Concurrent Systems

Laboratory Assignment 1

Aaron Musgrave (17762478)

# Estimating the value of π using a Monte Carlo approximation

## Describe the purpose of the following lines.

31 MPI\_Init(NULL, NULL);

32 comm = MPI\_COMM\_WORLD;

33 MPI\_Comm\_size(comm, &comm\_sz);

34 MPI\_Comm\_rank(comm, &my\_rank);

The above lines are responsible setting up the message passing interface (MPI) environment. The first line, MPI\_Init(NULL, NULL), instructs the MPI package to initialise all of the required functions. The third and fourth lines set the values of comm\_sz and my\_rank to the number of parallel processes and the rank of the current process respectively.

52 MPI\_Finalize();

The above line ends all the child processes removes the MPI environment from the current runtime.

## Broadcast number\_of\_tosses to all processes.

68 MPI\_Bcast(number\_of\_tosses, 1, MPI\_LONG\_LONG\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Bcast broadcasts the number\_of\_tosses variable from process 0 to all processes.

## Calculate a random dart location.

81 x = (random()\*2.0 / RAND\_MAX) - 1.0;    /\* random double between -1 and 1 \*/

82 y = (random()\*2.0 / RAND\_MAX) - 1.0;    /\* random double between -1 and 1 \*/

84 distance\_squared = (x\*x) + (y\*y);       /\* square of distance from origin \*/

/\* if dart falls in unit circle, increment the count number\_in\_circle \*/

if (distance\_squared < 1) {

88 number\_in\_circle++;

}

random() generates a random number between 0 and RAND\_MAX. This number is then scaled and shifted so that x and y are set as random real numbers between negative one and positive 1. The square of the distance to the origin (distance\_squared) is calculated using Pythagoras’ theorem which states that in a right-angled triangle where *c* is the length of the hypotenuse and *a* and *b* are the length of the two other sides. If the distance to the origin is less than one, then the current dart is inside the circle and number\_in\_circle is incremented.

## Compute global sum of dart tosses.

45 MPI\_Reduce(&local\_number\_in\_circle, &number\_in\_circle, 1,

MPI\_LONG\_LONG\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

The MPI\_Reduce() functions collects the local\_number\_in\_circle variable from each process, sums them and returns the result to process 0.

## Compile and run your code.

The code was executed in an Ubuntu instance using Windows Subsystem for Linux on a Microsoft Surface Book with an Intel Core i7-6600U CPU. This CPU has two cores, four threads and a maximum frequency of 3.40GHz.

A screenshot of a computer screen

Description generated with very high confidence

The above bash command console shows the compilation and execution of the program, firstly with four threads and then with one thread. The program returned estimated the value of pi to be 3.141605 and 3.141576 respectively. Both of these values are correct to at least three decimal places.

A screenshot of a cell phone

Description generated with very high confidence

Shown above is a graph of CPU usage while the program was running with four threads. It can be seen that all four logical processors are working at 100% capacity for approximately 15 seconds.

A screenshot of text

Description generated with very high confidence

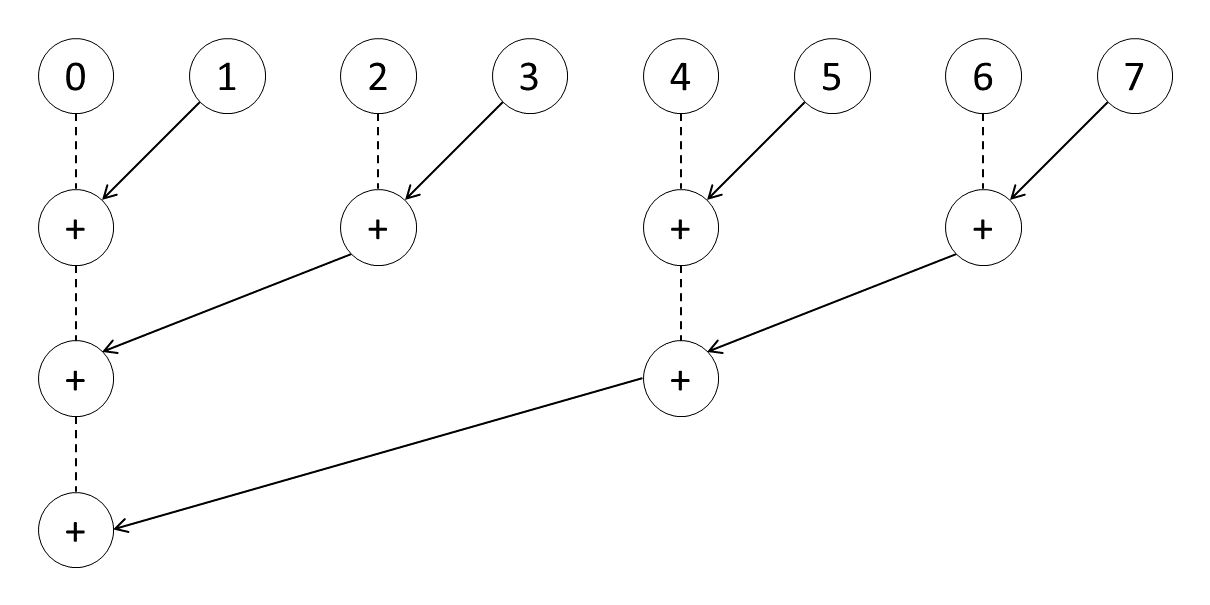
Shown above is a graph of CPU usage while the program was running with just one thread. It is interesting to note that all four logical processors appear to be working at approximately 25% of capacity, rather than one processor working at 100% of capacity and the other three idling. This demonstrates that the MPI library’s usage of hardware is not directly controlled by the number of threads running. The calculation lasted approximately 30 seconds.

The calculation was approximately twice as fast when using four threads instead of just one because it was able to fully utilise all four logical processors available on the machine rather than distributing one thread across four processors. The program would consistently estimate pi correct to four decimal places with a minimum of 2 billion (2×109) tosses.

# Tree Structured Global Sum

## Explain the function Global\_sum().

The purpose of the Global\_sum() function is to calculate the sum of a variable across all threads and store the result in process 0’s memory using the tree reduction pattern shown below.



Using this method of reduction, each process calculates its partner process rank and compares it to its own rank. If its rank is the lower value of the pair it will be the receiver and if its rank is the higher value of the pair it will be the sender. This is repeated until process zero has collected the values from all other processes. The following table states the functions of the partner, my\_sum, and bitmask variables.

|  |  |
| --- | --- |
| **Variable** | **Function** |
| partner | Used to store the rank of the current process’s partner.  Calculated as my\_rank XOR bitmask |
| my\_sum | The total of received value for the current process.  Initialised as my\_int and updated each time a new value is received. |
| bitmask | Used to calculate the current process’s partner. Initialised as an unsigned binary value 1 and bit shifted left by each receiving process. |

## Receive and send values between partner processes.

76 MPI\_Recv(&recvtemp, 1, MPI\_INT, partner, 0, comm, MPI\_STATUS\_IGNORE);

MPI\_Recv() saves the received value from the partner process into recvtemp.

81 MPI\_Send(&my\_sum, 1, MPI\_INT, partner, 0, comm);

MPI\_Send() sends the value of my\_sum to the partner process.

## Describe the purpose and execution of the following if statement.

38 if (my\_rank == 0) {

39 all\_ints = malloc(comm\_sz\*sizeof(int));

40 MPI\_Gather(&my\_int, 1, MPI\_INT, all\_ints, 1, MPI\_INT, 0, comm);

41 printf("Ints being summed:\n ");

42 for (i = 0; i < comm\_sz; i++) {

43 printf("%d ", all\_ints[i]);

}

44 printf("\n");

45 printf("Sum = %d\n",sum);

46 free(all\_ints);

}

47 else {

48 MPI\_Gather(&my\_int, 1, MPI\_INT, all\_ints, 1, MPI\_INT, 0, comm);

49 }

The first line, if (my\_rank == 0), checks to see if the current process is the parent process. The parent process will perform the following steps:

* Manually allocate the appropriate amount of memory to the all\_ints array
* Call MPI\_Gather() to place its values of my\_int into all\_ints at the correct location
* Print the values of all\_ints and the calculated value of sum
* Release the memory allocated to all\_ints.

All other process will call MPI\_Gather() and place their respective values of my\_int into all\_ints at the correct location and send them to process 0.

To summarise, the above code will gather each process’s value of my\_int so that they can be printed to the console along with the calculated sum.

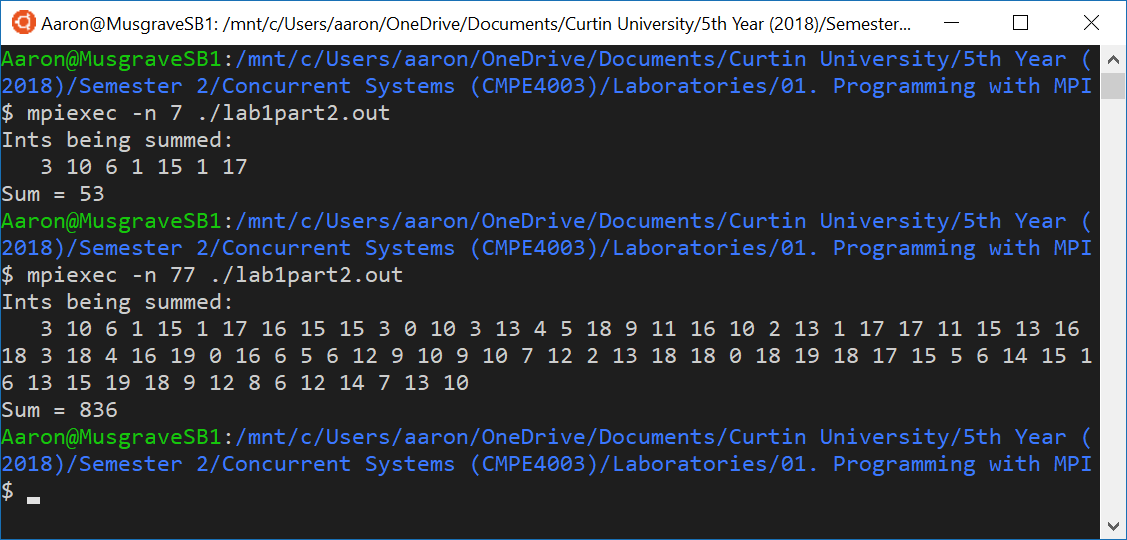
## Implement MPI functions so that data can be outputted.

40 MPI\_Gather(&my\_int, 1, MPI\_INT, all\_ints, 1, MPI\_INT, 0, comm);

48 MPI\_Gather(&my\_int, 1, MPI\_INT, all\_ints, 1, MPI\_INT, 0, comm);

Replacing the commented pseudo code at lines 40 and 48 with the above MPI\_Gather() call will collect the values of my\_int from each process into the all\_ints array at process 0.

## Compile and run your code



The above console window shows the program execution, first with 7 threads, then with 77. The results of the calculation can be confirmed as follows.

## Explain how a single MPI function call could be used to compute the global sum.

An MPI\_Reduce() function could be used to calculate the value for sum instead of using Global\_sum(). The appropriate code is shown below.

MPI\_Reduce(&my\_int, &sum, 1, MPI\_INT, MPI\_SUM, 0, comm);

# Butterfly Structured Global Sum

## Explain the input and output of the function Floor\_log().

The function Floor\_log() rounds the input parameter down to the closest power of two and returns the result. In this case, the input parameter is the number of threads, comm\_sz, so the function can be represented as the following equation.

This is useful in this application because a butterfly pattern requires the number of processes to be a power of two, so the result of this function can be used to account for scenarios where the number of processes is not a power of two. When this is the case, threads with ranks greater than or equal to floor\_log\_p will communicate with partner threads with ranks less than floor\_log\_p, shown below where process 8 to 11 are not involved in the butterfly sum.

A close up of a map

Description generated with very high confidence

## Implement MPI Send and Receive calls.

Implement send and receive calls for threads with ranks larger than floor\_log\_p to communicate with their partner threads before the butterfly add.

81 MPI\_Send(&my\_sum, 1, MPI\_INT, partner, 0, comm);

85 MPI\_Recv(&recvtemp, 1, MPI\_INT, partner, 0, comm, MPI\_STATUS\_IGNORE);

Implement a single function for two-way communication between partner threads during butterfly add.

94 MPI\_Sendrecv(&my\_sum, 1, MPI\_INT, partner, 0, &recvtemp, 1,

MPI\_INT, partner, 0, comm, MPI\_STATUS\_IGNORE);

Implement send and receive calls for threads with ranks larger than floor\_log\_p to communicate with their partner threads after the butterfly add.

102 MPI\_Recv(&my\_sum, 1, MPI\_INT, partner, 0, comm, MPI\_STATUS\_IGNORE);

106 MPI\_Send(&my\_sum, 1, MPI\_INT, partner, 0, comm);

## Describe the purpose and execution of the following if statement.

39 if (my\_rank == 0) {

40 all\_ints = malloc(comm\_sz\*sizeof(int));

sum\_proc = malloc(comm\_sz\*sizeof(int));

41 MPI\_Gather(&my\_int, 1, MPI\_INT, all\_ints, 1, MPI\_INT, 0, comm);

42 printf("Ints being summed:\n ");

43 for (i = 0; i < comm\_sz; i++) {

44 printf("%d ", all\_ints[i]);

}

45 printf("\n");

46 MPI\_Gather(&sum, 1, MPI\_INT, sum\_proc, 1, MPI\_INT, 0, comm);

46 printf("Sums on the processes:\n ");

48 for (i = 0; i < comm\_sz; i++) {

49 printf("%d ", sum\_proc[i]);

}

50 printf("\n");

51 free(all\_ints);

}

52 else {

53 MPI\_Gather(&my\_int, 1, MPI\_INT, all\_ints, 1, MPI\_INT, 0, comm);

54 MPI\_Gather(&sum, 1, MPI\_INT, sum\_proc, 1, MPI\_INT, 0, comm);

55 }

The condition of the if statement, my\_rank == 0, checks to see if the current process is the parent process. The parent process will then perform the following steps:

* Allocate memory for the all\_ints and sum\_proc arrays
* Gather the values of my\_int from all processes into all\_ints
* Print the values in all\_ints to the console
* Gather the values of sum from all process into sum\_proc
* Print the values in sum\_proc to the console
* Release the memory being held by all\_ints

All other process will perform these steps instead:

* Send their respective values of my\_int to the all\_ints array at process 0
* Send their respective values of sum to the sum\_proc array at process 0

To summarise, the above code block will gather the values of my\_int and sum from all processes before printing the collected values to the console.

## Implement MPI Gather calls.

Gather the values of my\_int from all processes, storing the result in all\_ints at process 0.

41 MPI\_Gather(&my\_int, 1, MPI\_INT, all\_ints, 1, MPI\_INT, 0, comm);

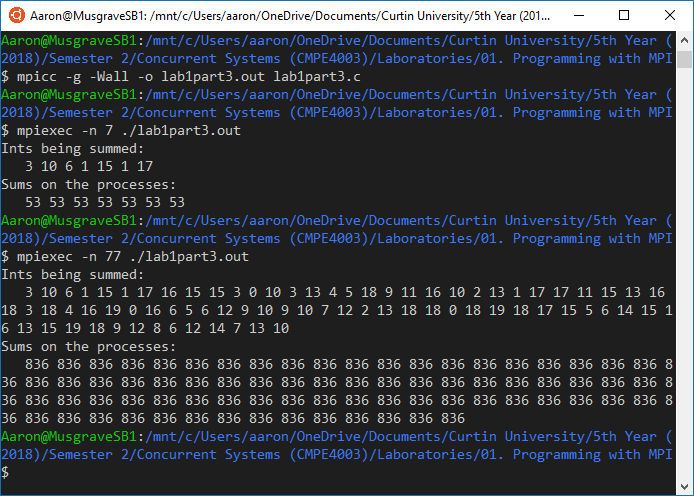
53 MPI\_Gather(&my\_int, 1, MPI\_INT, all\_ints, 1, MPI\_INT, 0, comm);

Gather the values of sum from all processes, storing the result in sum\_proc at process 0.

46 MPI\_Gather(&sum, 1, MPI\_INT, sum\_proc, 1, MPI\_INT, 0, comm);

54 MPI\_Gather(&sum, 1, MPI\_INT, sum\_proc, 1, MPI\_INT, 0, comm);

## Compile and run your code.



The above console window shows the program execution, first with 7 threads, then with 77. The results of the calculation can be confirmed as follows.

## Explain how a single MPI function call could be used to compute the global sum as a butterfly.

An MPI\_Allreduce() function could be used to calculate the value for sum as a butterfly instead of using Global\_sum(). The appropriate code is shown below.

MPI\_Allreduce(&my\_int, &sum, 1, MPI\_INT, MPI\_SUM, comm);

# Estimate the Time Cost of Sending Messages

## Explain the functionality of the C++ function clock() and macro CLOCKS\_PER\_SEC, and how these could be used to measure performance.

The function clock() returns the time used by the program so far in number of clock cycles executed. CLOCKS\_PER\_SEC gives the speed of the computer’s processor in clock cycles per second. Therefore, the result of clock() divided by CLOCKS\_PER\_SEC gives the time used by the program so far in seconds. By recording the value of clock() at the start and end of an algorithm the time taken for the algorithm to execute can be calculated by dividing the difference in the recorded values by CLOCKS\_PER\_SEC.

## Explain the functionality of the MPI call MPI\_Wtime(), and how it could be used to measure performance.

The function MPI\_Wtime() returns the real time as per a wall clock. To measure the time taken for execution of an algorithm, simply record the value of MPI\_Wtime() at the start and end of the algorithm in question and calculate the difference.

## Discuss the difference between the two methods for measuring performance.

TAKEN FROM LECTURE NOTES:

*CPU time is only time on CPU instruction, e.g. user code, library functions, OS calls. Does not include idle time (time spent waiting), counts the number of instruction cycles taken divided by the digital clock frequency. For a multicore processor, the time reported is the total CPU time as the sum of the CPU time take on all cores. Thus, for a single core processor it is always true that CPU time <= real time, which can be similar if there is very little idling. For a multi core process or it could be true that CPU time >= real time because the former involves a sum of times on each core.*

## Describe the purpose of the if statement in the ping-pong() function.

## State what message is being sent for the ping-pong and what range of message sizes are being timed.

## Calculate and return the value for the total elapsed time using clock().

120 start = clock();

129 return (clock() - start) / CLOCKS\_PER\_SEC;

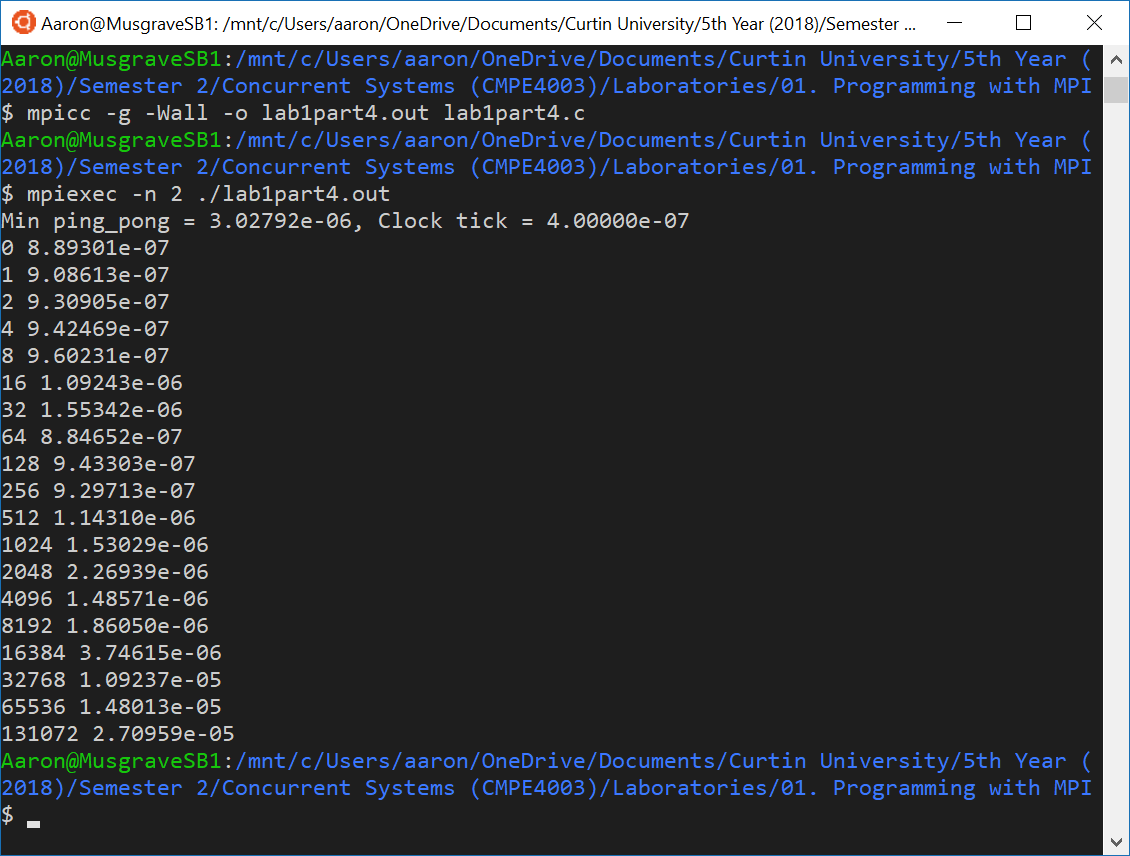
## Calculate and return the value for the total elapsed time using MPI\_Wtime().

118 start = MPI\_Wtime();

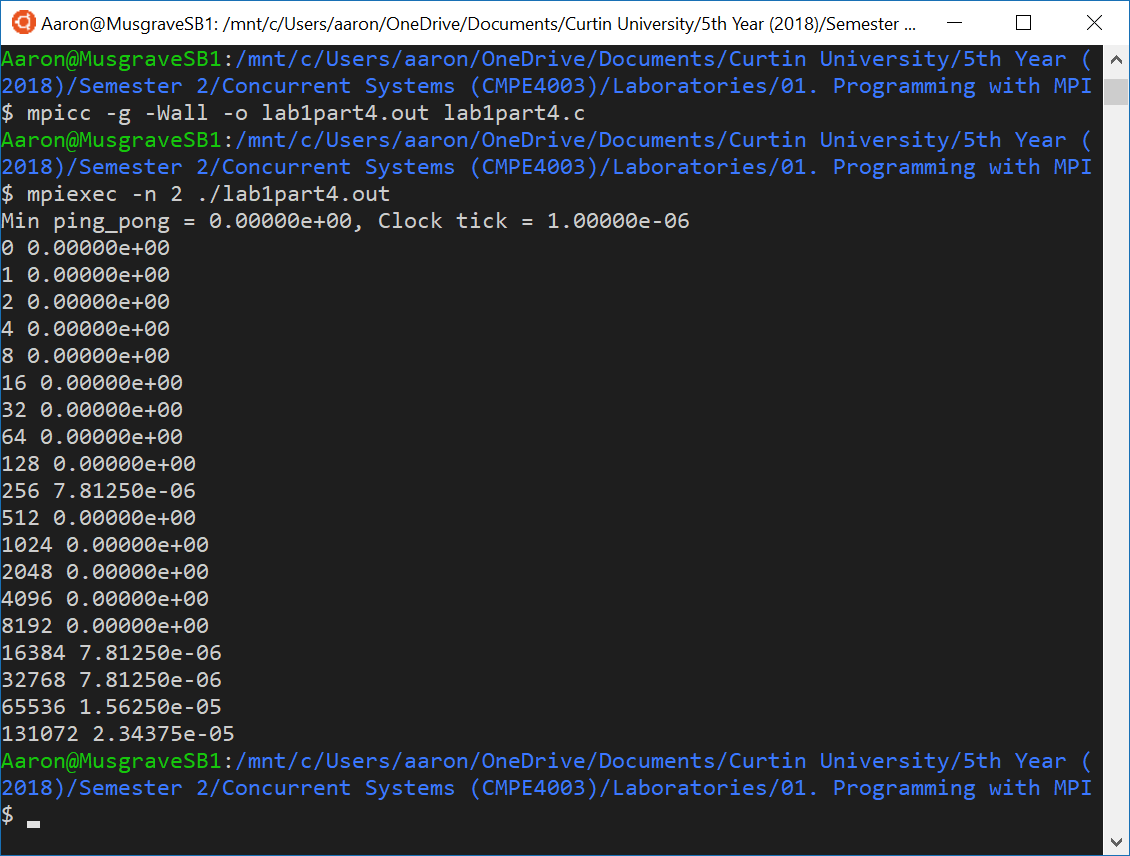
127 return (MPI\_Wtime() - start);

## Compile and run your code for both methods.

Without clock defined, the program will use the MPI\_Wtime() function to measure performance.



When clock is defined, the program uses the C++ clock() function to measure performance.



# Printing one message per process

This task required the output of the program to remain constant.

Original:

for (src = 1; src < comm\_sz; src++) {

MPI\_Recv(msg, MAX\_STRING, MPI\_CHAR, **MPI\_ANY\_SOURCE**, 0, comm, MPI\_STATUS\_IGNORE);

printf("%s", msg);

}

Fixed:

for (src = 1; src < comm\_sz; src++) {

MPI\_Recv(msg, MAX\_STRING, MPI\_CHAR, **src**, 0, comm, MPI\_STATUS\_IGNORE);

printf("%s", msg);

}