Cpt S 411 Assignment Cover Sheet

Assignment #2

For team projects:

List of all students (Last, First): (Valdez, Paul), (Hellwig, Benjamin)

List of collaborative personnel (excluding team participants):
Ananth Kalyanaraman

NOTE:
I[1] 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.

Assignment Project Participant(s):
Paul Valdez
Benjamin Hellwig

Today's Date:
September 16th, 2019

[1] If you worked as a team, then the word "I" includes yourself and your team members.

a) Concise presentation of the results (in a way the performance results are easier to understand). Results should include three components: Parallel Runtime, Speedup, and Efficiency. It is best to tabulate these calculations in an excel or similar spreadsheet, and then plot them.

Below, we show the results for the calculations of Efficiency and Speedup for n = 1029 and n = 2^{29}, with varying numbers of processors.

MyReduce():

Procs	Speedup-1024	Efficiency-2_29	Speedup-2_29	Efficiency-1029	Parallel_1024	Parallel_2_29
64	0.030588688	0.000477948	24.23947783	0.378741841	519.8	86355.8
32	0.031422925	0.000981966	31.70218969	0.990693428	506	66027.6
16	0.042558887	0.00265993	15.43279549	0.964549718	373.6	135634.5
8	0.043729373	0.005466172	7.995084665	0.999385583	363.6	261813.3
4	0.063701923	0.015925481	3.539504028	0.884876007	249.6	591387.8
2	0.105437666	0.052718833	1.997337322	0.998668661	150.8	1048005
1	1	1	1	1	15.9	2093219.5

NaiveReduce():

Procs	Speedup-1024	Speedup-2_29	Efficiency-1029	Efficiency-2_29	Parallel_1024	Parallel_2_29
64	0.002331932	41.01767214	3.64E-05	0.640901127	5017.3	51040.8
32	0.004694647	29.94791381	0.000146708	0.935872306	2492.2	69907.2
16	0.010483871	15.69966015	0.000655242	0.981228759	1116	133351.6
8	0.027863777	7.971220107	0.003482972	0.996402513	419.9	262641.7
4	0.074856046	3.996297616	0.018714012	0.999074404	156.3	523878.6
2	0.428571429	1.999393948	0.214285714	0.999696974	27.3	1047104.7
1	1	1	1	1	11.7	2093574.8

AllReduce():

Procs	Speedup-1024	Speedup-2_29	Efficiency-1029	Efficiency-2_29	Parallel_1024	Parallel_2_29
64	0.230941704	51.50358524	0.003608464	0.804743519	44.6	82435.3
32	0.238425926	30.42774071	0.00745081	0.950866897	43.2	139534.3
16	0.168026101	15.57737591	0.010501631	0.973585994	61.3	272556.4
8	0.274666667	7.970247719	0.034333333	0.996280965	37.5	532695.3
4	0.331189711	3.991522566	0.082797428	0.997880642	31.1	1063682.7
2	0.432773109	1.998295122	0.216386555	0.999147561	23.8	2124667.9
1	1	1	1	1	10.3	4245713.5

Once calculated, we graphed Parallel Runtime vs Processors, Speedup vs Processors and Efficiency vs Processors. Below in each figure we will show the graphs for both n = 1024 and n = 2^{29}.

Figure 1. (NaiveReduce() Parallel Runtime):

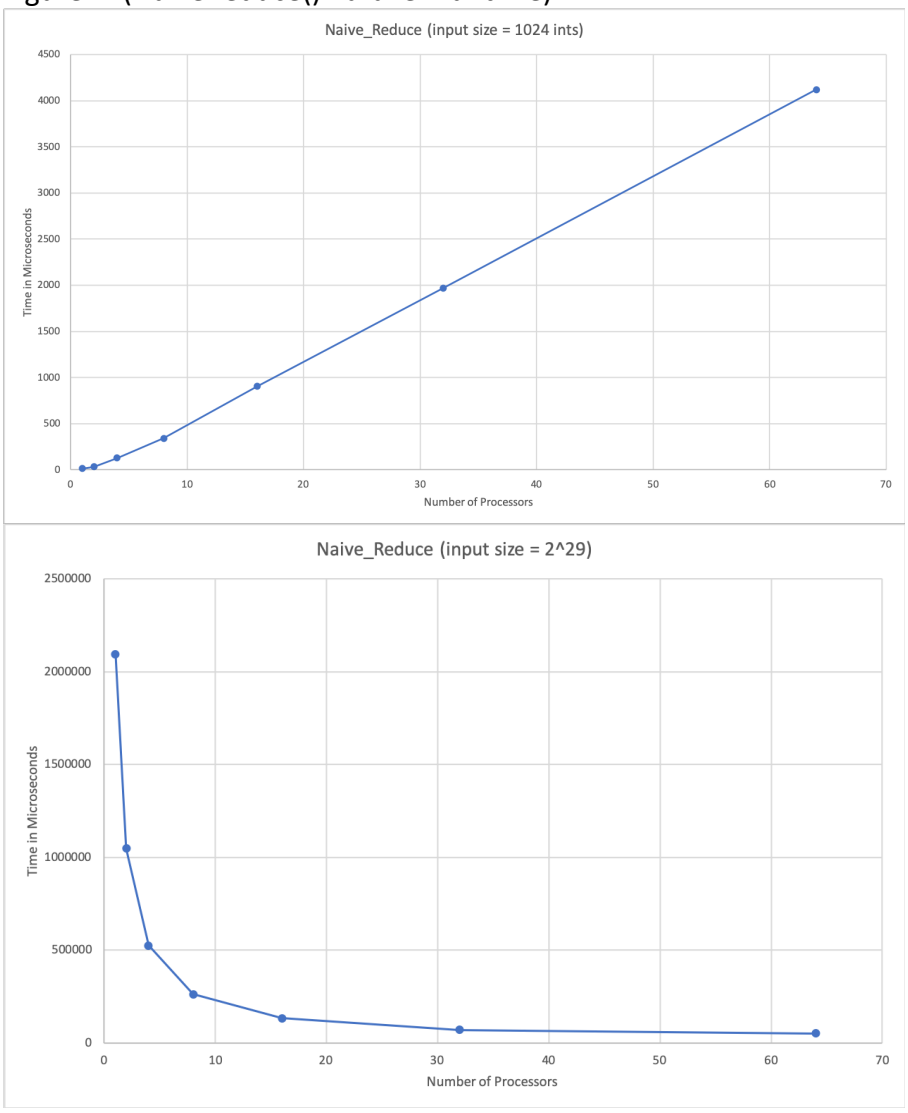


Figure 2. (NaiveReduce() Speedup):

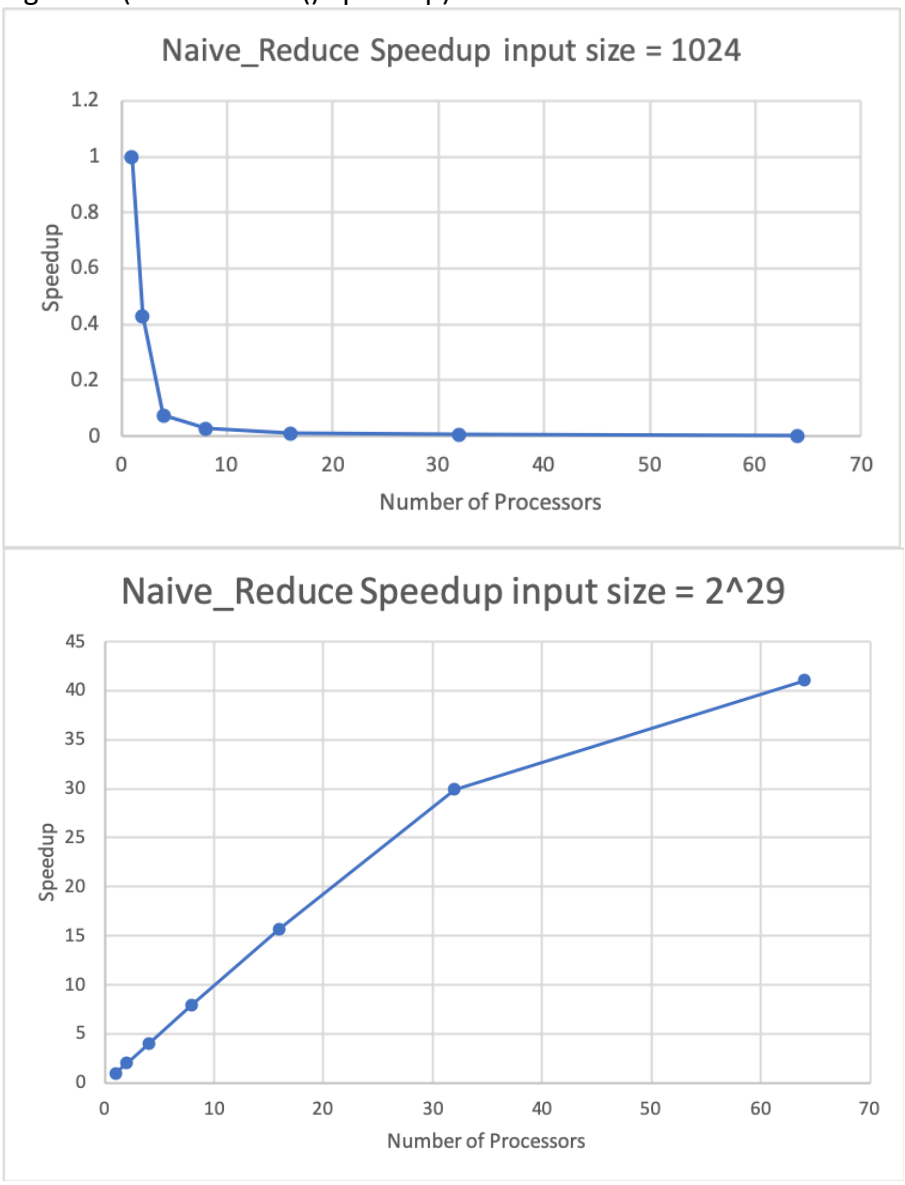
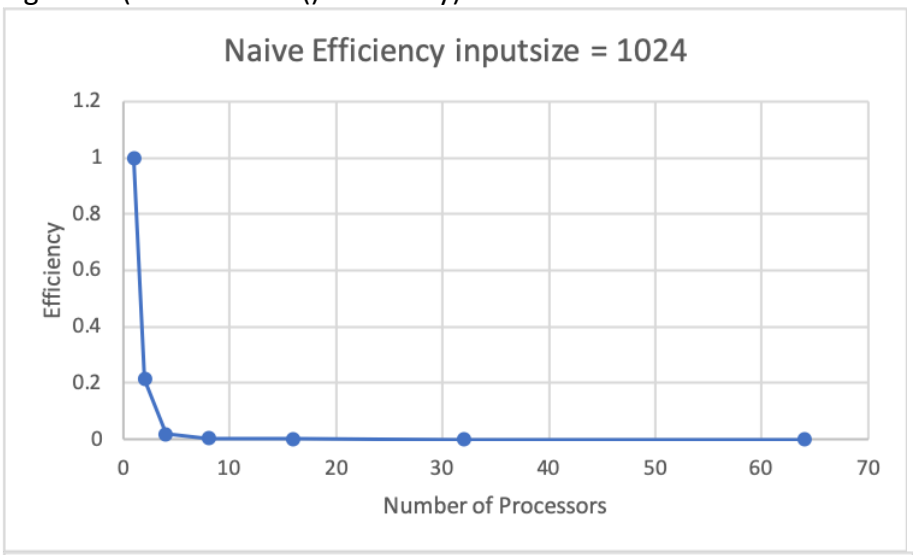


Figure 3. (NaiveReduce() Efficiency):



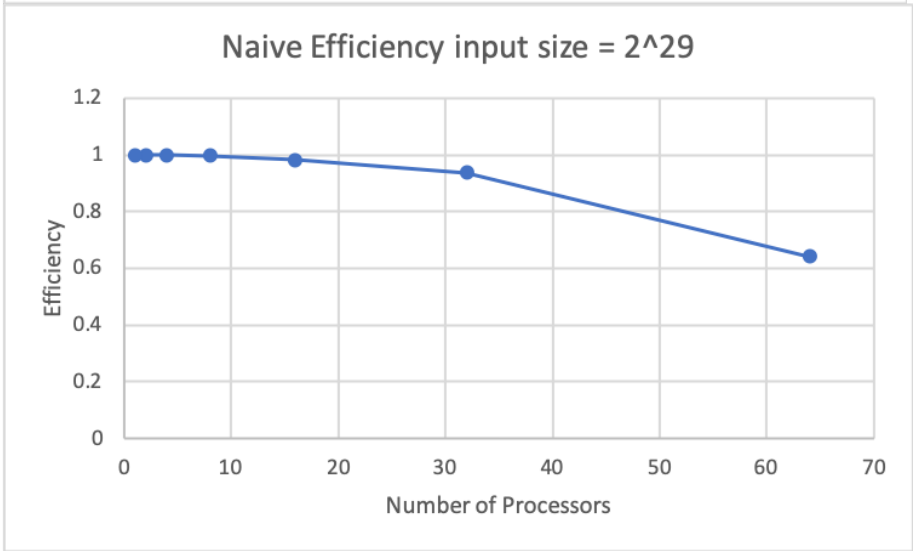


Figure 4. (MyReduce() Parallel Runtime):

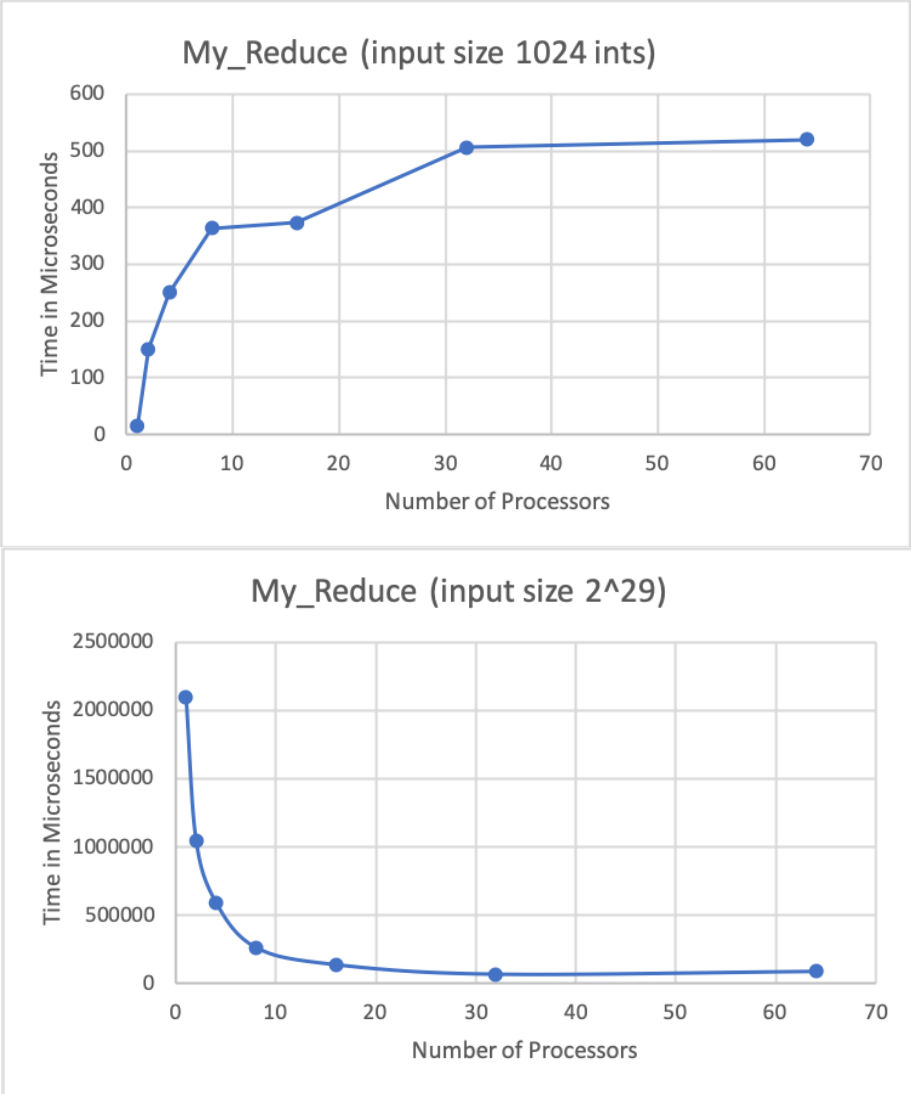


Figure 5. (MyReduce() Speedup):

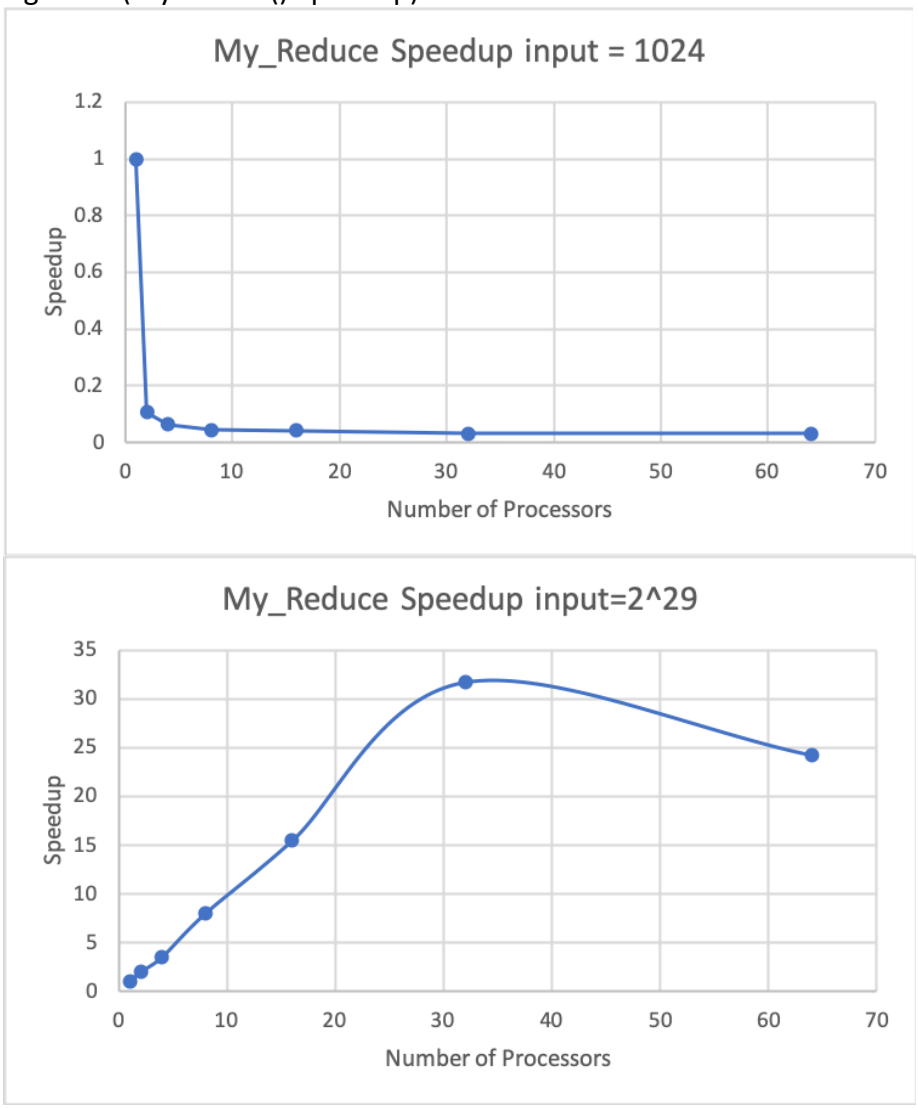
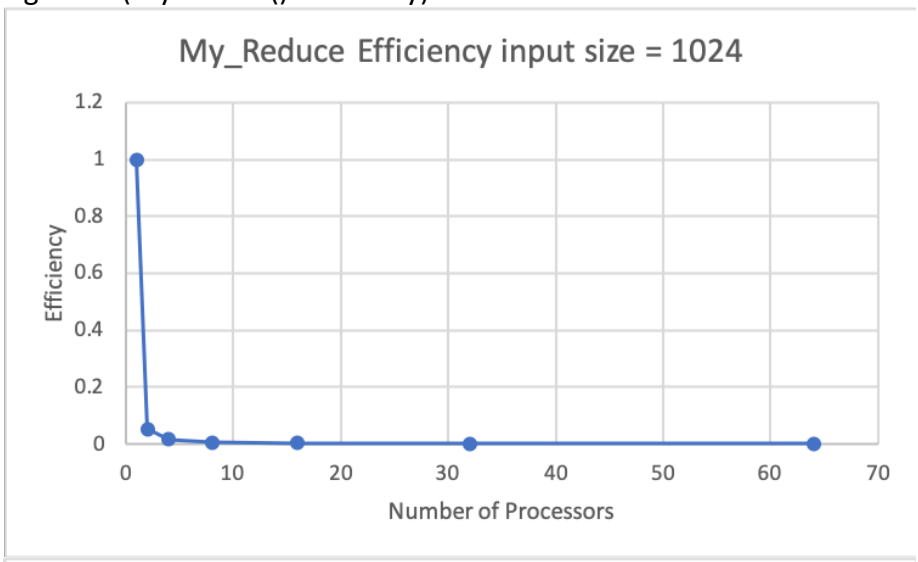


Figure 6. (MyReduce() Efficiency):



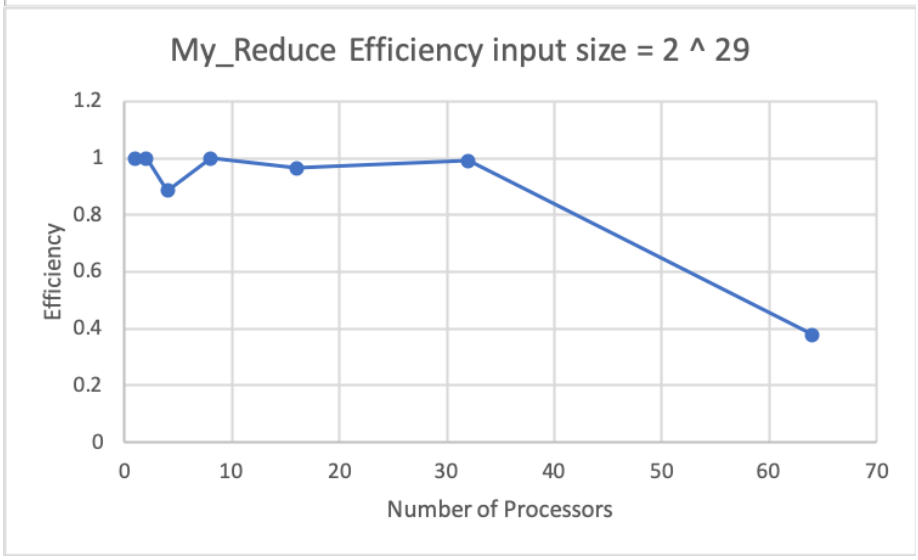
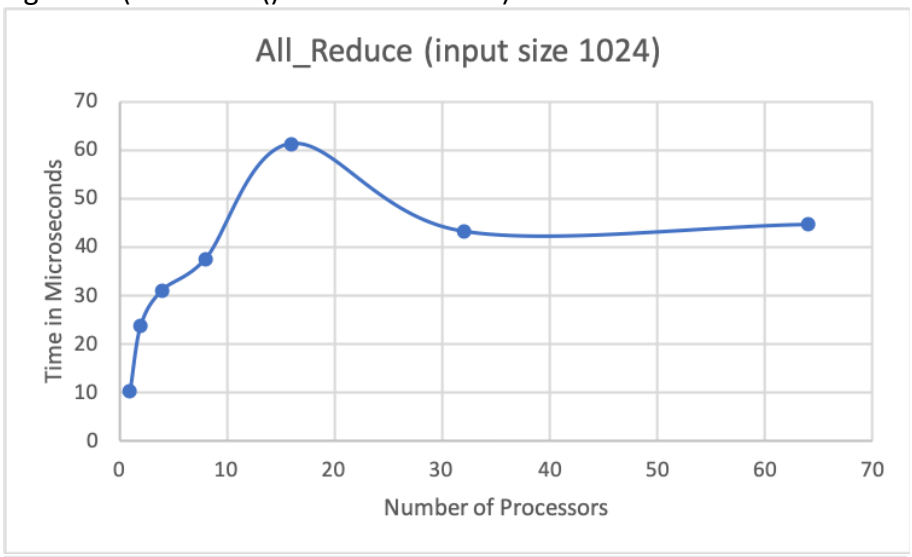


Figure 7. (AllReduce() Parallel Runtime):



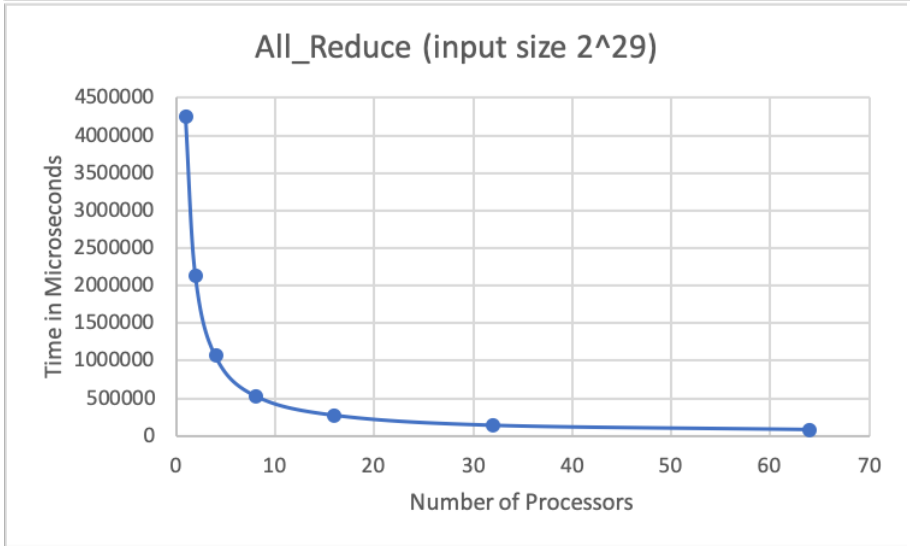
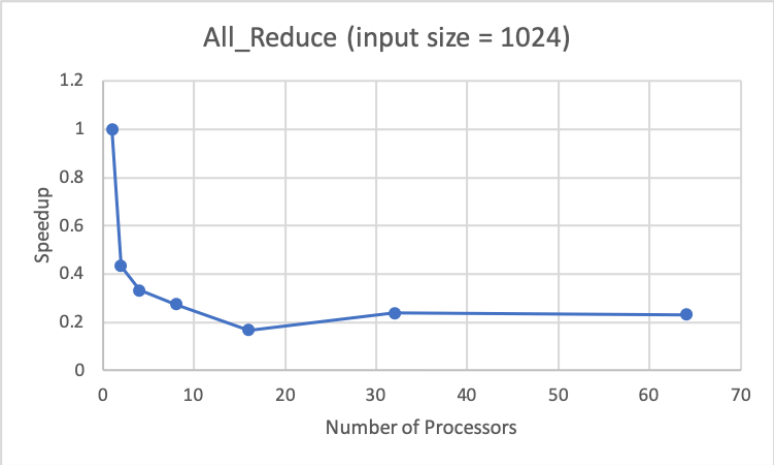


Figure 8. (AllReduce() Speedup):



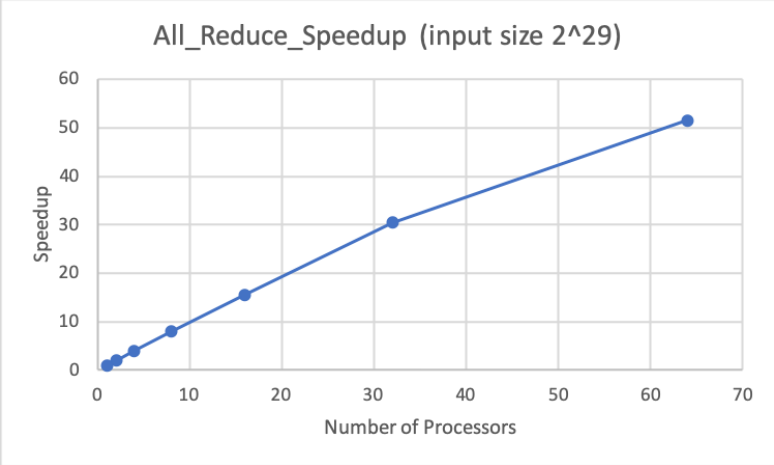
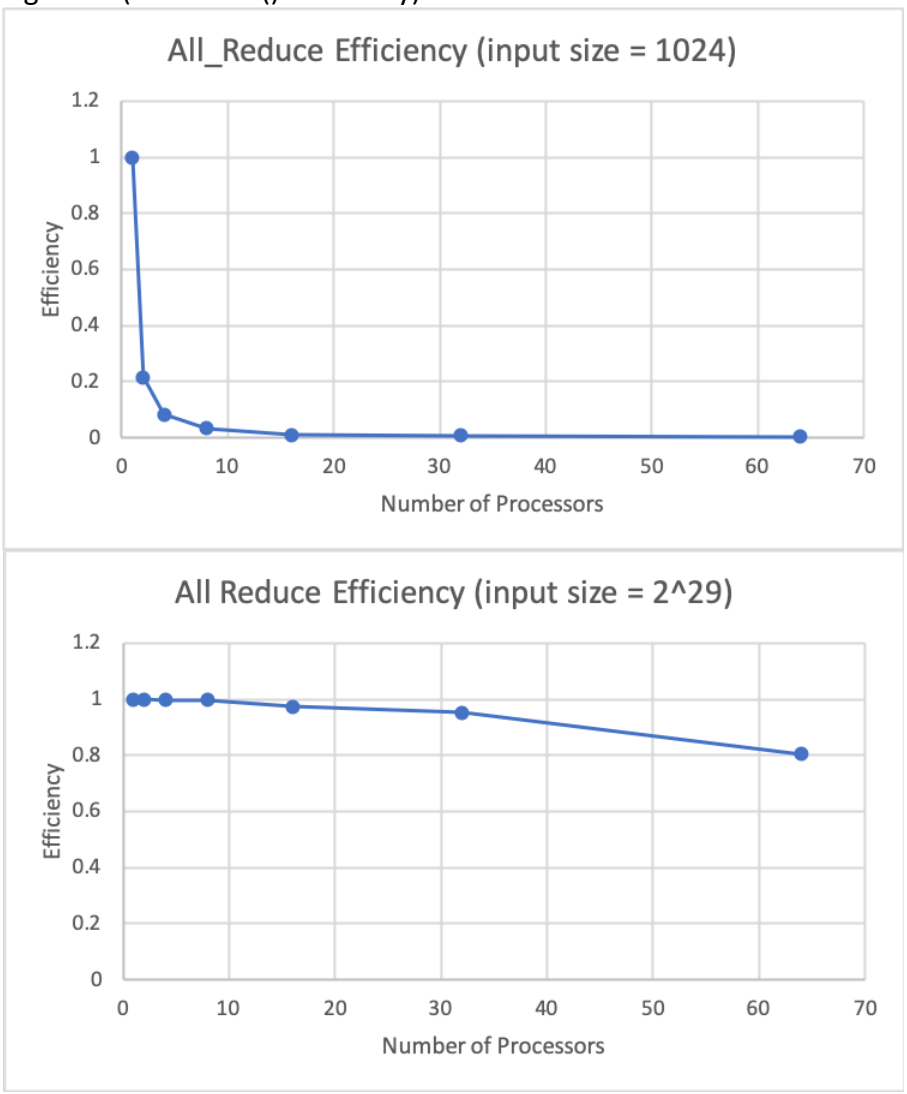


Figure 9. (AllReduce() Efficiency):



b) Clear statement of what you observed vs. what you were expecting in terms of performance scaling (as a function of *p* and as a function *n*)

For all three algorithms implemented, we observed that for small arrays and a large number of processors, we had slower running times than for small arrays with fewer processors. We hypothesize this is because the time saved by computing the sums in parallel does not outweigh the time lost from communicating between nodes for small input sizes. However, the results were as expected once the input sizes increased; for large input sizes, runtime decreased as the number of processors increased. We have included graphs for both extremes. Likewise, speedup decreased as a function of p with small input sizes, and as expected, it increased as a function of p once the input size became very large, though the growth was sublinear (though this is also expected because the absolute best performance for these algorithms would lead to linear speedup). In terms of efficiency, the parallel algorithms were extremely inefficient for small input sizes. At large input sizes, efficiency was better for fewer processors and gradually decreased as p grew larger. This is also expected because efficiency can only go down for a fixed input size as the number of processors increases. One thing we noticed that we cannot explain is the tendency of the NaïveReduce algorithm to perform faster than the other two algorithms once the input size grows very large. We can only guess that for very large input sizes, the difference between the NaiveReduce and other two algorithms should decrease (since the vast majority of the runtime is spent in the serial portion of the code), but we are unsure why the NaiveReduce algorithm ran more quickly overall. It is possible there is a bug in our code.

c) Statement of any experimental setup issues or assumptions (if any) – for instance, for some input size if you decided to not run because it was taking too long (say >15 mins) then you should state that.

We realized after calculating all of our parallel runtimes, speedup, and efficiency, that for naïve reduce, we had forgotten to send back the sum to each processor. Thus, all of these values will be off slightly. We fixed our code and re-ran it on 64 processors, but the run time for large arrays was still faster than the runtimes of the other two algorithms. We would have re-run this algorithm for each p and charted/graphed the updated results, but we ended up not having enough time to do this.

In our experimental setup, we tested our code using SUM. However, we also added implementation to do MAX. The way you specify this is passing in MAX as an argument in the bash script.

When running, we also had to use -lm to include the math library upon compiling the code.