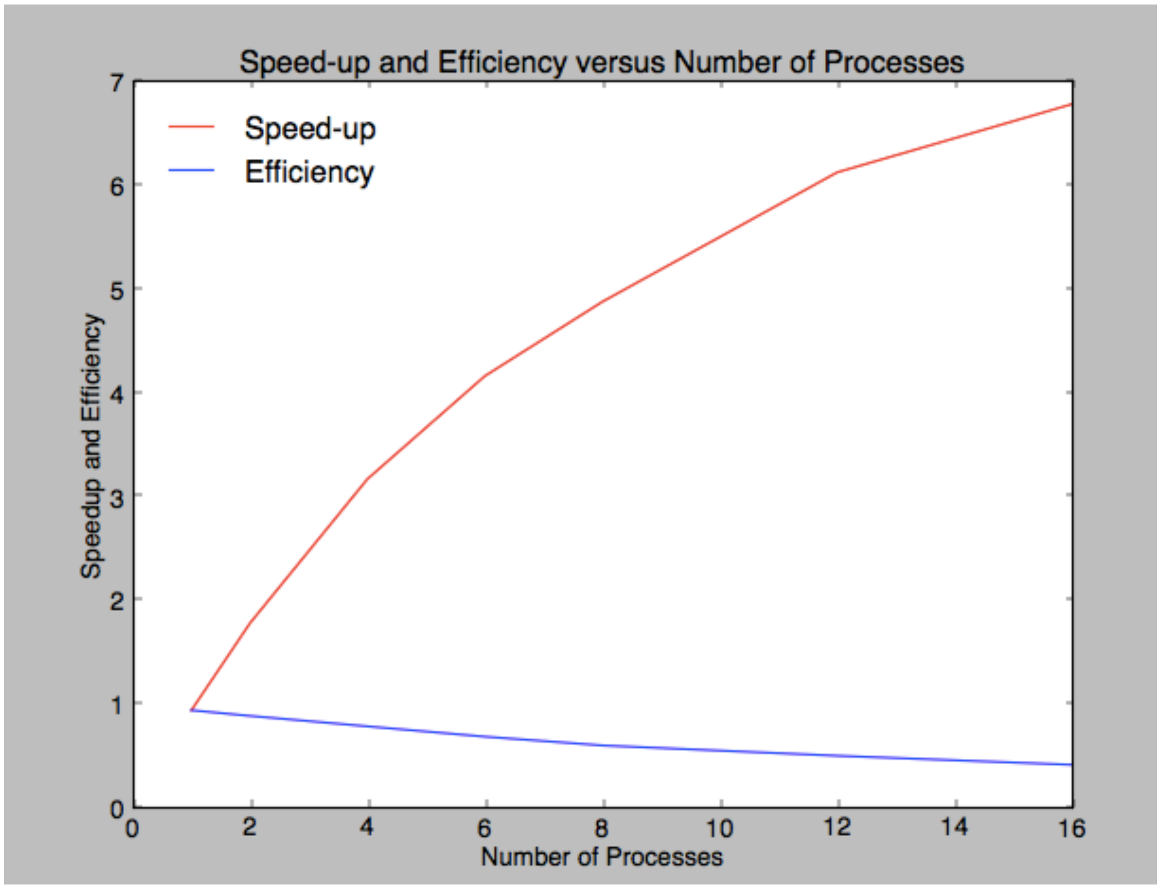
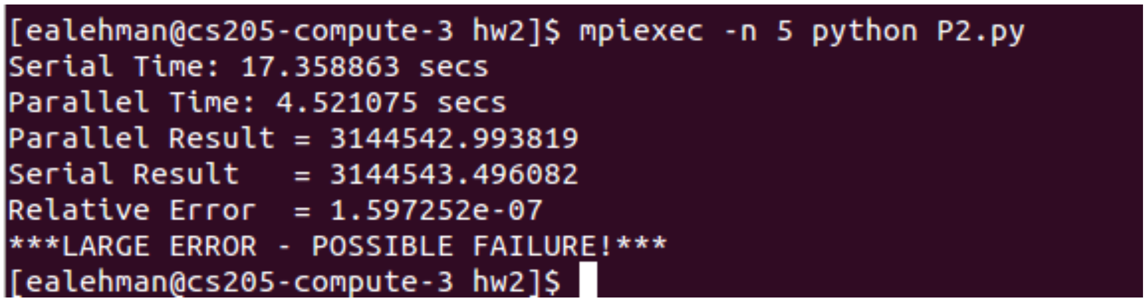
P2



The trajectory of the speed-up ($S = T_s / T_p$) reflects the result of the fact that N is constant. It starts to level out after a certain number of processes because eventually, no matter how many processes are added, you can't get a faster result. The efficiency decreases with the number of processes as expected since $E = T_s/(P*T_p)$ and with a constant N, is O(1/P).

When you try to run the code with P = 5 processes, you get the following result:



```
[ealehman@cs205-compute-3 hw2]$ mpiexec -n 5 python P2.py
Serial Time: 17.358863 secs
Parallel Time: 4.521075 secs
Parallel Result = 3144542.993819
Serial Result   = 3144543.496082
Relative Error  = 1.597252e-07
***LARGE ERROR - POSSIBLE FAILURE!***
[ealehman@cs205-compute-3 hw2]$
```

The reason that this happened was because the length of the array is not divisible by the number of processes. So when you do len(a)/size, you get a number that python automatically rounds. Then, take the sum of the different local dots starting and stopping at various places in arrays *a* and *b*, these designated start and stop places are not accurate because you are leaving out parts of the array that we're supposed to get summed. I fixed this by using np.array_split, which divides the array into groups that are distributed as evenly as possible but that include all elements of the array. I called local_dot on the various sections of this array, using rank as an index so that the section that was being analyzed depended on which process was computing the local sum.