Cpt S 411 Assignment Cover Sheet

Assignment #3

Team Members:

Lorphanpaibul, Allison

Nate Jensvold

I[[1]](#footnote-0) certify that I have listed above all the sources that I consulted regarding this assignment, and that I have not received or given any assistance that is contrary to the letter or the spirit of the collaboration guidelines for this assignment. I also certify that I have not referred to online solutions that may be available on the web or sought the help of other students outside the class, in preparing my solution. I attest that the solution is my own and if evidence is found to the contrary, I understand that I will be subject to the academic dishonesty policy as outlined in the course syllabus.

Please print your names.

Allison Lorphanapaibul

Nate Jensvold

Today’s Date:

October 27, 2020

1. **Data Tables:** 
   1. MyAllReduce

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| MyAllReduce (Computation time per process) | | | | | | | |
| Size of Array (thousands) | Number of Processes | | | | | | |
|  | 1 | 2 | 4 | 8 | 16 | 32 |
| 1 | 109.377 | 133.341 | 176.844 | 240.4695 | 673.003 | 1328.554 |
| 2 | 186.504 | 178.185 | 178.966 | 243.8116 | 773.4122 | 1339.977 |
| 4 | 315.599 | 245.2395 | 218.2738 | 306.606 | 662.565 | 998.3536 |
| 8 | 552.329 | 368.9955 | 291.84 | 286.1873 | 873.5209 | 1336.389 |
| 16 | 1036.277 | 615.618 | 417.0878 | 337.8332 | 729.1331 | 1282.914 |
| 32 | 1998.091 | 1096.587 | 656.4847 | 456.888 | 879.2376 | 1417.595 |
| 64 | 3963.703 | 2056.689 | 1160.363 | 726.6446 | 897.5224 | 1517.594 |
| 128 | 7886.785 | 4044.419 | 2138.575 | 1255.432 | 1260.011 | 1621.177 |
| 256 | 15738.05 | 7950.989 | 4058.08 | 2199.308 | 1740.545 | 1276.242 |

* 1. MPI\_AllReduce

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| MPI\_AllReduce (Computation time per process) | | | | | | | |
| Size of Array (thousands) | Number of Processes | | | | | | |
|  | 1 | 2 | 4 | 8 | 16 | 32 |
| 1 | 96.459 | 146.7715 | 138.3918 | 197.0055 | 771.0047 | 833.857 |
| 2 | 164.58 | 195.987 | 167.4947 | 166.5146 | 571.3855 | 1189.205 |
| 4 | 290.656 | 235.9095 | 209.562 | 213.6303 | 767.6932 | 1441.252 |
| 8 | 533.748 | 362.9425 | 293.671 | 269.3251 | 717.9629 | 1266.35 |
| 16 | 1023.985 | 583.015 | 374.7388 | 407.4421 | 702.9888 | 1182.432 |
| 32 | 2306.991 | 1083.751 | 774.9893 | 457.3216 | 793.2448 | 1350.686 |
| 64 | 3951.334 | 2067.527 | 1139.96 | 777.0364 | 1011.143 | 1440.913 |
| 128 | 7870.284 | 3997.578 | 2080.156 | 1241.958 | 1218.183 | 1632.346 |
| 256 | 15721.74 | 7947.679 | 4089.864 | 2264.048 | 1770.574 | 1772.151 |

* 1. NaiveAllReduce

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| NaiveAllReduce (Computation time per process) | | | | | | | |
| Size of Array (thousands) | Number of Processes | | | | | | |
|  | 1 | 2 | 4 | 8 | 16 | 32 |
| 1 | 78.940 | 142.500 | 9891.423 | 9999.198 | 9980.877 | 9776.363 |
| 2 | 137.591 | 176.029 | 9899.551 | 9956.870 | 9804.268 | 9985.577 |
| 4 | 260.747 | 245.692 | 9909.648 | 9815.531 | 9867.788 | 9829.731 |
| 8 | 520.047 | 364.063 | 9968.637 | 10004.383 | 9860.551 | 9940.606 |
| 16 | 1011.522 | 606.502 | 10164.385 | 10059.582 | 9862.319 | 10111.381 |
| 32 | 1959.548 | 1071.402 | 10357.228 | 10073.516 | 10033.878 | 9981.259 |
| 64 | 3926.541 | 2058.746 | 10886.701 | 10360.637 | 9984.956 | 9953.475 |
| 128 | 7840.171 | 4028.549 | 11814.793 | 10831.498 | 10371.554 | 10215.372 |
| 256 | 15675.887 | 7937.026 | 13895.516 | 11835.319 | 10408.238 | 10474.105 |

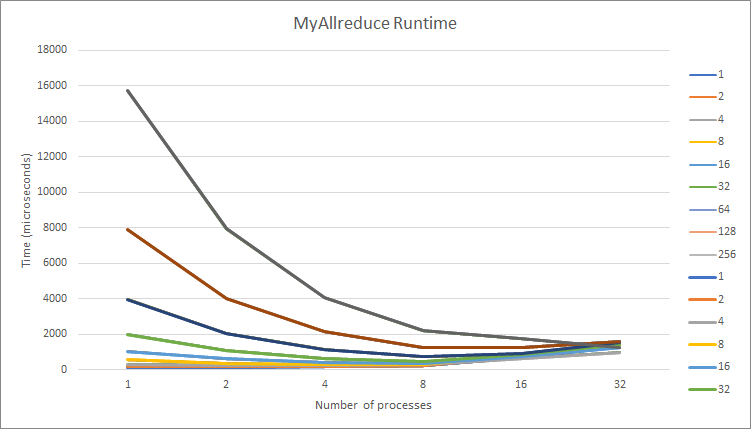
1. **Parallel Runtime:**

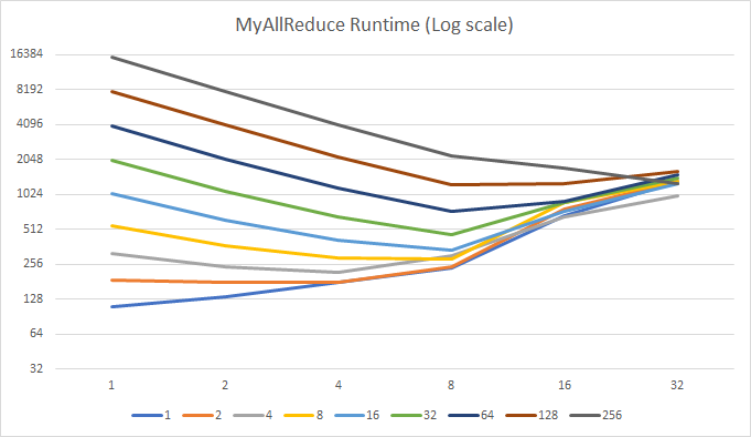
Below are the runtime curves for each implementation of the reduce function. Each line on the graphs represents a different input size that we tested. For observation purposes, both the original and logarithmic scale runtime graphs were included for each implementation.

In general, the runtime curves for the implementations would slope downward as the number of processes increased. This was a good indication that the functions were parallelized successfully.

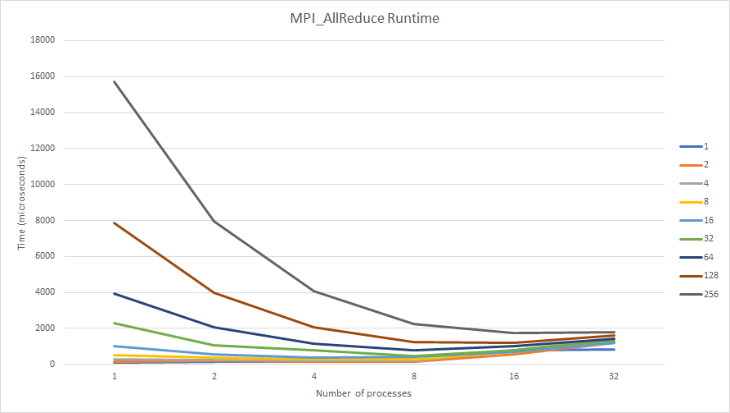
Taking a closer look at the runtime curves, we can compare the different implementations more easily:

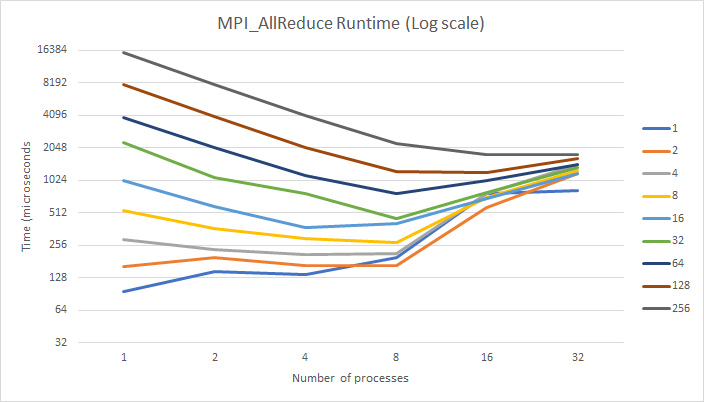
* 1. MyAllReduce





* 1. MPI\_AllReduce





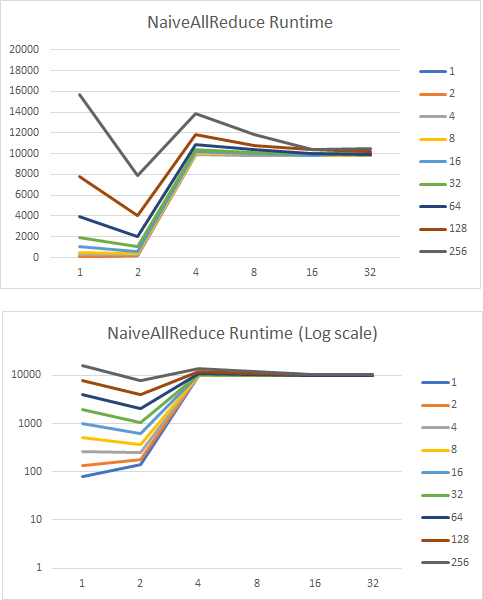
Runtime curves for both MPI and myAllReduce were as expected. Both produced very similar curves for each size, and most of the curves generally had a downward slope. Since these implementations were supposed to run at similar runtimes, the similarity was to be expected.

Viewing each line individually, it is clear that larger sizes were able to take advantage of the additional processors more than lower input sizes. Additionally, lower input sizes would actually see a runtime increase with more processors.

* 1. NaiveAllReduce

The Runtime curves for the naive reduction implementation did not follow the ideal downward curve; however, this was to be expected. The naive implementation does not optimize its processors very well, so the runtime actually increases a large amount with increased processors.

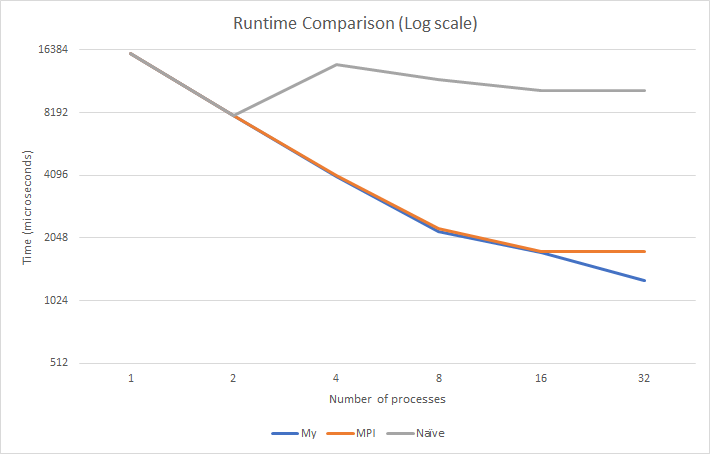
Another thing to observe with the curves is that it actually did see a decrease in runtime at two processors, before sharply increasing at four. Also, all of the curves began to plateau at 16 processes.



* 1. Runtime Comparisons

Below shows a comparison of the three implementations at input size 256 thousand. We can observe that the MPI and MyAllReduce functions had very similar runtime results, while the Naive had very different results.

Interestingly, the Naive curve had similar results until two processors, before it diverged upwards from the other curves.

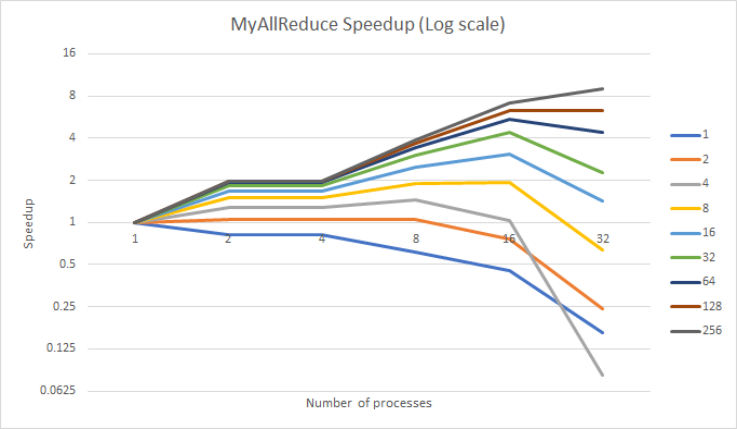


1. **Speedup**

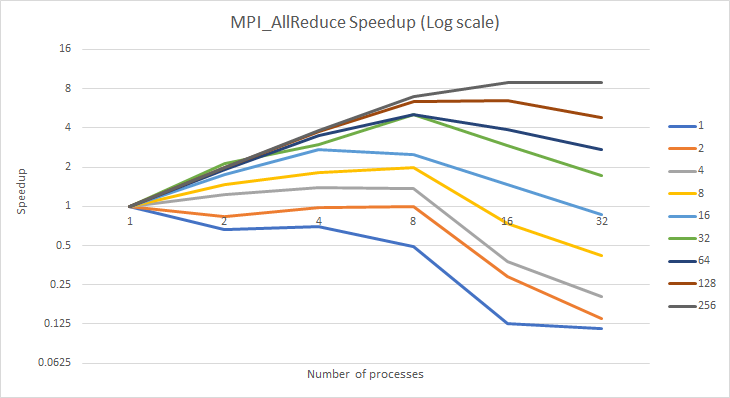
The speedup curves were created by using the runtime tables from above. Similar to the runtime graphs, each graph is for a different implementation, and each line is for a different input size.

In general, the results for the speedup curves were varied depending on the size.

* 1. MyAllReduce

****

* 1. MPI\_AllReduce

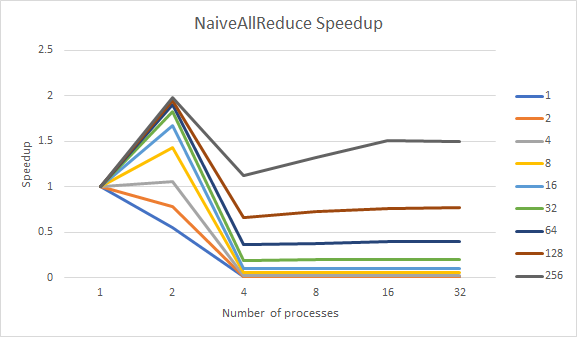


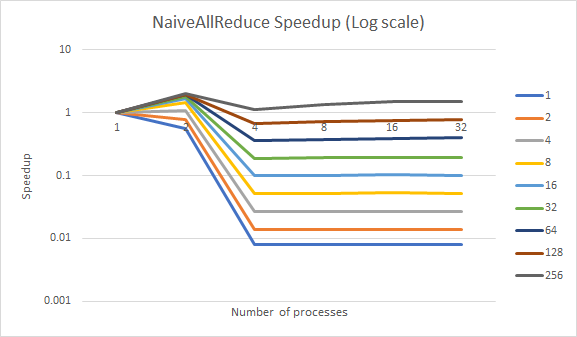
Speedup for the MPI and myAllReduce functions yielded similar curves. As with the runtime curves, larger input sizes were able to utilize having more processors better than smaller input sizes. This can be seen in the curves as larger input sizes trend upwards while lower input sizes trend downwards.

While the two implementations did provide similar speedups, at higher input sizes, the myAllReduce would reach higher speedups. At size 256, the myAllReduce was able to reach a peak speedup of 9, where the MPI implementation reached a peak speedup of 8.

Although the myAllReduce performed better at higher input sizes, the MPI function on average had higher speedups across the board.

* 1. NaiveAllReduce





As expected, the NaiveAllReduce implementation did not reach the level of speedups as the last two. The highest speedup for the NaiveAllReduce was two.

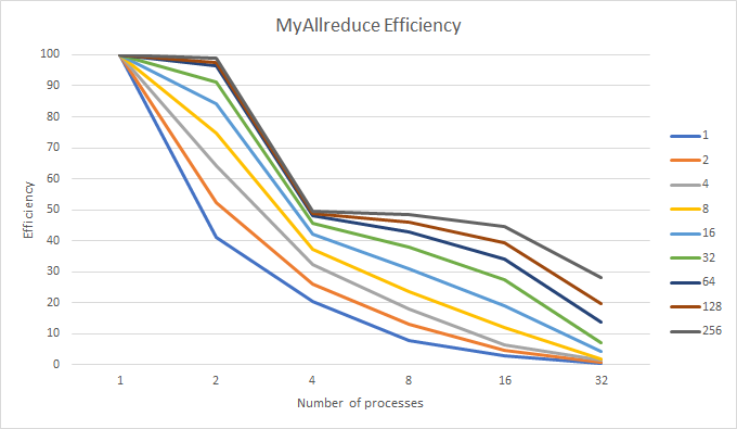
Something we found interesting was how quickly the speedup plummeted at four processors.Also, all the curves would plateau at four processors, similar to the runtime curves for the naive function.

1. **Efficiency**

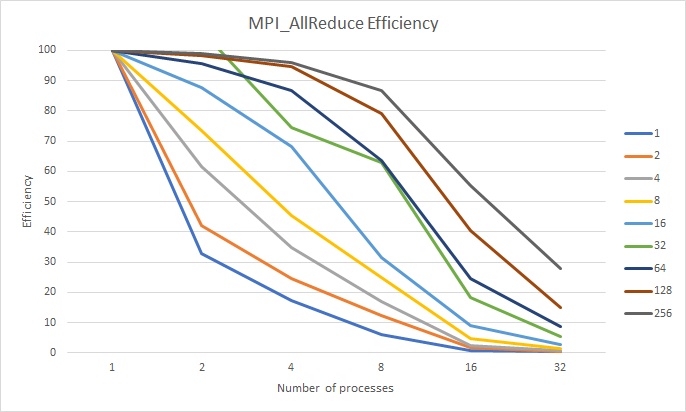
The efficiency curves were created using the data from the speedup curves. As with before, there is a graph for each implementation and line for each input size.

In all three graphs, there was a downward trend. This is to be expected in efficiency graphs. How much and how quickly the efficiency goes down can tell us how well the graph was using the resources given.

* 1. MyAllReduce



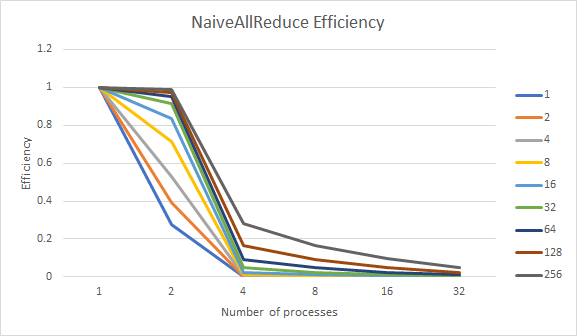
* 1. MPI\_AllReduce

****

The efficiency curves are the first time we noticed a big difference between the MPI and myAllReduce implementations. The MPI library’s function performed much better, keeping efficiency high for a much longer time before decreasing. On the other hand, the myAllReduce function had a sharp increase at four processors for larger input sizes.

This difference was much less noticeable in smaller input sizes. Looking at size 1000 and 16000, the curves are very similar. These results were interesting considering the MyAllReduce usually had better speedups at higher sizes.

* 1. NaiveAllReduce



As with the previous results, the Naive implementation performed much worse in efficiency than the optimized reduction functions. The efficiency would plummet to 20% at higher sizes and as low as 10% for lower sizes. This decrease in performance was expected in the naive function.

1. **Conclusion**

Going into this experiment we felt that the reduction algorithm was a prime candidate for parallel performance scaling. We expected the MyAllIReduction and MPIAllReduction to both scale very well with a large input size (n) and high number of processes(p), where the cost of communication could be amortized by the increased computation time. We also expected the NaiveReduction implementation to have some speedup with large p and n but very poor performance on small p and n where the extra cost of communication would overshadow any gains in computation efficiency.

Comparing these expectations to the results, the assumptions holds true for the most part as we observed a speedup of up to 8.8 for the MPIAllReduce, and a speedup of 9.0 for the MyallReduce implementations, where input size was 256k and 32 processes were used. Additionally when the NaiveAllReduce algorithm was used with the same input size and number of processes a speedup of only 1.49.

Looking over our data we noticed several things that immediately jumped out as us. When the MPIAllReduction and MyAllReduction algorithms were run using 8 and 16 processes there was an abnormal increase in computation time. On average the computation increase between 1, 2, 4 and 8 processes was a factor of 1.5-2x while the jump between 8 to 16 processes was over 7x. While we could normally attribute this increase due to the network latency that occurs when sending data between different nodes, the data that we noticed these increases on had already had the network communication times removed so it only represented computation time. That means that there are computation increases that occur as processors go up even as when the input size is kept constant. Additionally we were only able to run the algorithms on 32 processes/4 nodes instead of 64 processes/8 nodes due to the cluster having hardware failure across two of the nodes during our data collection.

Overall the results from all 3 of the algorithms follow the scaling expected of parallel algorithms. The relationship between p(number of processes) and n(input size) also holds true. As p increases, computation time increases proportional to a serial implementation. However even though computation time increases proportional, on very low input sizes the communication overhead completely offsets the computation times saved meaning the overall time performance of parallel implementation is very poor. Also as expected on a higher number of processes when the input size was large speedup became the most noticeable while efficiency was at its greatest.

1. If you worked as a team, then the word “I” includes yourself and your team members. [↑](#footnote-ref-0)