Kevin Cho
A20393339

MPI

Algorithm

The algorithm for the original serial gaussian elimination was approximately N^3 + N^2. The new algorithm in my mpi implementation is N + (N/P)^3 + N^2. I first split the workload by N based on the id number of each processor. Then I parrelized the norm loop by only doing work if a row corresponded to its assigned processor which results in (N/P)^3 iterations in the loop. Afterwards, processor 0 would do back substitution by itself which would be N^2.

Iterations

In another iteration, I tried to assign alternating rows to each processor within the norm loop skipping every nproc row. This iteration accomplished the same goal of (N/P)^3 iterations as my current algorithm, but there was significant overhead with the messaging. In this iteration, I waited until all gaussian reduction calculations were done to send data from the non-zero processors back to processor 0. The problem here was that the runtime was bottlenecked by processor 0 doing MPI_Recv sequentially for every other processor which took longer than the actual calculations. My current iteration fixes this problem by sending data immediately to processor 0 after doing a calculation which ensures that the send and receive is also being done in parallel.

Efficiency

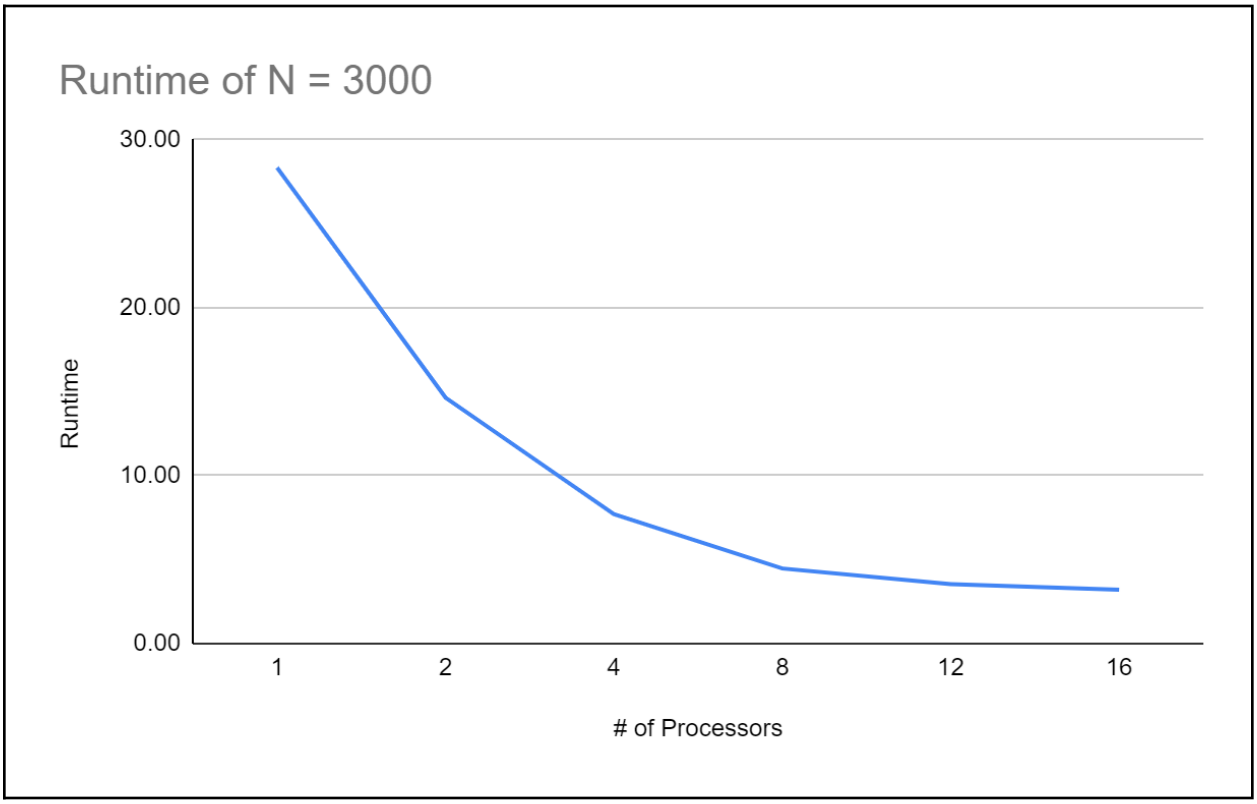
I had processor 0 divide the matrix into workloads for each processor and do back substitution. For the main gaussian reduction calculations, processor 0 and the other nproc processors would all work on it at the same time. Message sending and receiving would also occur immediately after calculations which reduced the overhead time for communication.

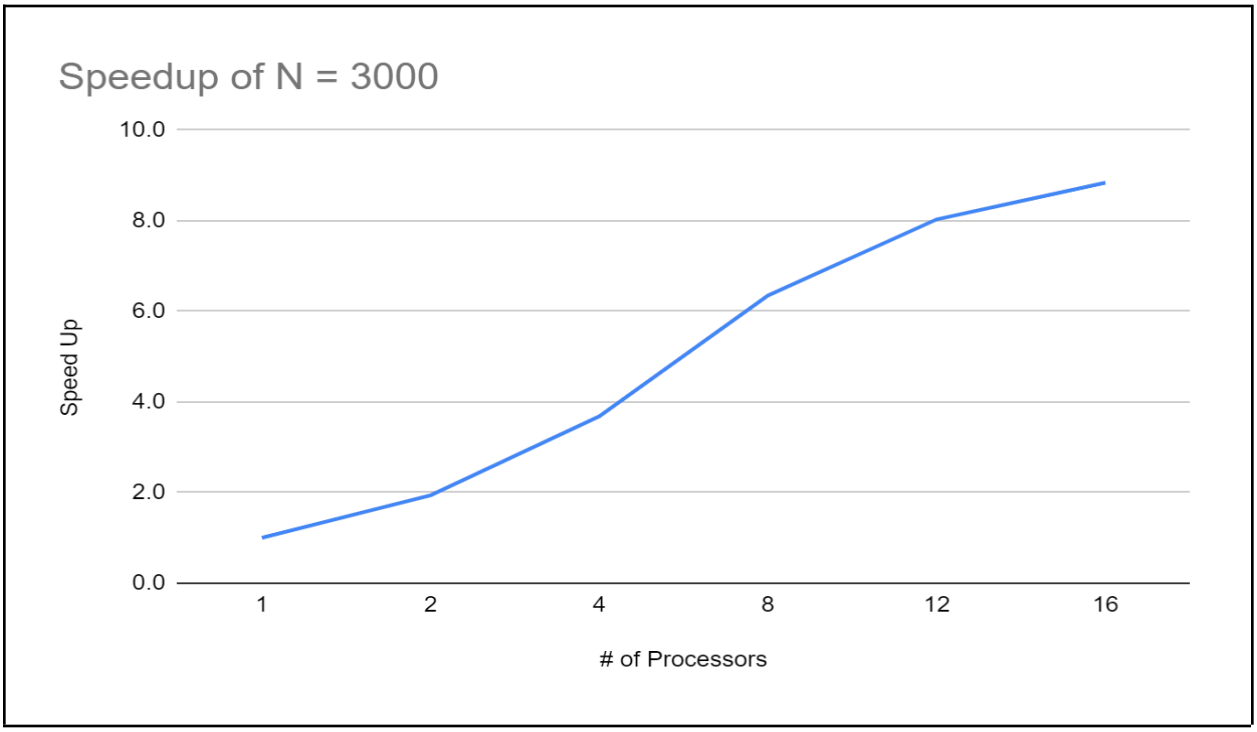
Correctness

I know my solution is correct because I divided the N rows of the matrix based on processor id and sent those rows only to those processors. The initial matrix of those processors are empty before they receive the data, so there was no data overlap or race condition in my solution. In the norm loop, a processor would only do calculations on the row if the row number divided by nproc matches its processor id. Also in the README, I ran some correctness checks on small matrices to ensure that the output was the same for serial and MPI.

Performance Analysis

Data can be found in README





MPI	Processor	Runtime	Speed up
	1	28.34	1.0
	2	14.64	1.9
	4	7.71	3.7
	8	4.47	6.4
	12	3.54	8.0
	16	3.21	8.8

The MPI code is scaling. The more processors there are the quicker the runtime. Speed up is nearly linear at first being 1.9 from 1 processor to 2 processors and then increases as expected until 12 to 16 processors where it plateaus. At this point, it seems like the algorithm reaches its maximum efficiency for N = 3000.