Cpt S 411 Assignment Cover Sheet

(To be turned in along with each homework and program project submission)

Assignment # 3

For individual assignments:

Student name (Last, First): Barik, Reet

For team projects:

List of all students (Last, First): Barik, Reet

List of collaborative personnel (excluding team participants):

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): Barik, Reet

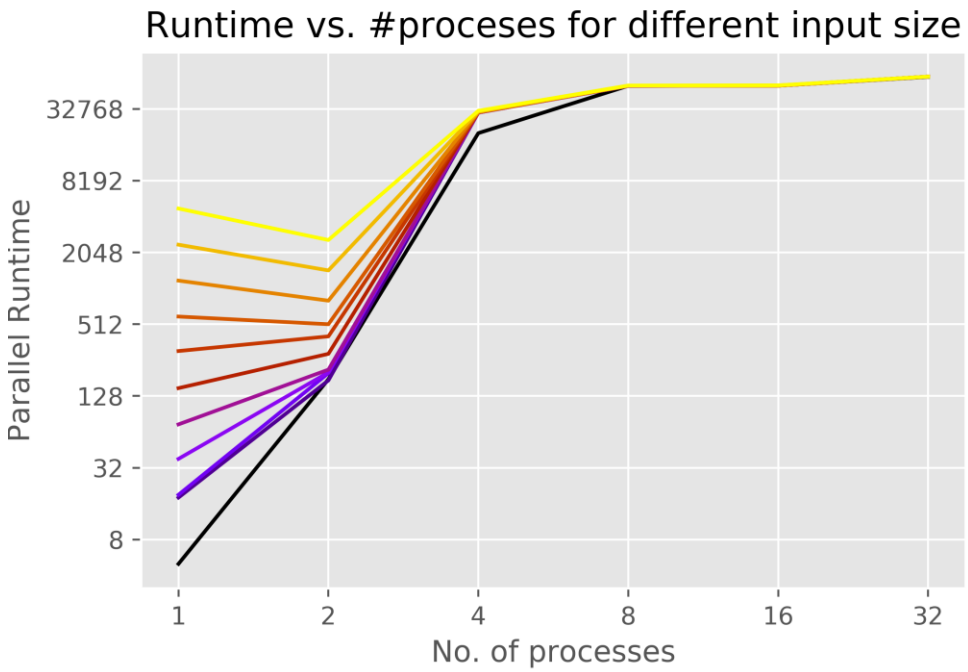
Today's Date: 26 October 2020

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

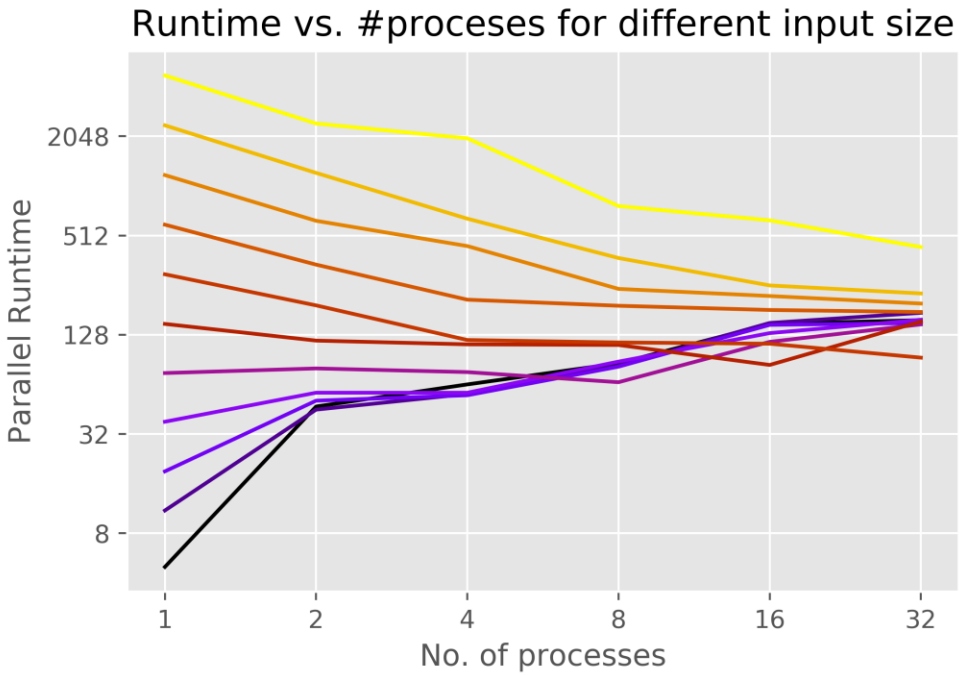
The required plots are as follows: [The binary associative operator used throughout is ADD]

1. Parallel Runtime (in microseconds)

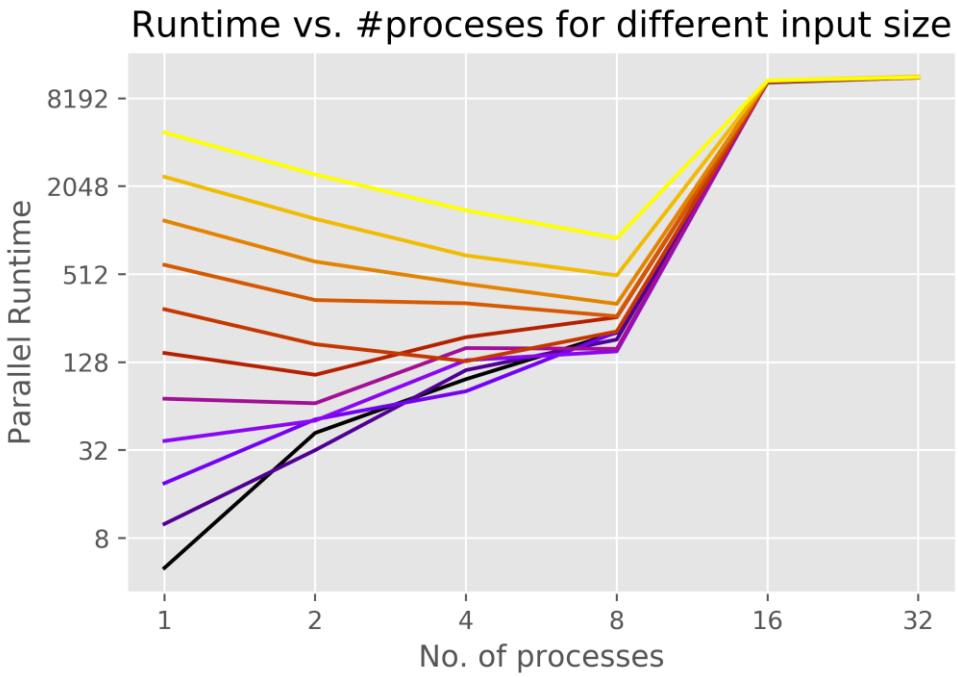
a. NaiveAllReduce



b. MyAllReduce



c. LibraryAllReduce

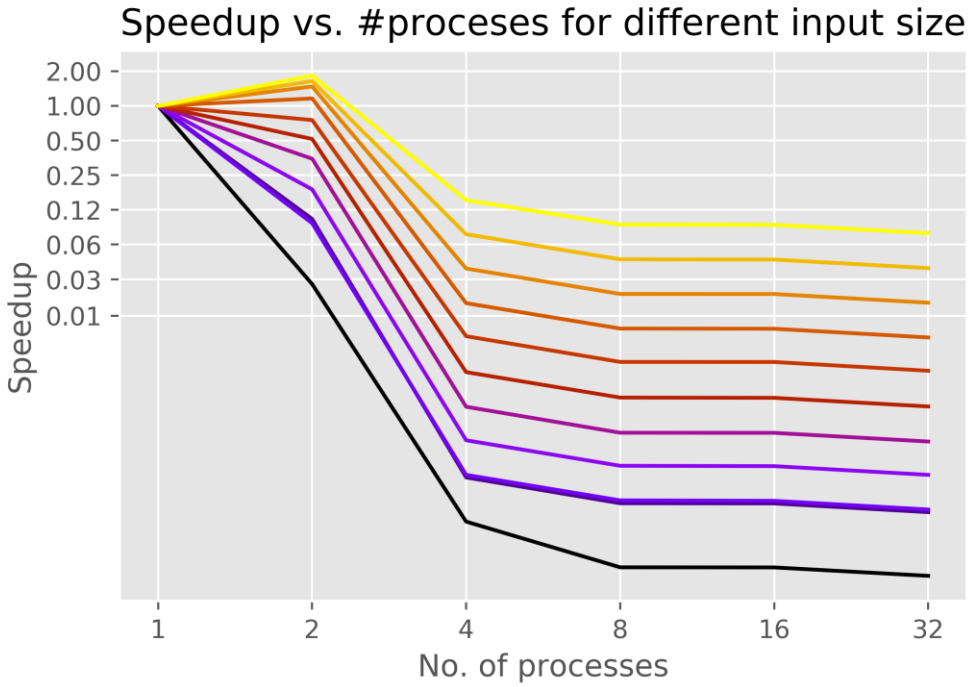


Inference:

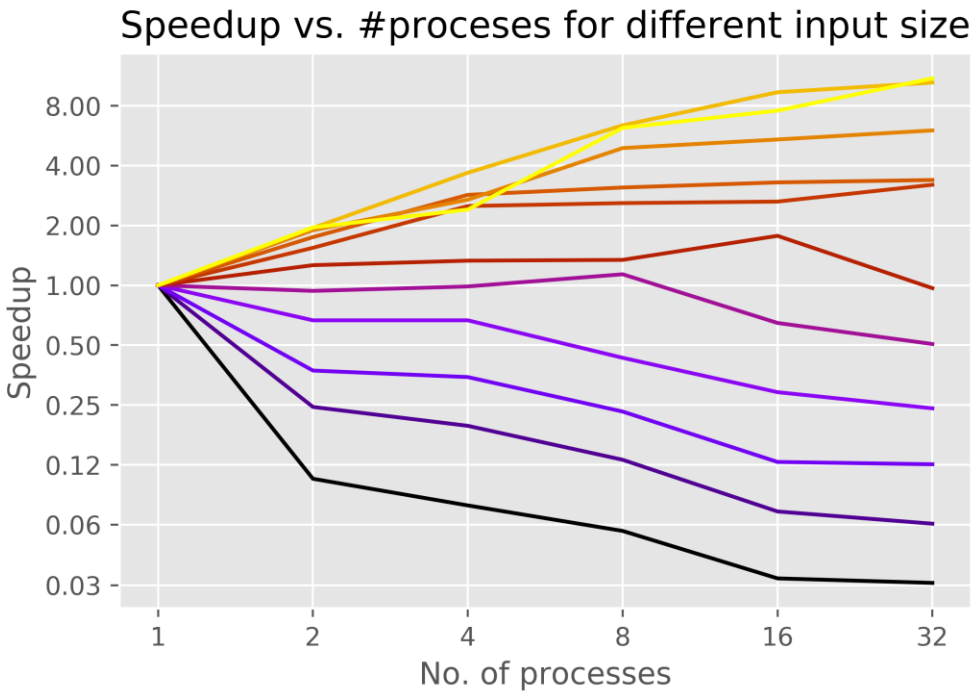
1. The parallel runtimes for all versions of 'AllReduce' follow the trend of decreasing performance for smaller of the input sizes (n < 64K), wherein, the time taken increases with increasing the number of processors.
2. In the naïve version, the serial case (p = 1) takes the least time (except for the biggest 3 input sizes) since it does not involve the added communication costs of sending the partial sums from one process to the one with the next rank.
3. For 'NaiveAllReduce', for #processes > 2, the runtime doesn't change much across the different input size. This can be explained by the fact that at that point, the communication costs become the dominant factor and hence the computation cost which is a function of the input size becomes negligible.
4. The deteriorating performance (increasing runtime) for smaller inputs and improvement (decreasing runtime) for larger inputs can be seen in both 'MyAllReduce' and 'LibraryAllReduce'.
5. For 'LibraryAllReduce', I executed the code using the command '*sbatch -n <#processes> sub.sh*' because the command '*sbatch -N <#nodes> -n <#processes> sub.sh*' was resulting in a net slowdown as soon as the number of nodes was increased beyond 1 (which is very weird).
   The above can be observed from the significantly higher runtimes for p = 16 upwards for 'LibraryAllReduce'.
   Interestingly, though comparable to 'LibraryAllReduce' in terms of performance (for #processes ≤ 8) this bottleneck is not seen in the results for 'MyAllReduce'. I am yet to figure out how 'MyAllReduce' is outperforming 'LibraryAllReduce' for #processes ≥ 8 [I executed both 'NaiveAllReduce' and 'MyAllReduce' using '*sbatch -N <#nodes> -n <#processes> sub.sh*'].
6. The spread of values for each 'p' keeps decreasing faster for 'LibraryAllReduce' than 'MyAllReduce'. This means 'LibraryAllReduce' becomes an overkill faster (increasing #processes) for smaller input sizes and scales better for larger input sizes than 'MyAllReduce'. This is in keeping with the expectations.

2. Speedup

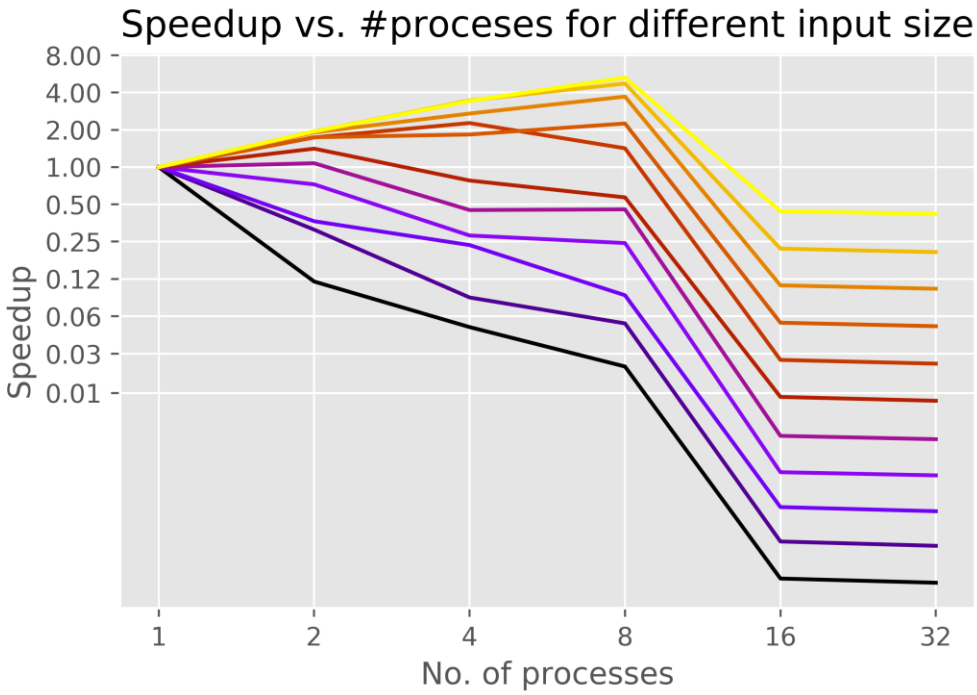
a. NaiveAllReduce



b. MyAllReduce



c. LibraryAllReduce

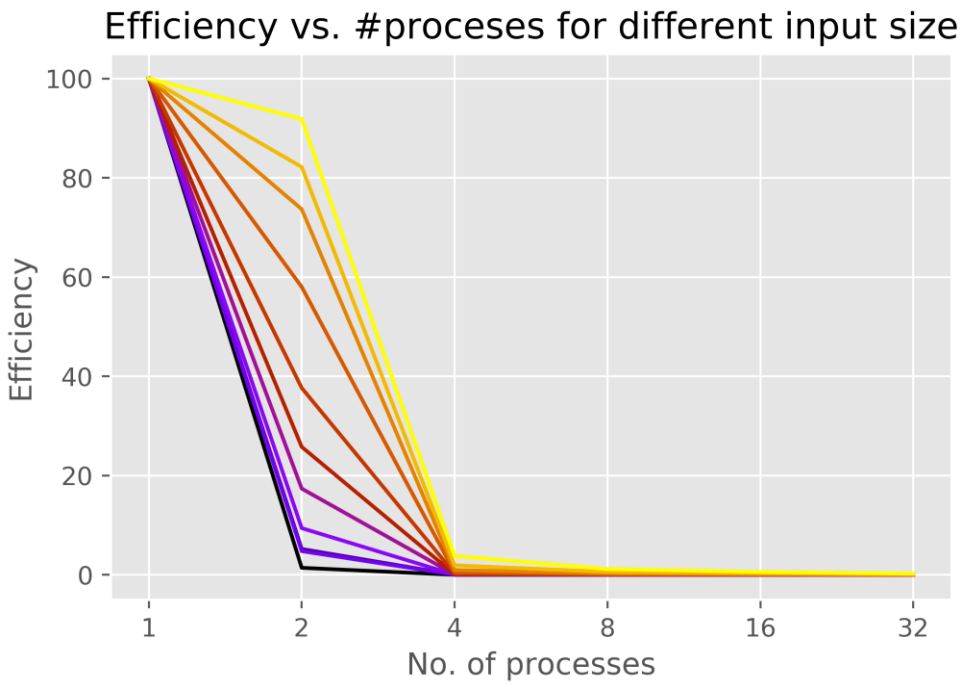


Inference:

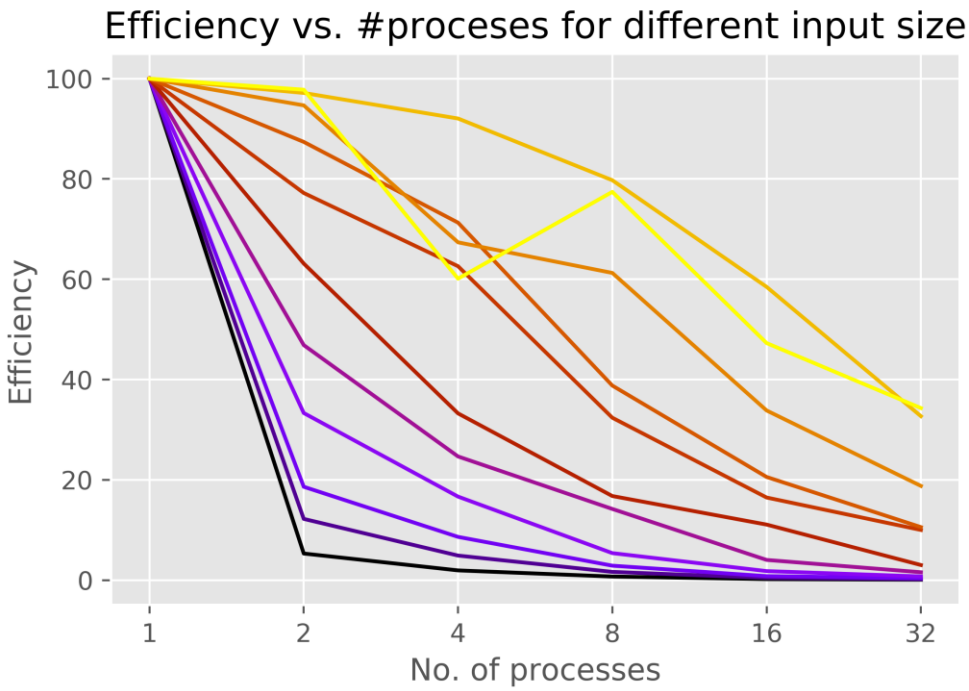
1. Based on the runtime performances, we see similar trends holding up in the Speedup charts.
2. Except for a few border cases (for some of the smallest input sizes and #processes ≤ 2), we see a net slowdown for 'NaiveAllReduce'.
3. Because of the surprising lack of communication bottleneck (#processes ≥ 16) in the 'MyAllReduce' version, it is able to achieve a maximum speedup of ~10X. Whereas, 'LibraryAllReduce' manages a maximum speedup of ~5.5X (for #process = 8) before succumbing to the communication bottleneck.

1. Efficiency (in percentage)

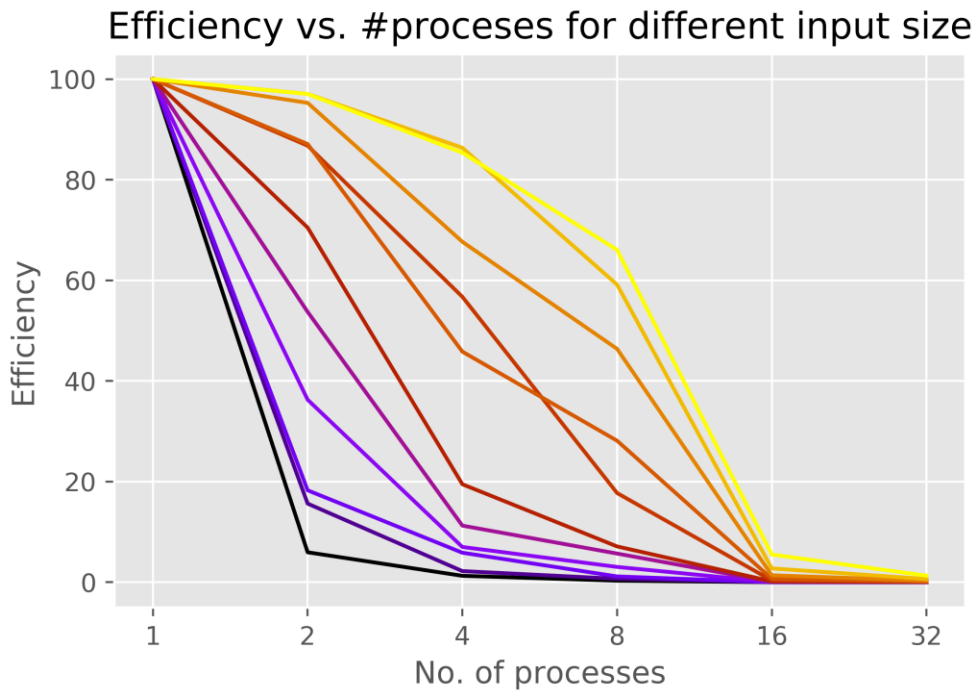
a. NaiveAllReduce



b. MyAllReduce



c. LibraryAllReduce



Interpretation:

1. The efficiency values of 'MyAllReduce' show a more gradual decline than that of 'LibraryAllReduce' (which is not at all efficient for #process ≥ 16, in keeping with the bottlenecking trend).
2. Interestingly, the efficiency values for n = 512K is better than that of n = 1M for 'MyAllReduce'. This is not so for 'LibraryAllReduce' which show the expected trend of having better efficiency for n = 1M than for n = 512K (at least for #processes ≤ 8).

Conclusion:

Given the results, we can conclude that:

1. For an application like vector addition (or any binary associative operator) increasing the number of processors does not make sense for smaller input sizes (n ≤ 8K). Parallelizing is an overkill.
2. The cost of communication soon becomes so dominating in the naïve version that runtimes for different sizes of input start converging since the computation time becomes negligible in comparison.
3. Communication remains the enemy in terms of being the bottleneck and can significantly hamper performance even after increasing the number of processors. The key is to find a way to minimize it (something that I have inadvertently managed to do in 'MyAllReduce'). What is weirder is that 'LibraryAllReduce' seems to suffer from it as well which is unexpected.