Damian Franco

101789677

dfranco24@unm.edu

CS-542

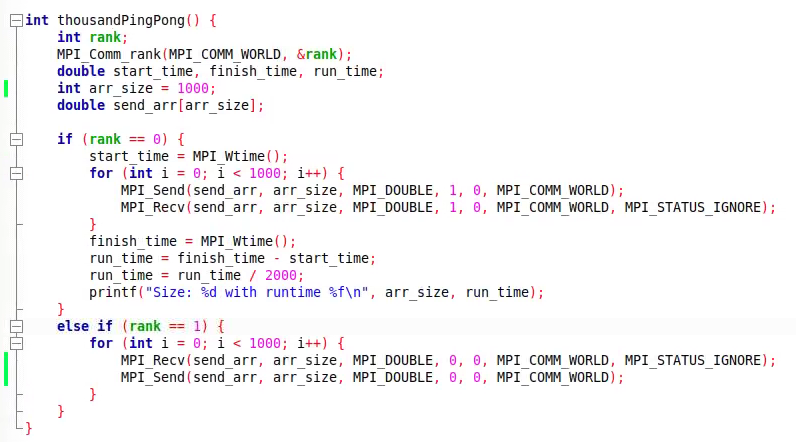
**Homework 3**

*Link to my github repo:*

[*https://github.com/ProfessorBienz/homework-3-code-dfranco24*](https://github.com/ProfessorBienz/homework-3-code-dfranco24)

1. The first problem asked us to create a program that utilizes MPI to test and time point to point communications. Utilizing the ping pong method, we can successfully time MPI point to point communications. Ping pong tests occur by having two processes send messages back and forth to one another. For example, process 0 will send a message to process 1 and will receive an equivalent message from process 1. So technically, after sending messages from both processes, the ping pong test will send a total of 2 messages. This will successfully measure the cost of a single message if we do some arithmetic to find the time taken.

The first unofficial part of this question asks us to measure the cost of a message by doing 1000 ping pong tests and dividing the measured time by the number of messages sent. There are a total of two messages sent every ping pong test so if we take that into account then we know that there will be a total of 2000 messages sent. I used this logic to find an approximate time for the cost of sending a single message. First I created a new method called *thousandPingPong()* which does exactly as it says. This first will make a new double array to send as a placeholder for data. This array did not have data assigned to it, rather just an empty buffer to send and receive to each ping pong process. Once that is made then there would be 1000 calls to send and receive from process 0 and the same for process 1. This would successfully ping pong around. After I then timed the algorithm. You can see the code in the image below or in my git repo under *point2point.c*.

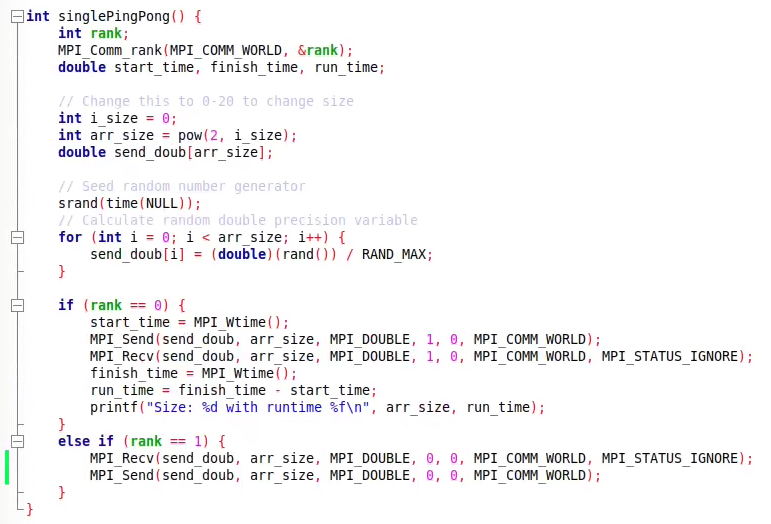
***Code for thousand ping pong tests:***

The test runtimes were anywhere between 0.00003 seconds to and 0.00006 seconds but never higher or lower. This would be the time that it would take to send a single message with an empty buffer using point to point communication within MPI. Below you can see an example of the runtime.

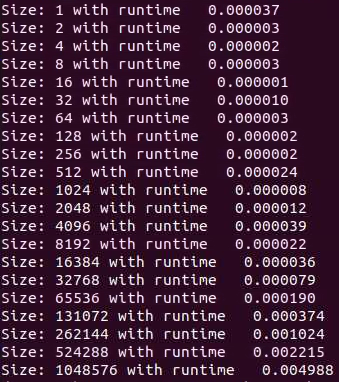
***Runtimes for single ping pong tests (CS Machines):***

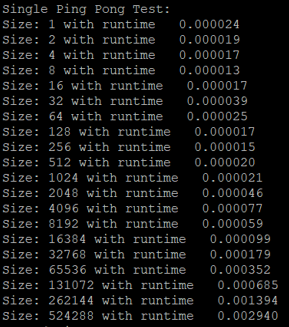
***Runtimes for single ping pong tests ( Wheeler):***

* 1. For the first part of the question, we were asking to measure the cost of sending a single message between two processes, process 0 and process 1. The size of the messages must be of size double and there must be 2^i where i is equal to 0 to 18. I tested this out with where i is equal to 0 to 20. To implement the code, I first took the thousand ping pong test code and instead of doing a thousand ping pong tests, I did a single ping pong test. The main difference between the thousand ping pong test and the single ping pong test is that in the single ping pong method, I now created a random array of doubles of the size 2^i. This array would be the sending and receiving message that each process will be using for this ping pong test. I then did a for loop where i is 0 to 20 and tested it out. Below you will see the code.

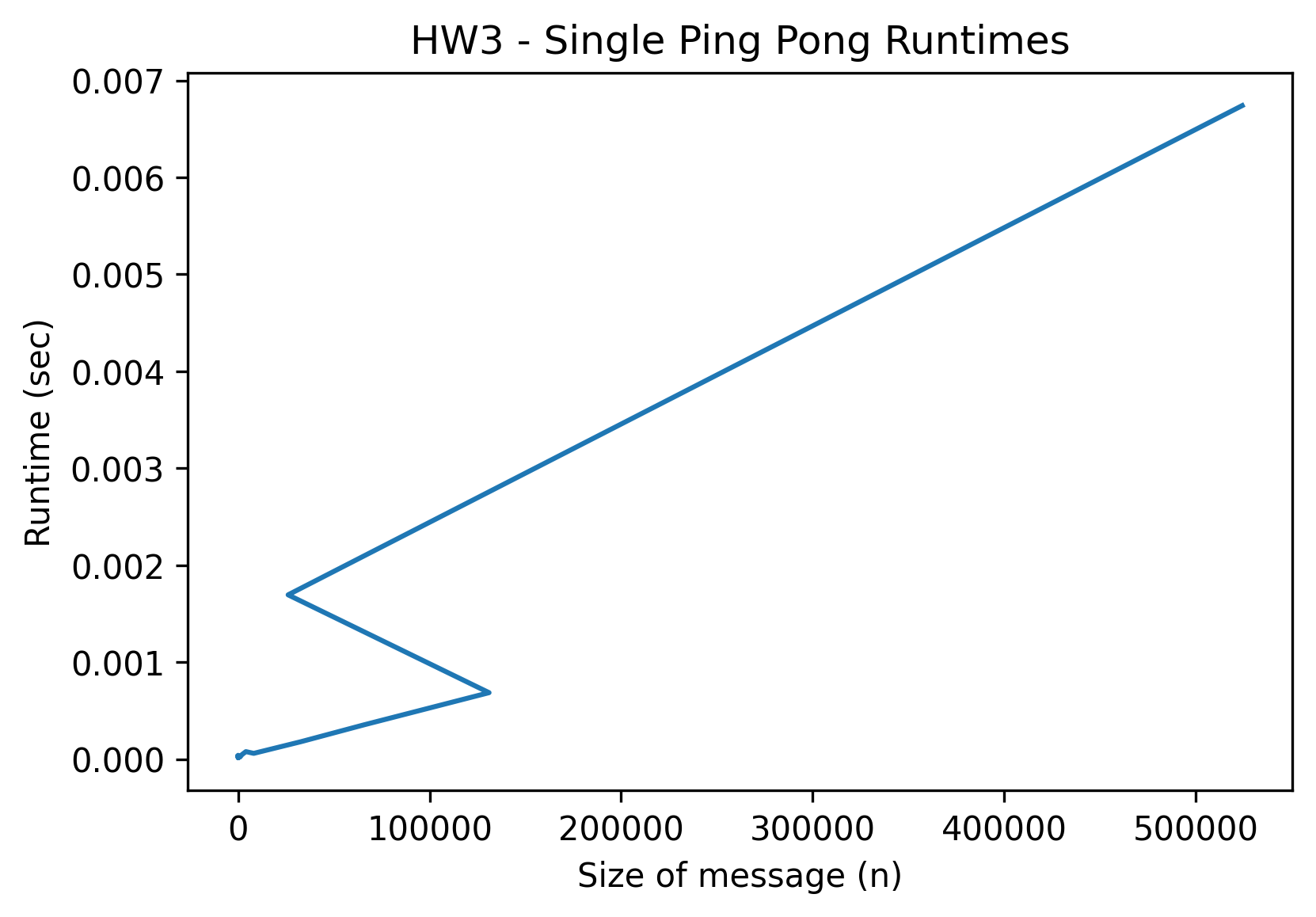
***Code for single ping pong test for different sizes:***

The test results were very unsurprising. As the sizes of the messages grew, the timing grew as well. This makes sense due to the fact that it will take much longer to send and receive a larger message then much smaller messages. The only anomaly that I noticed here was that sometimes there were points where a message of lower size will have a substantially larger runtime then some with a bigger message size. I chalked this up to some MPI bug within my code as the issue for this to be occurring. Regardless, the main idea of the runtime here stands true. Below you can see the test runs and a plot generated by Python of the runtimes.

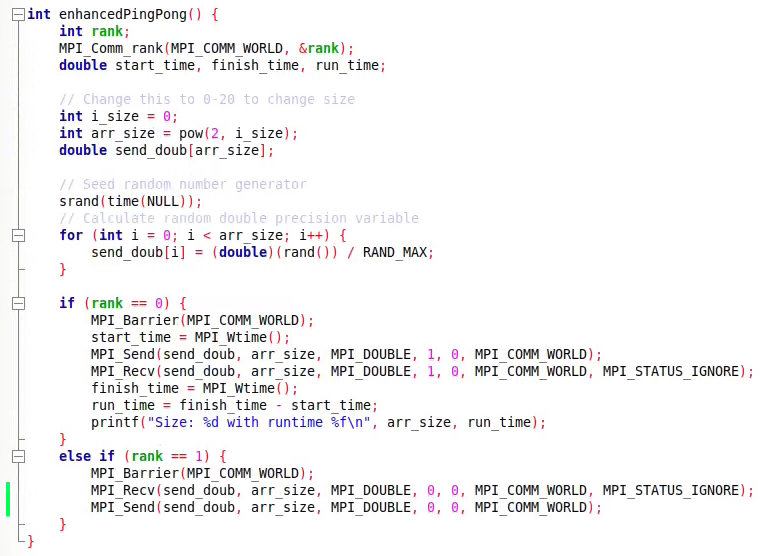
***Runtimes for single ping pong tests (CS Machines):***

***Runtimes for single ping pong tests (Wheeler):***

***Plot of single runtimes:***

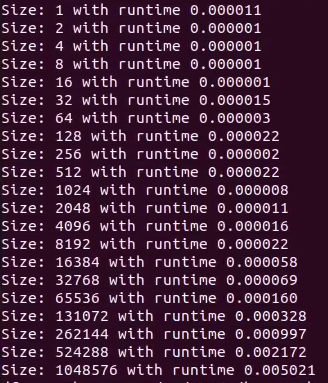


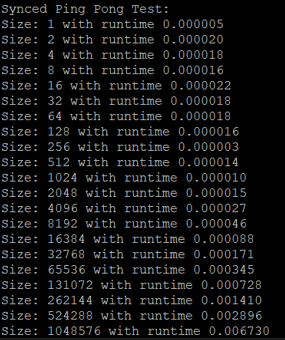
* 1. For the second part of the point to point testing, we were asked to improve the previous method we wrote by utilizing some MPI functions with timings. We were taught that there may exist some overhead when not using proper timing due to improper timing when sending and receiving processes. Process 0 may be sending a message while process 1 is still allocating the data that will be sent back. I first thought that this would all be fixed with some MPI collective operations, and looked online into various resources and found that some collective operations are great for send and receive optimizations. The one that I settled on was MPI\_Barrier, this would allow for fast synchronization with all processes so the overhead does not occur when improper timing occurs. So I implemented a couple of MPI\_Barrier calls to stop each process and allow for better synchronization within my ping pong tests. You can see that code below.

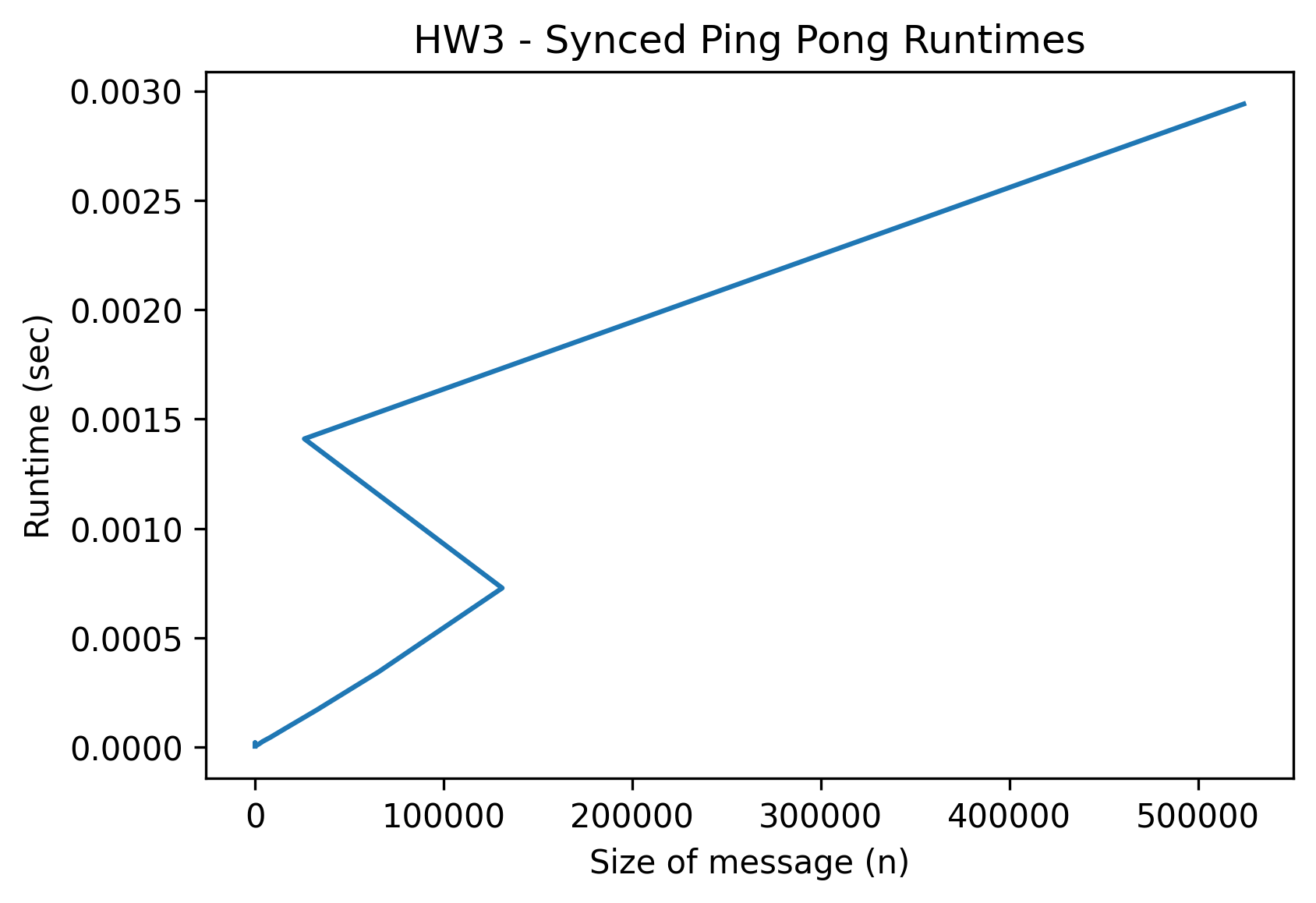
***Code for synced ping pong tests:***

This allowed for seeing results that were much faster then the processes that did not have synchronized processes. This test also made me realize that optimization techniques are very important to understand because although it is not a very wide difference, there is still a very good reason to optimize MPI programs to run more efficiently. You can view the runtimes and a plot generated in Python for these runtimes.

***Runtimes for synced ping pong tests (CS Machines):***

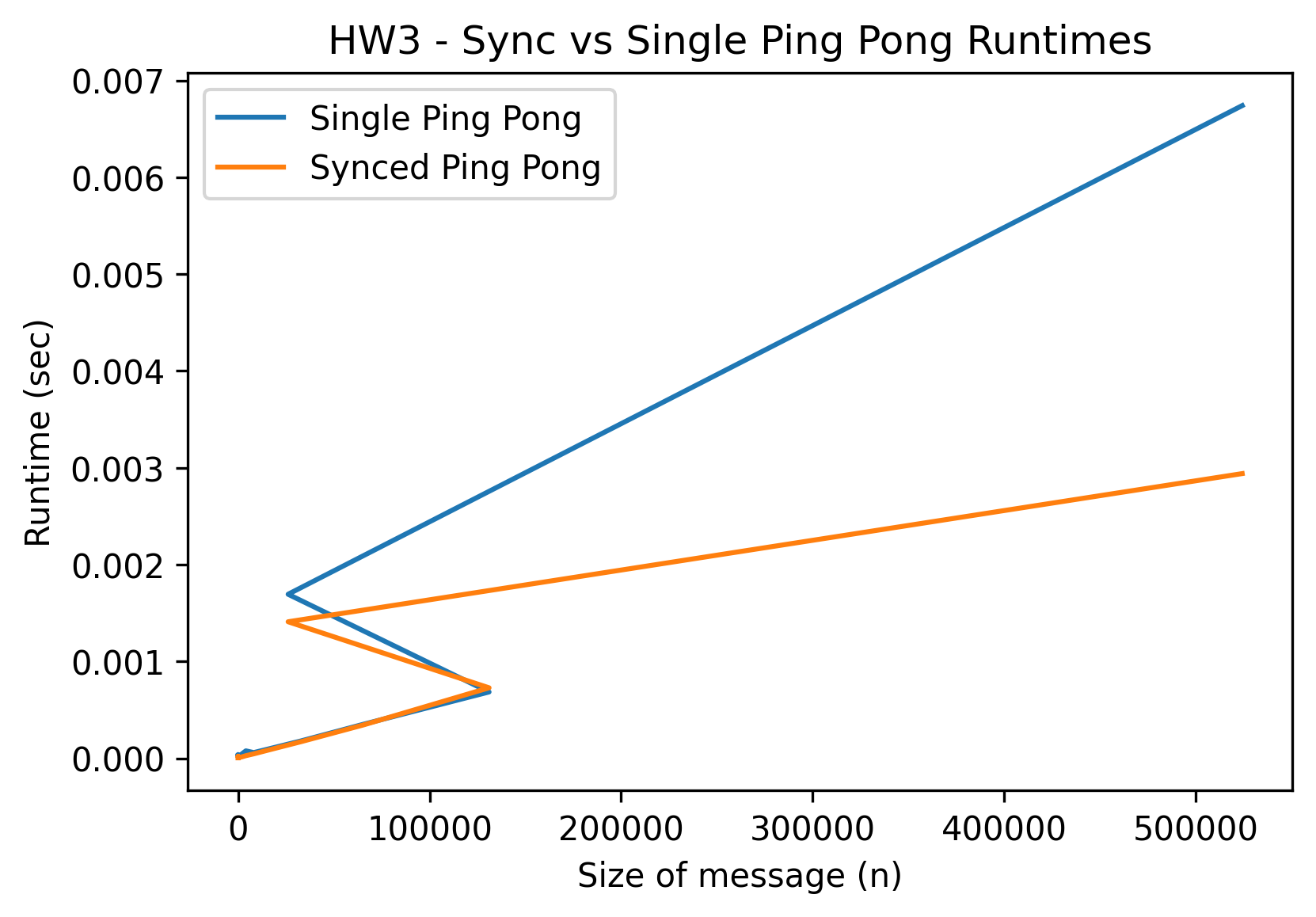


***Runtimes for synced gle ping pong tests (Wheeler):***

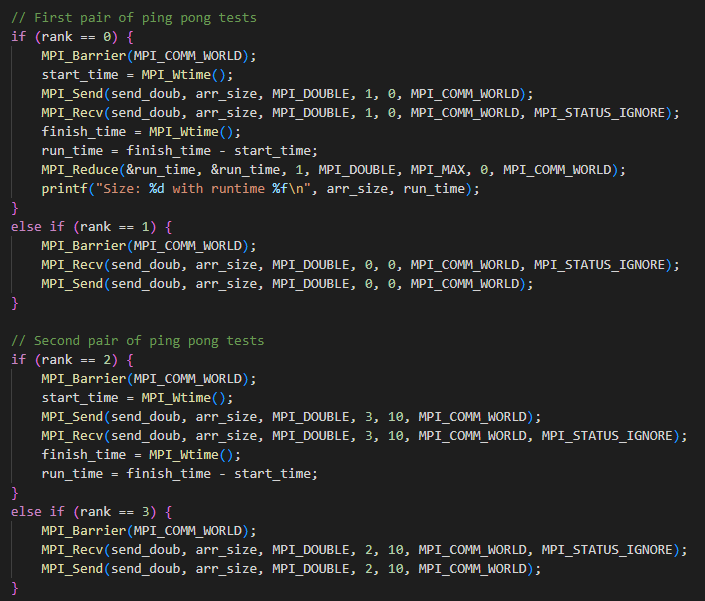
***Plot of runtimes:***

* 1. For this part we were asked to plot the costs which is what I already did, but for this I will simply provide a combined plot of the two plots above. Plotting both the non-synchronized and synchronized ping pong tests shows the wide difference between the two tests and exactly how important it is to optimize programs. Below you will see that the ping pong tests for the synchronized programs run much more efficiently then the non-synchronized programs. There seems to be an anomaly with my x-axis elements that I could not figure out how to fixe for each of these graphs, but it still shows the information I was expecting to see.

***Plot of runtimes:***

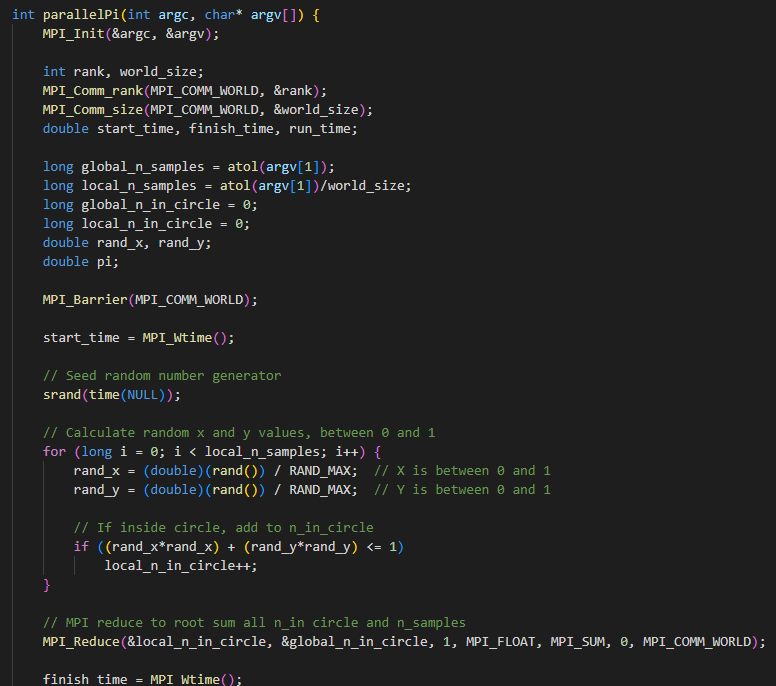
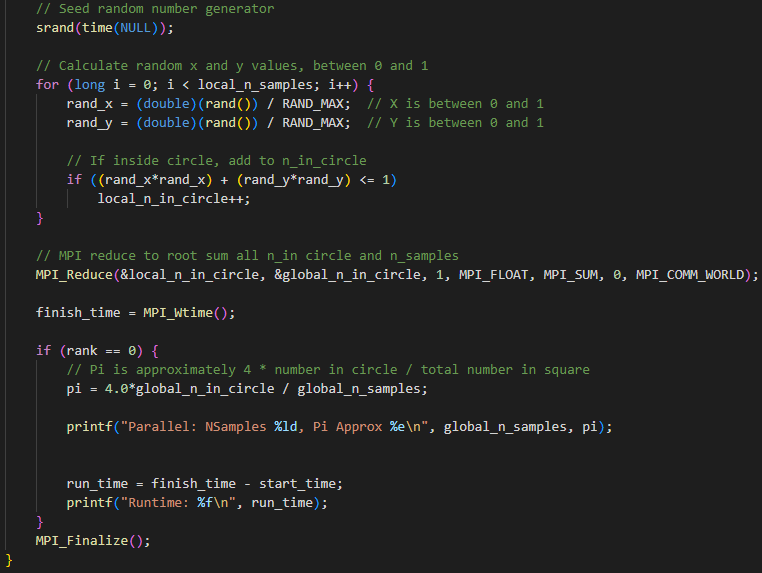


* 1. For the final part of the first question, we were asked to extend the ping pong tests into multiple pairs of processes sending messages to one another. This was a hypothetical question, but I decided to implement a simple version of this with processes 2 and 3 doing a ping pong test alongside processes 0 and 1 doing the same. The main question that was asked was how would we measure the total time to do a pair of ping pong tests knowing that we would have to measure timing for each process. This can be answered by doing a *MPI\_Reduce* call and using the MPI Operation *MPI\_MAX* to find the max time all the times it took for each process to finish. We would use the max as the timing because we know if other processes finished in less time then it would be improbable that it would take longer than the max time, which in turn means that the whole program finishes in max time. At first, I thought *MPI\_Sum* would be efficient to check the time for this program but that is not true since finding the max would be sufficient enough. Below you can see code and runtimes for my simple two pair test.

***Code for synced ping pong tests:***

1. For this problem, we are asked to create a program that utilizes MPI to calculate pi. This calculating pi method that we are using would approximate the value of pi by finding random numbers that exist inside of a unit circle. If the random numbers lie within the unit circle then an accumulator will be incremented within the code to represent how many fall within it. This number will then be placed within an equation that would successfully approximate pi. A serial version of the code was given to help us start and this helped me a great amount in understanding exactly what was needed to do to parallelize this code.

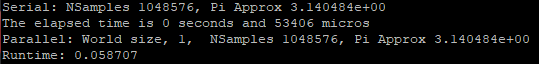
First let me explain how I did the parallelization with the code. After doing the initial MPI set up, then I first set up two long variables that would hold the number of *n* samples there are globally and local to every process. The *n* samples are input passed by the command line argument that would determine how many random variables to generate. The global *n* samples variable is used in the final equation (*4.0 \* global n in circle / global n samples*), and the local *n* samples determine the workload of each process based on the number of processes we have to expand our workload on. The next two important variables are the global *n* in circle and local *n* in circle which track how many random variables land on the unit circle and can be used in the final equation. Each of these are accumulators and the global and local keywords here are important in determining if the variable is local to its process or contains the global count of how many random numbers landed within the unit circle. Moving on, the program will now have each process generate their local *n* sample size and update their local *n* in circle accumulator when they land on the unit circle. Next, a *MPI\_Reduce* call is used to reduce all the local *n* in circle accumulators into the global variable through the MPI operation *MPI\_Sum.* This will successfully send the count of all the random numbers that land within the unit circle to process 0 or the root. Finally the pi equation will be run with the global variables and pi will be approximated. Below you can see the code.

***Code for parallel pi approximation:***

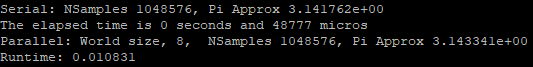
With the testing, I expected very efficient speedups as I increased the number of processes and nodes. This is due to the fact that there is evenly computational work that is sent through each process and each process does not have to wait for any other processes to conclude their computation of generating random doubles and checking if they are located within the unit circle. The speedup here is the very reason why parallel processing is very effective and why it can be used to highly optimize many programs and improve their efficiency. Below you can see the runtimes I calculated, as well as the plot of the timings.

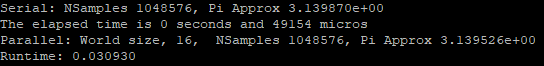
***Runtimes for pi approximation tests (Wheeler):***

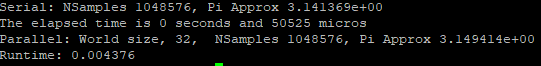
***Size - 1 process***

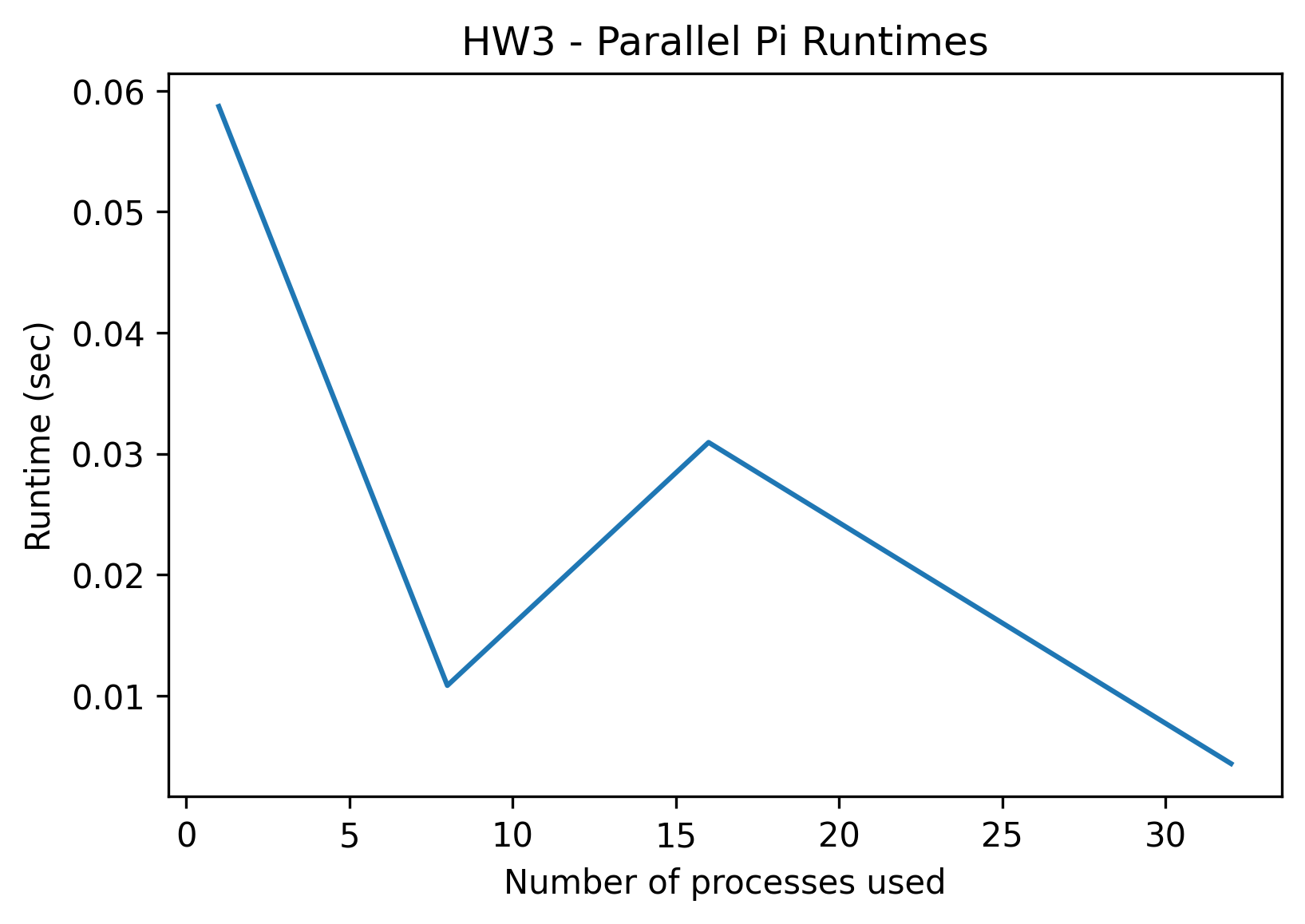
******

***Size - 8 processes***

******

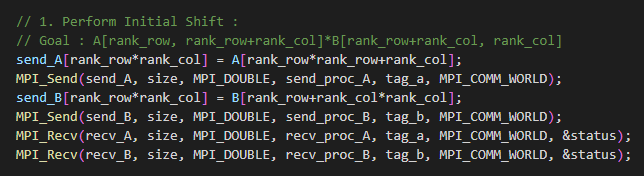
***Size - 16 processes, 2 nodes***

***Size - 32 processes, 4 nodes***

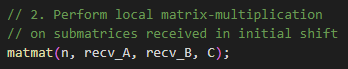
***Plot of runtimes:***

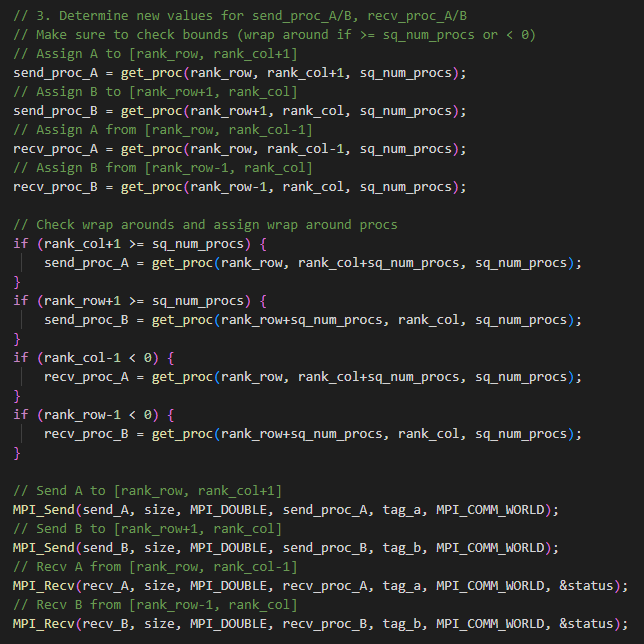
If I was to add more processes then I would assume that the speedup will only increase, which is exactly what occurs above. This is due to the fact that more processes can finish the program more efficiently in a parallel setting. The part of the program that we can expect to speed up and we increase the process count is the computation part, specifically the part where the program generates random numbers and checks if they are in the unit circle. With more processes, the less workload each process will have and the more efficient it will run when it comes to the computational side of things. I stated previously that it is important to note that distributing the workload evenly across multiple processes is the reason for speedups as we increase processes but I have not stated the cons. The part of the programs that can be negatively affected by the process count is the communications that take place when we use MPI functions. The more processes we have, the more time it will take for each process to quickly rely their information on other processes. This can cause some overhead in some instances and latency issues and occur but in my opinion the trade off is well worth it.

1. The last problem asks us to create a program that utilizes MPI to calculate the matrix by matrix multiplication. There are three different implementations that are being focused on in this question which are the simple matrix-matrix multiplication, Fox’s algorithm for matrix-matrix multiplication and Cannon’s algorithm for matrix-matrix multiplication. Each of these are different approaches in how to successfully implement matrix-matrix multiplication within a parallel setting. With the help of the professor's instructions, we were tasked to implement Cannon’s algorithm. We would then find out which approach from the three would be the best approach for this problem. The simple and Fox’s approach was done for us.

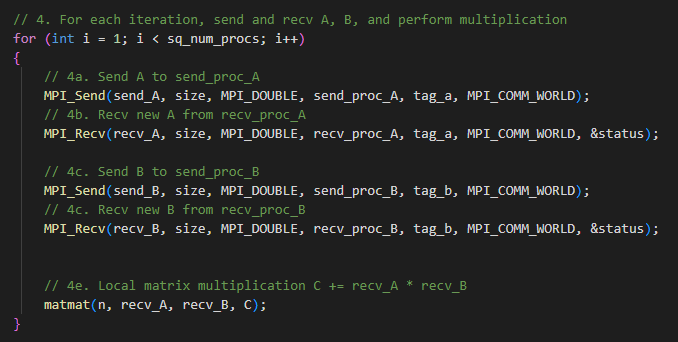
First, let me walk through the code for Cannon’s algorithm. The first goal of ours was to perform the initial shift in Cannon’s algorithm. The initial shift takes place at the beginning of the algorithm and performs a swap to rows and columns to line up the data in the appropriate processes. In the algorithm, data will be placed next to each other in the same process for fast and efficient multiplication to occur. The shift in this case will first send the data of matrix A from each row to the left by the number of the row. For example, if you are in row 3, then there will be a shift of data 3 places to your left. In the case of the code, that means it will be sent to the process that is exactly 3 places to its left. The same will be done with the matrix B, but instead of shifting left (columns), it will shift up (rows). This means that each column will shift up by the number of their columns. Shifting in the code will be handled by *MPI\_Send* function calls and the assigning of new send variables to the appropriate value. While shifting of both A and B, each process will receive from its neighbor processes until shifting is done. After shifting is completed, the initial shift is done. Below you can see my code for this initial shift.

Next, the algorithm needs to multiply the current local data in each process. This will be the first set of local matrix-matrix multiplication to occur within this problem. This was done by calling the *matmat* function that was given to us.



After performing the local matrix multiplication then the algorithm will perform more shifts to place data in the appropriate process. After multiplication then each process will send its local matrix A data to its neighbor process to its right. After that shift occurs, then the process will send its local matrix B data to the neighboring processes up. To simplify it for myself, I think of this step as the one-one step process where there is one step right and one step up in sending to neighboring processes. There is also a need to check for processes that are outside the bounds of the processes that we have available. If processes are trying to send data to other processes outside the bounds, then a wrap-around must be done by sending data to the opposite side. This is how I determined the new values for the sending and receiving processes in the code below.

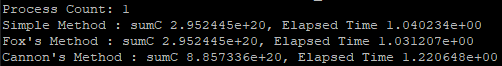
Lastly, the algorithm will continue to do the same shifting as described for each iteration until all of the processes have been multiplied together. To achieve this, every process will be visited and send its information to the receiving buffer and every local matrix that is received will perform matrix-matrix multiplication and store its values in the solution matrix C. That is what is shown in the code below.



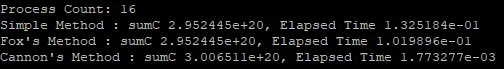
Overall, the runtime of this algorithm surprised me because it is not very efficient compared to the other two approaches. Cannon’s algorithm was much faster at any scale then the other two algorithms. Fox’s algorithm seems to be the best algorithm all around when it comes to comparing the two. In fact, I was almost 100% sure that the way I implemented Cannon’s would cause an issue for the much larger runtime because of the countless communication calls I had compared to how efficiently Fox’s algorithm and the simple approach was implemented, but was surprised to see that Cannon’s performed better than both of the approaches. Scalability within each algorithm shows the same outcomes at any scale that Fox’s seems to perform better when scaling larger due to the fact that Fox’s may be more efficient in memory management and usage than the other two algorithms. Below you can see a plot of my timing for all three methods.

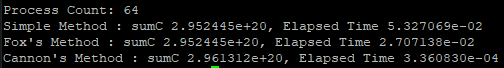
***Runtimes for all matrix-matrix methods (Wheeler):***

***Size - 1 process, 1 node***

******

***Size - 16 processes, 2 nodes***

******

***Size - 64 processes, 8 nodes***

***Plot of runtimes:***

