Parallel and distributed computing
Lab 1 Report
Yukti Abrol(ya246), Rohit Jain(rj288)

Part 3:

We opened the hello.c file to examine the code. The MPI_Init command initializes MPI. The MPI_Comm_size command returns the number of processes in the communication group. The MPI_Comm_rank command returns the rank of the current process in the communication group. MPI_COMM_WORLD is a default code for the communication group that contains all the processes. The program then prints out the rank of the current process. MPI_Finalize terminates the environment.

We compiled the hello.c file and used mpirun to run the file. Using the -np option, we are able to select the number of processes we want to run. This selects the number of copies of the program to execute. We previously listed the hosts in a file named hosts, so we use the -hostfile option to select this. The last argument was the name of the compiled c program (object file).

Upon running the mpirun command, we see an output such that our result is:

Hello, I am 0 of 4 processors!
Hello, I am 2 of 4 processors!
Hello, I am 3 of 4 processors!
Hello, I am 1 of 4 processors!

We tried running this program several times with the same number of processes. We see that the ranking argument changes values each time.

Hello, I am 3 of 4 processors!
Hello, I am 0 of 4 processors!
Hello, I am 2 of 4 processors!
Hello, I am 1 of 4 processors!

We also tried running this program a few times with a different number of processes. When the argument of the number of processes is greater than 0, we get an output that is the inputted arguments lines long. This also results in the maximum ranking argument changing. When the argument of number of processes is 0, we get the default number of processes, which is 8. When the argument of the number of processes is negative, the program crashes. The highest number we tested was 100, and the order of rankings still seemed to be random. Possible explanation for this is that these are the id's and are assigned randomly and execute in different order.

Hello, I am 1 of 6 processors!
Hello, I am 4 of 6 processors!
Hello, I am 3 of 6 processors!
Hello, I am 0 of 6 processors!
Hello, I am 5 of 6 processors!
Hello, I am 2 of 6 processors!

Parallel and distributed computing
Lab 1 Report
Yukti Abrol(ya246), Rohit Jain(rj288)

Part 4 (Profiling):

Message Latency:

The average latency for 1,000,000 tests came out to be between 360-380 ns. The number slightly changed every time we ran the code.

The N_TESTS value is used to declare the number of tests that the program will compute. We initialized MPI and got the number of processes in the communication group and the rank of the current process in the communication group.

For this program, we need exactly two processes, so we filter any calls to this program that do not use two processes. Then we declare and initialize some of the variables that we will be using to determine message latency. These include the status (for error handling), a counter, the id of the sender and receiver, the total time, the average time, and the time it took to send the message. The message that we want to send is a single character because it is one byte.

We use a counter to ensure that the message is sent from process 0 to process 1 N_TESTS times, such that we can calculate the average message latency. If the rank(id) is 0, we take the time before sending the message to process 1 using MPI_WTime. Once the message is sent, process 1 receives the character. Upon receipt, we take the time at which message was received using MPI_WTime.

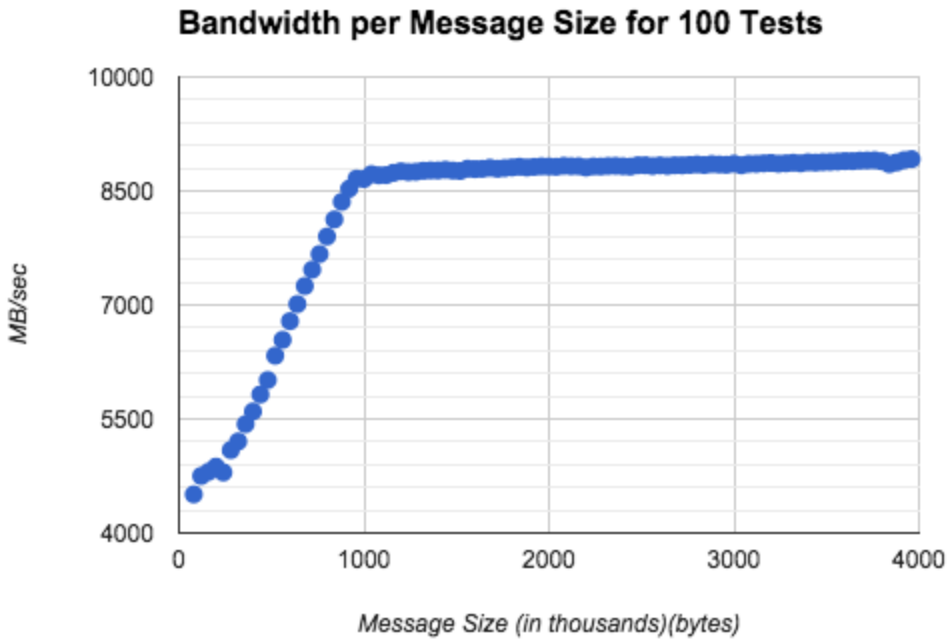
The difference between this time and the original time taken before sending the message from process 0 to process 1 is the message latency. To calculate this, process 1 sends the time upon receipt of the character message from process 0 to process 1. Process 0 then calculates the difference of the two times. This is added to a total_time variable to get the sum of the message latency time of all tests that the program computes. After the message latency is calculated N_TESTS times, then process 0 will calculate the average time by dividing the total time by N_TESTS, and prints this value. MPI_Finalize is finally called to terminate the environment.

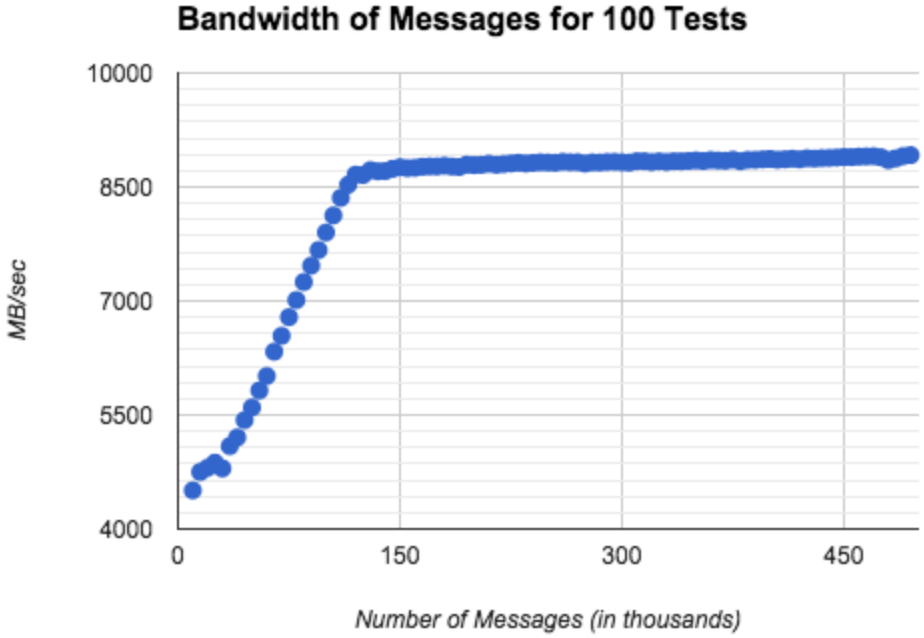
Message Bandwidth:

Observation: It increases linearly as we increase the message size that is being sent from one process to another. Initial the rate is very high but after the message size goes above million bytes the rate decreases and graph is almost flat. For the message size we multiply the number of messages by size of double.

Possible reason for this behaviour is that as the message size increases it affects the speed at which the network sends it. Maybe it needs to break it down and send it into multiple chunks as we hit the upper limit leading to a constant rate.

Parallel and distributed computing
Lab 1 Report
Yukti Abrol(ya246), Rohit Jain(rj288)



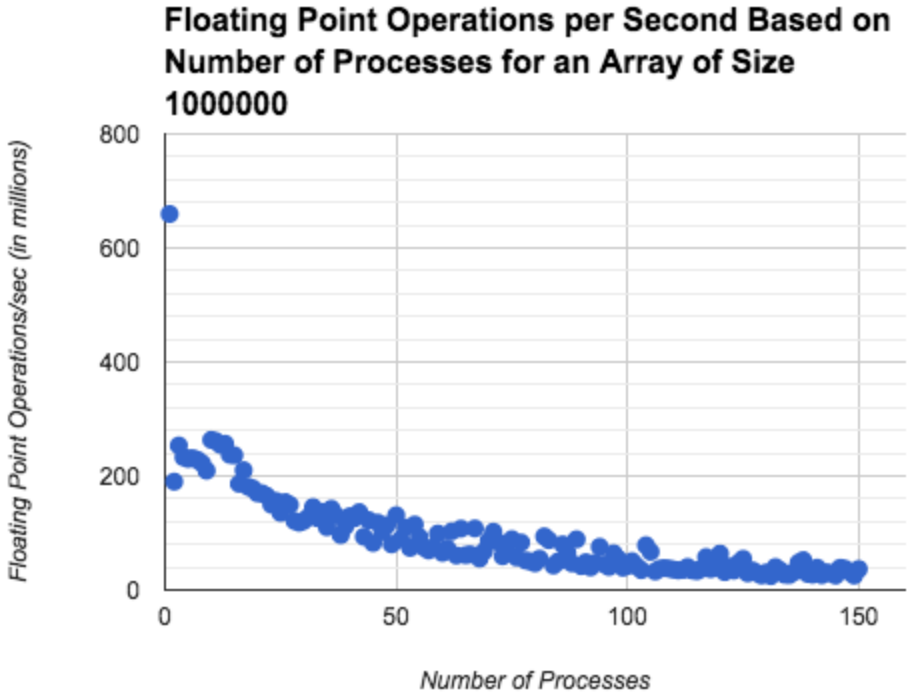


Method Explanation: We initialized MPI and got the number of processes in the communication group and the rank of the current process in the communication group. Then we declare and initialize some of the variables that we will be using to determine message bandwidth. These include the number of tests, the maximal amount of messages, the id of the sender and receiver, the number of messages to be sent, and the step size.

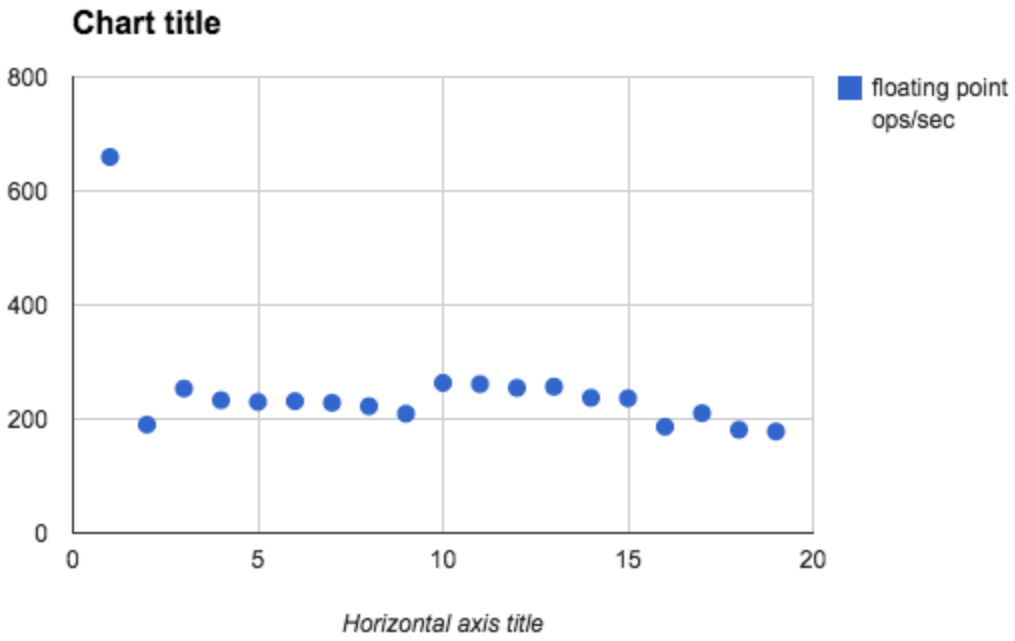
As in message latency, we keep a running count of the total bandwidth throughout the total number of tests that will be computed. In process 0, we create an array of size of the number of messages to be sent and initialize it with doubles. We then insert the current sending time using MPI_WTime before sending it to process 1. When process 1 receives the message, it gets the current time using MPI_WTime and then calculates the change in time from when the message was sent. The bandwidth is calculated as the size of the message divided by the message latency. This is then added to the total bandwidth. Process 1 then sends back an integer to process 0 to complete each test. Upon receipt of the integer, process 0 completes one test. After all the tests are completed, process 1 computes the average bandwidth for the number of messages and tests, and prints this. Then the number of messages is incremented by the step size and the bandwidth is computed the same way until the number of messages is the same as the maximal amount of messages.

Computation Time:

Observations: We observe that the floating point operations per second tend to increase or stay same for number of processes upto 10-15. After that they decrease continuously as the number of processes increase. Possible reason for that is we just have 8 slots in hosts. so as the number of processes increase there is a waiting time involved before each process can run. That will increase the computation time required to do calculations.



Parallel and distributed computing
Lab 1 Report
Yukti Abrol(ya246), Rohit Jain(rj288)



We created a helper method that will calculate the dot product. It takes the pointers of two arrays and the array length as its parameters. The dot product is calculated by multiplying the values of each array at a particular index and then taking the sum of all the products. The result is then returned.

We initialized MPI and got the number of processes in the communication group and the rank of the current process in the communication group. Then we declare and initialize some of the variables that we will be using to determine computation time. These include the id of the master and the running sum. We also declared three statuses and three tags for MPI_Send and MPI_Recv.

For this program, if there is only one process, it has to compute all the dot products by itself. For this case, two arrays with doubles are created. The time before the dotProduct method is called and the time after the dotProduct method is completed are obtained by using MPI_WTime and saved as start and end time. The floating point operations is calculated by assuming that the number of floating point operations is twice the array length since there is an addition and multiplication operation for every element of the array. To get the flops (floating point operations per second), we divide the number of floating point operations by the change in time. Then this is printed before freeing the arrays.

However, if there are multiple processes, we can take advantage of parallel computing. We decided to use a master-slave model, where the master will assign segments of the arrays to each slave. The slave will compute the partial dot product and return the result to master, who will take all the slaves' results and add them together to get the complete dot product. The master also is responsible for timing the operation from when it first sends commands to each slave to when it obtains the complete dot product.

The master creates two arrays with doubles, as in the the case when there is only one process. It then initializes the chunk sizes based on the total number of processes available minus itself. It starts the computation time from right before it breaks the arrays into chunks. The master determines how large each

chunk has to be and takes care of the instances where the array length cannot be equally divided into chunks. The master then sends the pointer to the first element of the array that the slave needs to process and the chunk size to each slave from process id 1 to the number of processes in the group.

The slave receives the array pointers and calculates the result using the dotProduct helper method. It sends the results back to the master and frees the memory allocated for the arrays. The master then receives the result from each of the slaves and adds the result to the running sum. Upon receiving all the results from all the slaves, it then gets the end of the computation time. The difference between the start and end of the computation time is calculated. The floating point operations is also assumed to be twice the array length since there is an addition and multiplication operation for every element of the array. To get the flops (floating point operations per second), we divide the number of floating point operations by the change in time. Then this is printed before freeing the arrays.