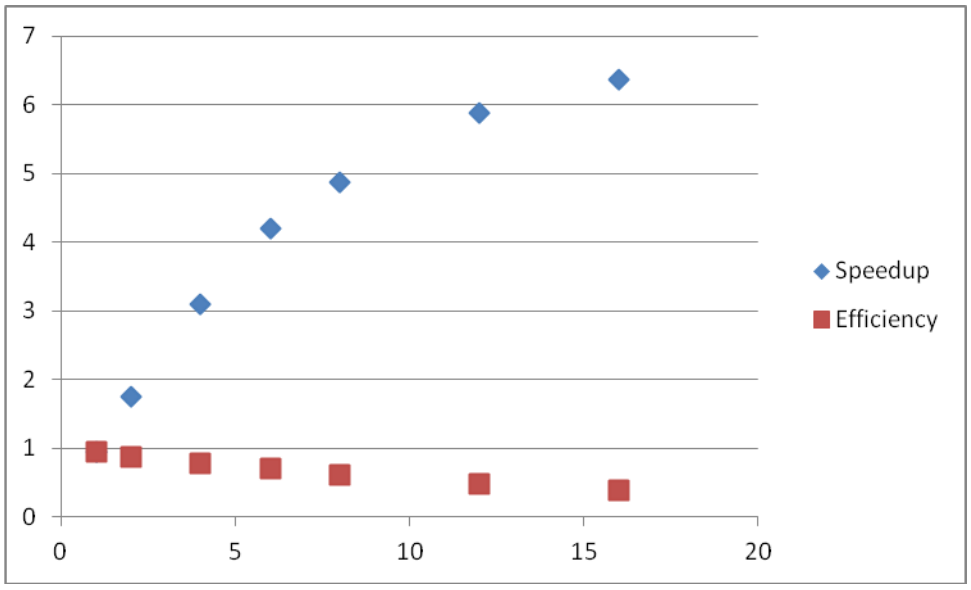
Wesley Chen

CS205 HW 2 P2

The full outputs of all of the runs with P = 1, 2, 4, 6, 8, 12, 16 and 5 before and after are included after all discussion.

The follow graph shows the plots of the speedup and efficiency as a function of the number of processes. We see that the speedup, follows Amdahl's law and begins to logarithmically cap out at just over 6. The efficiency, as expected, is reduced with each additional processor used in what seems to be a linear decrease.



With 5 processes, the "start" point for the calculation would cause some data to be lost as 5 did not fully divide the total number of tasks. This is because the way the start point as calculated was to evenly divide the numtasks into p identical chunks. For it to handle an odd number, some processors would need to do more work than others. Thus, we see a "possible failure" error with the current setup.

To fix the code, one could image just rounding up the "chunk size" per each processor and have the last processor do the remaining ones. So each processor would do one more than the current implementation and the last one would clean up the remainder. However, this decreases the efficiency of the last process so I decided to evenly divide the "extra" processes into giving only 1 extra to the 0^{th} through numtask % size processors. This would balance out the work and have only the minimum number of processors do one more task and all others do one less task. I did this through computing the start offset with adding in a conditional term to see if its rank came before or after the additional jobs should have been completed (see code). An even further improvement would be to tack on the jobs to the last processes instead of the first processes as process 0 handles other tasks so for the entire code to take less time, minimizing the work on process 0 would be ideal.

Yet another improvement would be to use parallel reads so that the first broadcast step won't be necessary – this would be a bit more complex and probably imitated a scatter implementation more, but would also help.

Data:

1

Serial Time	16.253754		
Parallel Time	17.040348	Speed Up	0.953839
Parallel Result	3144543.496	Efficiency	0.953839
Serial Result	3144543.496		
Relative Error	0.00E+00		

2

Serial Time	16.35383		
Parallel Time	9.303953	Speed Up	1.757729
Parallel Result	3144543.496	Efficiency	0.878865
Serial Result	3144543.496		
Relative Error	2.93E-13		

4

Serial Time	16.393337		
Parallel Time	5.292222	Speed Up	3.097628
Parallel Result	3144543.496	Efficiency	0.774407
Serial Result	3144543.496		
Relative Error	2.44E-13		

6

Serial Time	16.301828		
Parallel Time	3.878835	Speed Up	4.202764
Parallel Result	3144543.496	Efficiency	0.700461
Serial Result	3144543.496		
Relative Error	2.62E-13		

8

Serial Time	17.092713		
Parallel Time	3.502587	Speed Up	4.880025
Parallel Result	3144543.496	Efficiency	0.610003
Serial Result	3144543.496		
Relative Error	2.68E-13		

12

Serial Time	16.403791		
Parallel Time	2.785282	Speed Up	5.889454
Parallel Result	3144543.496	Efficiency	0.490788
Serial Result	3144543.496		
Relative Error	2.64E-13		

16

Serial Time	17.081942		
Parallel Time	2.683072	Speed Up	6.366561
Parallel Result	3144543.496	Efficiency	0.39791
Serial Result	3144543.496		
Relative Error	2.66E-13		

Original

5

Serial Time	16.549113		
Parallel Time	4.457347	Speed Up	3.712772
Parallel Result	3144542.994	Efficiency	0.742554
Serial Result	3144543.496		
Relative Error	1.60E-07		

Fixed 5

Serial Time	16.342556		
Parallel Time	4.51236	Speed Up	3.621731
Parallel Result	3144543.496	Efficiency	0.724346

Serial Result	3144543.496
Relative Error	2.57E-13