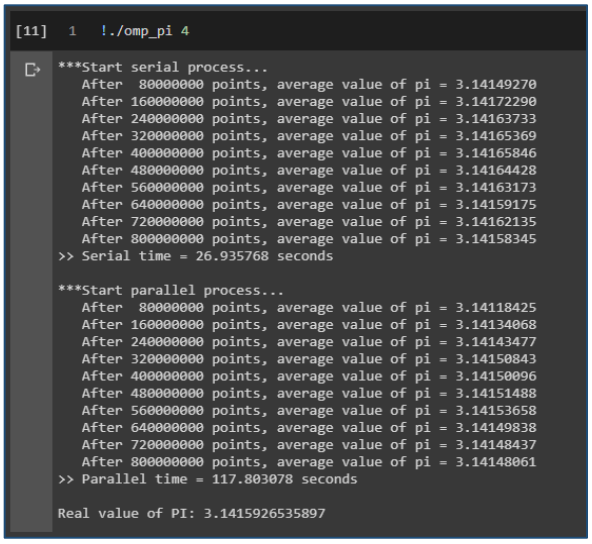
```
[10]  1  !./omp_pi 2

***Start serial process...
   After  40000000 points, average value of pi = 3.14175910
   After  80000000 points, average value of pi = 3.14164670
   After 120000000 points, average value of pi = 3.14145400
   After 160000000 points, average value of pi = 3.14158892
   After 200000000 points, average value of pi = 3.14153940
   After 240000000 points, average value of pi = 3.14150277
   After 280000000 points, average value of pi = 3.14148111
   After 320000000 points, average value of pi = 3.14150916
   After 360000000 points, average value of pi = 3.14152924
   After 400000000 points, average value of pi = 3.14153247
>> Serial time = 13.545638 seconds

***Start parallel process...
   After  40000000 points, average value of pi = 3.14167010
   After  80000000 points, average value of pi = 3.14149945
   After 120000000 points, average value of pi = 3.14155450
   After 160000000 points, average value of pi = 3.14148865
   After 200000000 points, average value of pi = 3.14158790
   After 240000000 points, average value of pi = 3.14156117
   After 280000000 points, average value of pi = 3.14155471
   After 320000000 points, average value of pi = 3.14153801
   After 360000000 points, average value of pi = 3.14157903
   After 400000000 points, average value of pi = 3.14154910
>> Parallel time = 71.058548 seconds

Real value of PI: 3.1415926535897
```

Speed-up = (Serial wall time / Parallel wall time)
Speed-up = 0.1906
**No speed-up

(Copy the command) **[Execute with 4 threads]**

```
!./omp_pi 4
```

Execution output:



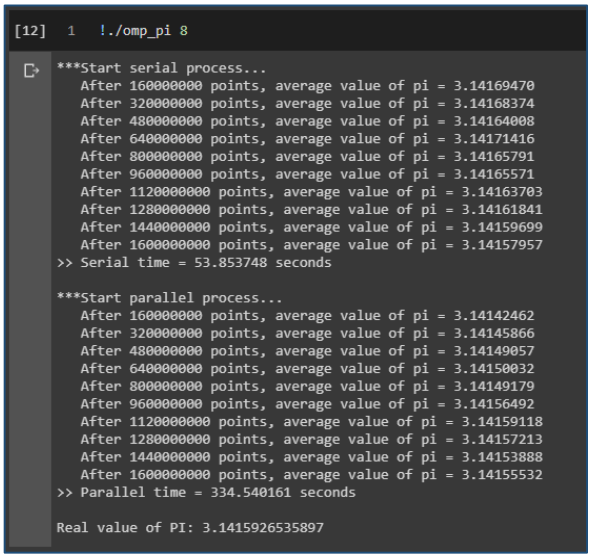
```
[11]  1  !./omp_pi 4

***Start serial process...
   After  80000000 points, average value of pi = 3.14149270
   After 160000000 points, average value of pi = 3.14172290
   After 240000000 points, average value of pi = 3.14163733
   After 320000000 points, average value of pi = 3.14165369
   After 400000000 points, average value of pi = 3.14165846
   After 480000000 points, average value of pi = 3.14164428
   After 560000000 points, average value of pi = 3.14163173
   After 640000000 points, average value of pi = 3.14159175
   After 720000000 points, average value of pi = 3.14162135
   After 800000000 points, average value of pi = 3.14158345
>> Serial time = 26.935768 seconds

***Start parallel process...
   After  80000000 points, average value of pi = 3.14118425
   After 160000000 points, average value of pi = 3.14134068
   After 240000000 points, average value of pi = 3.14143477
   After 320000000 points, average value of pi = 3.14150843
   After 400000000 points, average value of pi = 3.14150096
   After 480000000 points, average value of pi = 3.14151488
   After 560000000 points, average value of pi = 3.14153658
   After 640000000 points, average value of pi = 3.14149838
   After 720000000 points, average value of pi = 3.14148437
   After 800000000 points, average value of pi = 3.14148061
>> Parallel time = 117.803078 seconds

Real value of PI: 3.1415926535897
```

Speed-up = (Serial wall time / Parallel wall time)
Speed-up = 0.2287
**No speed-up

(Copy the command) **[Execute with 8 threads]**

```
!./omp_pi 8
```

Execution output:



```
[12]  1  !./omp_pi 8

***Start serial process...
   After 160000000 points, average value of pi = 3.14169470
   After 320000000 points, average value of pi = 3.14168374
   After 480000000 points, average value of pi = 3.14164008
   After 640000000 points, average value of pi = 3.14171416
   After 800000000 points, average value of pi = 3.14165791
   After 960000000 points, average value of pi = 3.14165571
   After 1120000000 points, average value of pi = 3.14163703
   After 1280000000 points, average value of pi = 3.14161841
   After 1440000000 points, average value of pi = 3.14159699
   After 1600000000 points, average value of pi = 3.14157957
>> Serial time = 53.853748 seconds

***Start parallel process...
   After 160000000 points, average value of pi = 3.14142462
   After 320000000 points, average value of pi = 3.14145866
   After 480000000 points, average value of pi = 3.14149057
   After 640000000 points, average value of pi = 3.14150032
   After 800000000 points, average value of pi = 3.14149179
   After 960000000 points, average value of pi = 3.14156492
   After 1120000000 points, average value of pi = 3.14159118
   After 1280000000 points, average value of pi = 3.14157213
   After 1440000000 points, average value of pi = 3.14153888
   After 1600000000 points, average value of pi = 3.14155532
>> Parallel time = 334.540161 seconds

Real value of PI: 3.1415926535897
```

Speed-up = (Serial wall time / Parallel wall time)
Speed-up = 0.1610
**No speed-up

3. DISCUSSION – In this OpenMP implementation, we observed that there is no speed-up for 2, 4, and 8 threads. It seems a bit odd at first look because there seem no data dependency for each thread. Or so we thought, let us not forget about the random number generator. Since the parameters of the random number generator is common among threads. Each thread will take turns in accessing it per sampling and it will take will a lot of time since we have a large sample size. This demonstrates that this problem is more suited to distributed-memory parallel computing (handling internal parameters of RNG might also work).

HOMEWORK: PARALLEL PROGRAMMING MULTIPLY 2 ARRAYS AND ADD ALL ELEMENTS

Implement the following operation in MPI and OpenMP:

Result = Add_All_Elements(Array_1[100000000] * Array_2[100000000])

****USE ELEMENT-WISE MULTIPLICATION**

****Fill Array_1 and Array_2 with random number using the random number generator:**

Pseudo code:

For each i elements in Array:

Array[i] <= (double)random() / (2^15 – 1) // 65535 will be the max value for each element

Then compare the time execution of each parallel implementation to the SERIAL implementation and compute the speed-up for 1, 2, 4, and 8 processes or threads.

****HINT: Please refer to or modify mpi_add.c and omp_add.c**

Present the following:

1. Full code of MPI and OpenMP parallel implementation with SERIAL implementation.

2. Execution output and speed-up for each number (1, 2, 4, and 8) of processes and threads.

3. Discussions regarding your implementation and the execution results.