Damian Franco

101789677

dfranco24@unm.edu

CS-542

Homework 4

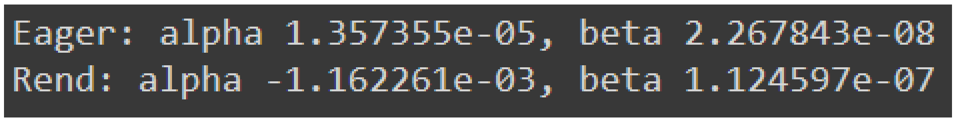
Link to my github repo:

https://github.com/ProfessorBienz/homework-4-code-repo-dfranco24

1) The first problem asked us to find the **α** and **β** parameters for the standard postal model for modeling inter-node communication on Wheeler. The **α** here will be the message latency. Beta **β** will represent the per-byte transport cost from node to node. The question asks us to refer back to our ping pong tests from the previous homework. I made some adjustments to correct the placement of my timer in that previous implementation of ping pong that you may view in my github repo. I found that I achieved a great amount of more accuracy when updating the placement of the timer. The output for my ping pong tests were then printed in a txt file which consisted of both the size tested and the amount of time it took to complete with that message size. When I say size here, I am meaning the number of bytes that are communicated across processes. Since I am using variables of the size *double*, I know each message sent with a single double would be multiplied by 8 due to the doubles being of byte size 8. Using the txt file, I then utilized the Python code provided by the professor to find the **α** and **β** parameters by solving a linear system of equations *Ax* = *b* where *A* would hold the size of messages I was sending, *b* would hold the time recorded for each size of message, and *x* would hold **α** and **β** which is what we would solve for. The professor provided a great Python script that uses the library *numpy* to solve the linear system for *x*. This would successfully solve the system, but there also is another layer of abstraction that I did not point out. Eager and rendezvous messages relate to the size of the messages. Eager messages are messages that do not fit in an envelope and are small, while rendezvous messages are the largest messages that we test. We were required to calculate both the eager and rendezvous **α** and **β** parameters which

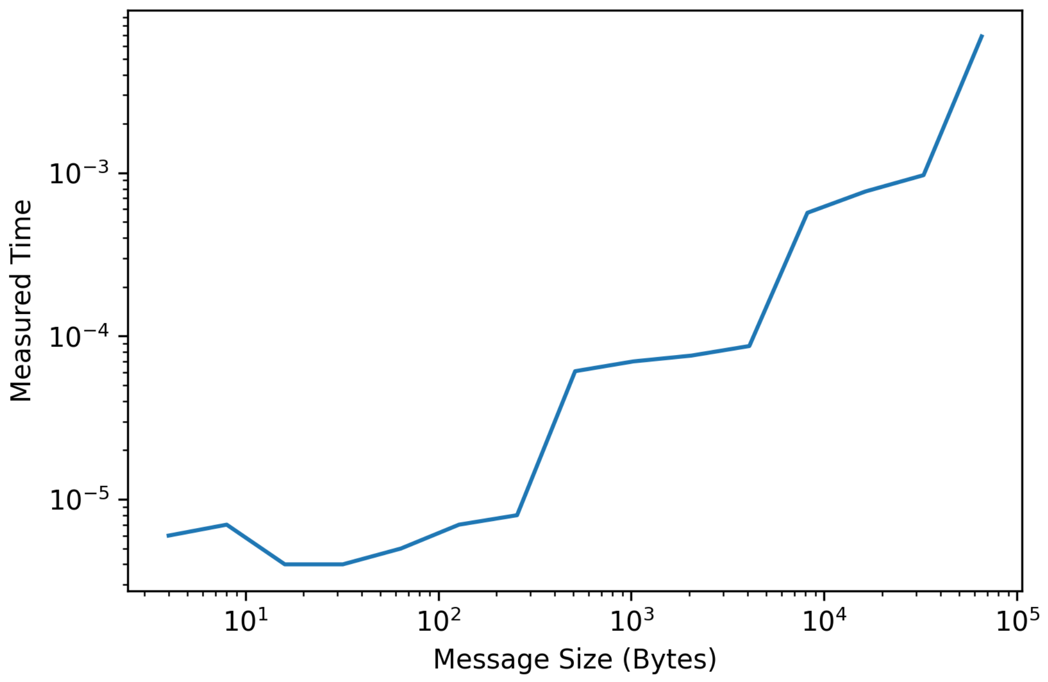
was done by solving two linear systems, one for the size of 8192 bytes and lower and the other for 8192 bytes or higher messages sizes. You can see that the **α** term for the eager messages are much larger than the **α** for the rendezvous messages. Meanwhile the **β** term here is a good amount smaller for the eager messages compared to the rendezvous messages. Another thing to note here is that both **α** values are much larger than **β**. These values were generated from a txt from a run of the ping pong test off of Wheeler.

***Eager and Rendezvous* α *and* β:**



```
Eager: alpha 1.357355e-05, beta 2.267843e-08
Rend: alpha -1.162261e-03, beta 1.124597e-07
```

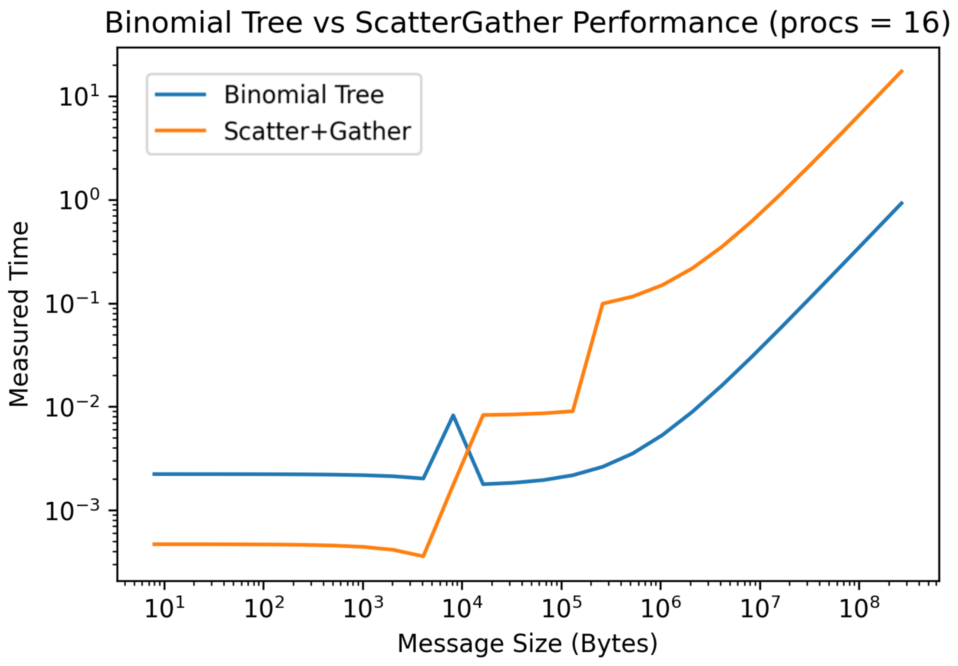
***Plot of message sizes sent and their runtimes (point2point.cpp)*:**

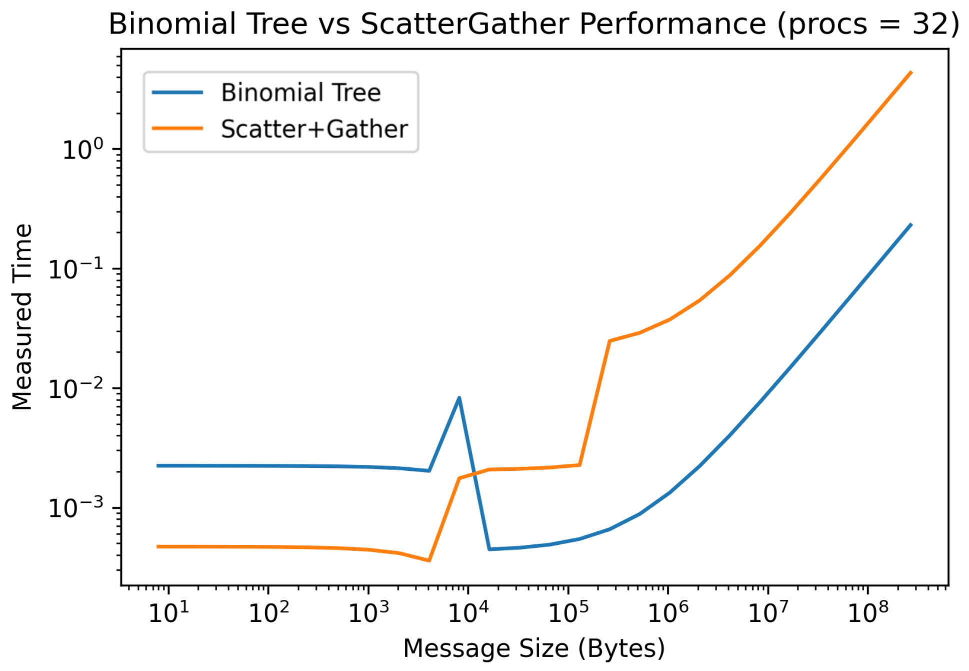


2) This question asks us to model the costs of two of the broadcast methods algorithms, the binomial tree algorithm and the scatter + allgather algorithm. Here, we are simply just implementing the performance model and solving for an estimated runtime for each message size we will be sending. To do so, I first had to understand why the performance models were designed in the way for each of the algorithms. The model for the binomial tree algorithm is used specifically to find high performance in sending smaller messages which can be seen by the simple performance model here. The *log_2*(*p)* represents the number of steps that the algorithm will take to complete. That is the main difference between this model and the postal model. The scatter + gather algorithm has much more depth than the binomial tree algorithm. The *log_2*(*p)* which represents the number of steps that the scatter command has to do while the *p - 1* represents how many steps the gather method has to do before it completes. The algorithm also sends a total of *n* values through the scatter method as well as utilizing the all-gather ring algorithm that sends *n/*p values at each step during the algorithm which represents the *2*(p-1/p*) value of the **β** side. Our task was then to implement this model and predict timings for 16, 32, and 64 processes. I did this by using a Jupyter notebook with some excellent numpy python methods for matrix-vector multiplication. This is very similar to the first problem, but instead of solving a linear system, here I will be simply doing a matrix-vector multiplication to solve for the *t* values or *b* vector. First, I implemented the vectors and matrices with the proper models for each of the algorithms by utilizing the techniques I learned throughout the performance model lectures, as well as following the example code the professor gave to us. This allowed me to simply perform the multiplication and find the estimated values. I noticed that these plots when plotted against one another show that the binomial tree algorithm is much more efficient at sending small messages and the scatter + gather algorithm is much more efficient at sending bigger messages. This was not very surprising due to the fact that this was said during the lecture as well as in the homework write-up. The thing I was surprised about is just how much more efficient it is, which is much, much more efficient than the binomial tree algorithm. Another note I'd like

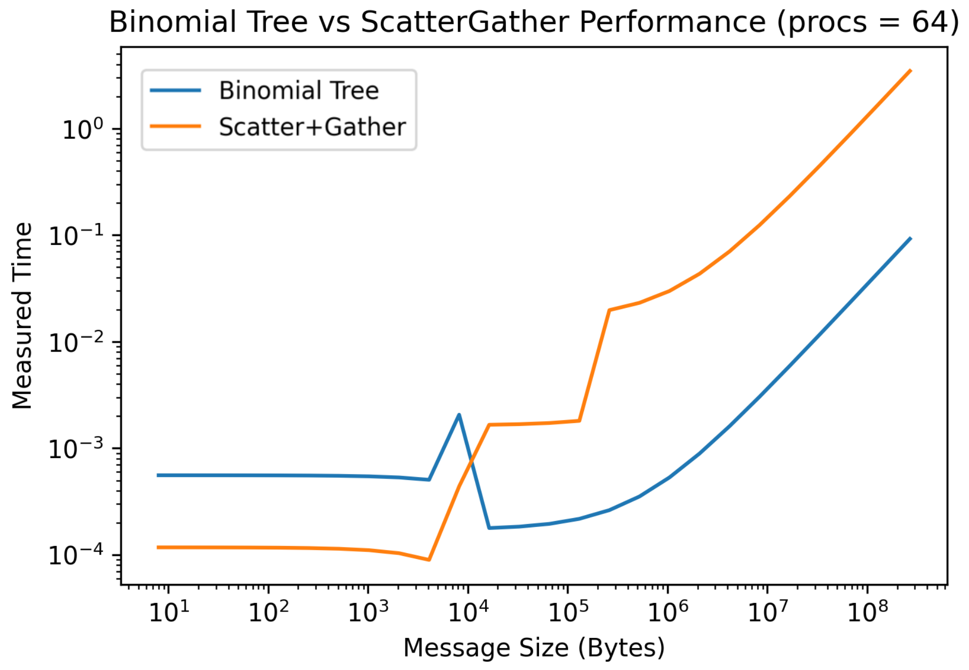
to make about the graphs is that both algorithms seem to only increase in performance when more processes are added which lines up properly with my understanding of parallel applications. I am very interested to see how my implementations of each algorithm relates to these plots and estimated timings in comparison. The interesting thing I picked up here was that the number of processes did not change efficiently in the measured runtime. This makes me believe that I must have a bug within my software that generated these estimates for these runtimes. You may view my Jupyter notebook code to see exactly how I was able to achieve this prediction of each algorithm's runtimes.

Plots of message sizes sent and their estimated runtimes:





Binomial Tree vs ScatterGather Performance (procs = 32)
Binomial Tree
Scatter+Gather
Measured Time
Message Size (Bytes)



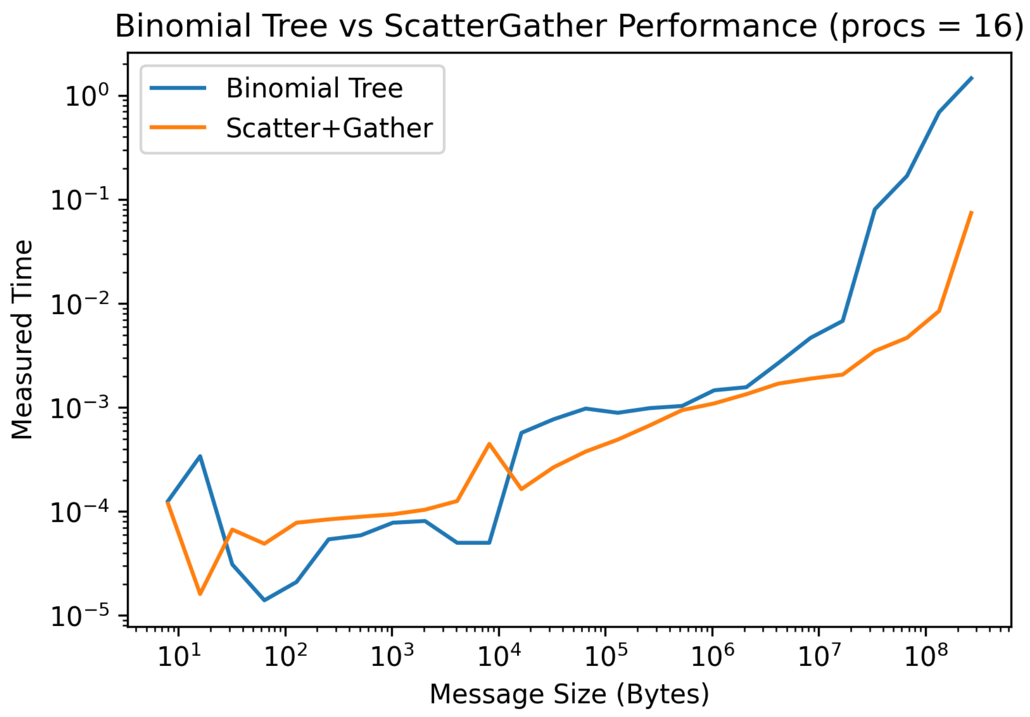
Binomial Tree vs ScatterGather Performance (procs = 64)
Binomial Tree
Scatter+Gather
Measured Time
Message Size (Bytes)

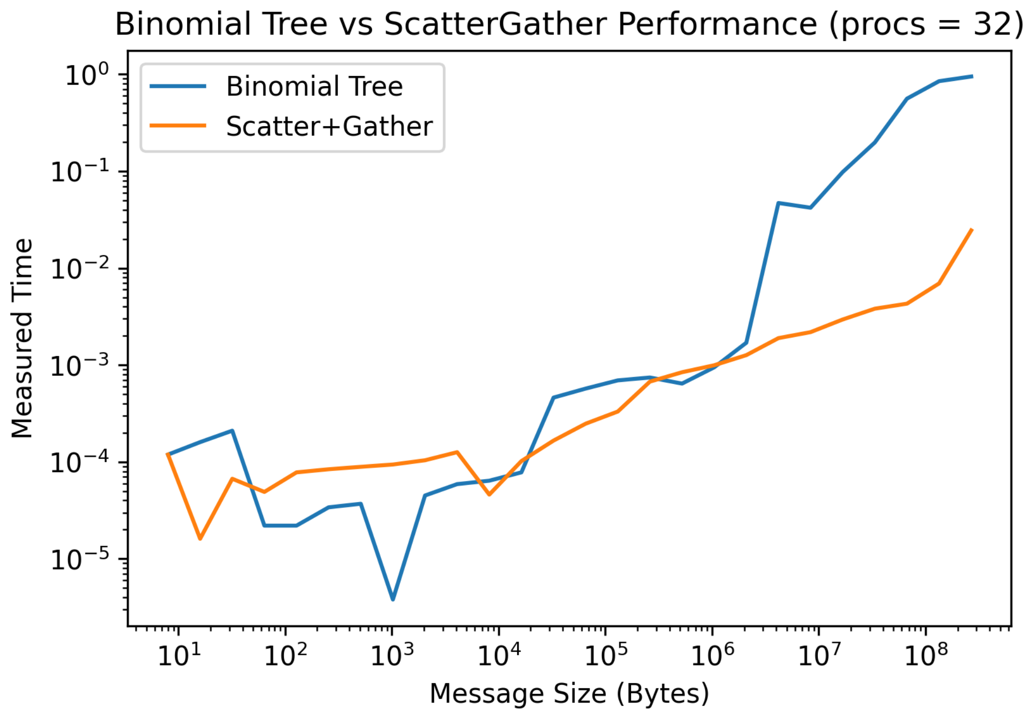
3) This question asks us to implement the two broadcast algorithms, binomial tree and scatter + gather. I first decided to implement the binomial tree algorithm which proved to be a bigger challenge to me than I expected. I first did the initial MPI and message memory allocation setup before starting to write the algorithm. I noticed that this algorithm will have *log2(p)* number of steps that needed to be taken in order for this algorithm to be successfully implemented. This was necessary to have to correct the amount of steps for the number of processes due to the fact that when I was implementing the algorithm, I would get a good amount of processes that would hang and wait for a send but never got one. This was due to the incorrect amount of times the for loop was iterated through. I then needed to fully understand which process and how the binomial algorithm works to make a successful MPI implementation. The first thing I noticed is after each step the ranks that are given the output *rank % numprocs/currentstep* were the nodes that would send data. This means that if we had 64 processes and were on step 2, then we would send data from ranks 0 and 32 because those are the process numbers that are equal to 0 when applying the mod function. This also indicated to me that the processes that we were sending to were also correlated with the step number we were on which I unfortunately did not find. I know that there is a solid correlation here, but I did not have enough brain power and time to do so which is why if you view in my code that everything is hard-coded. I can make a inference by viewing this hardcoded code is that we send from *currentstep*2* processes. I then made a useful for loop that involves a mathematical function to help where exactly to send to processes with the hard-coded values for the receive. Besides that, I added some barriers to make the code more sequential and a collective call to wait for all processes to finish before ending the program and recording the time. You can view this very long and jumbled code in the git repo.

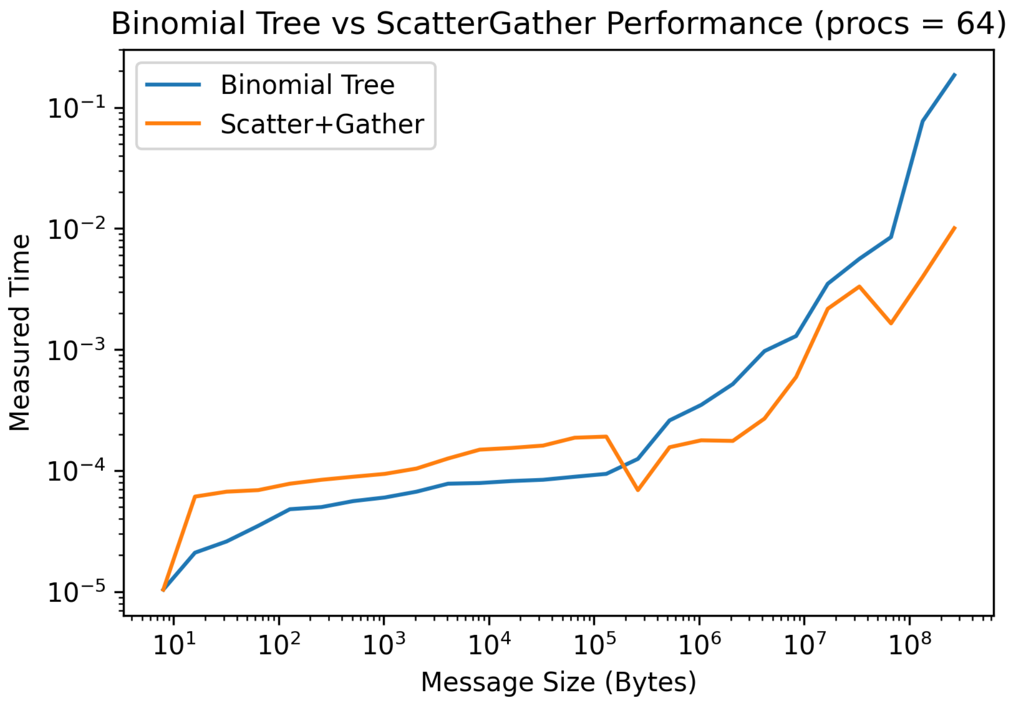
The scatter + gather method was much easier for me to implement. This code is using the MPI_Scatter method call as instructed in the write-up, but originally was trying to use a method like recursive doubling. I was running low on time so I decided to utilize some of the wonderful API that MPI has. First, I scatter the data to every process. Next, I implemented the ring all gather method which would send data from the current process to its neighbor and would ring around if it does not have a neighbor. When I say neighbor here, I mean the processes that have the *rank+1* from the current process. I noticed that all processes that are sending data will return a 0 when their rank is modded by 2 (even) and all other processes will receive (odd). Every even process will then send to its neighbor and all odd processes will receive from its neighbor. I then checked for boundaries by seeing if the current rank was the last rank in the total number of processes. If it was the last process, then it would need to "ring" around and send to the first process with rank 0. Rank 0 would also need to receive this from the last process which I also wrote as a boundary check in the code. Overall, this was a fairly straightforward and easy to implement algorithm, at least compared to the binomial algorithm for me.

When running both methods and plotting their runtime, I got the results I expected with a great amount of anomalies within the data. I know that the anomalies were most likely appearing due to the way I implemented both algorithms, but overall, the data looks very clear and seems to somewhat match the performance models predicted data within problem 2. There are some high differences when it comes to the size of the bigger data. I did have tests where the binomial algorithm would have runtimes much larger than the scatter + gather algorithm, even with smaller messages being sent. Once again, I am also 95% sure it is due to the way I implemented the algorithm. Other than that, the runs that used a larger number of processes would be much more efficient then the other runs which also surprises me because I was using a great amount of off-node communication because of running with 8 nodes and 8 processes per node. Below you can see my plots for the runtime of both algorithms.

***Plots of message sizes sent and their runtimes*:**






Binomial Tree vs ScatterGather Performance (procs = 64)
Binomial Tree
Scatter+Gather
Measured Time
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
10^{6}
10^{7}
10^{8}
Message Size (Bytes)