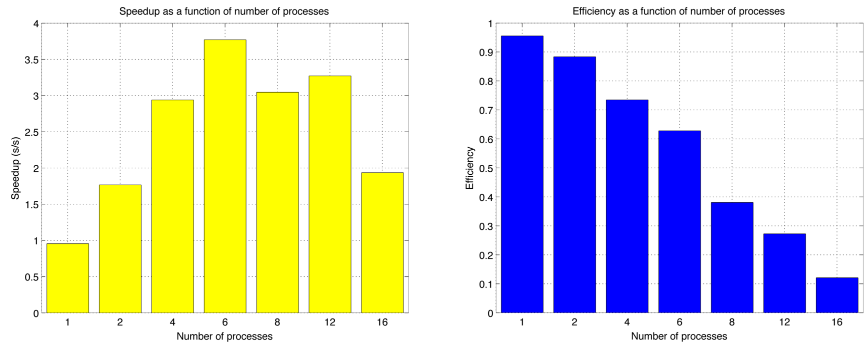
Alperen Degirmenci
CS 205 - HW 2
10/12/12

Problem 3



(a) Speedup (b) Efficiency

Figure 1: Speedup and efficiency plots for $P3$ executed on Resonance Node

As we can see in Figure 1a, the speed increases up to 6 processes, and drops after on. This is due to there being 6 physical cores in the Resonance Node. When we have more processes than cores, they are being run using hyperthreading. As we can see, this actually slows down the computation, especially when we have 16 processes. We can see the effects of hyperthreading on efficiency as well. There is a significant drop in efficiency between 6 and 8 processes. The overall decrease in efficiency can be attributed to the slowdown caused by more communication between processes.

Output when P = 5:
Serial Time: 9.063656 secs
Parallel Time: 2.690606 secs
Parallel Result = 3144542.993819
Serial Result = 3144543.496082
Relative Error = 1.597252e-07
LARGE ERROR - POSSIBLE FAILURE!

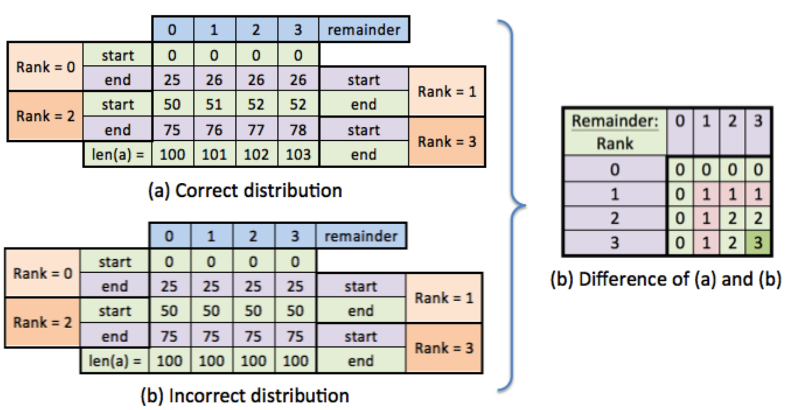


Figure 2: Test case.

We can see that the parallel result is less than the serial result. This suggests that some of the data has been omitted in the parallel calculations. A close inspection of the code confirms this theory. When determining the start and end indices for the inner product, $len(a)/size$ has a non-zero remainder when size is an odd integer (except when size = 1 or 3), which gets left out from the calculations. The most efficient way to distribute the remainder between processes is to give each process one more calculation until there are no remainders left. I looked at a simple test case with $len(a) = 100, 101, 102$,and103 elements and 4 processes. Let $n = \lfloor len(a)/size \rfloor$ and $rem = mod(len(a), size)$. Figure 3a represents this case. We can see the *begin* and *end* variables for each process for each remainder case. Figure 3b represents the (incorrect) distribution of the data when the provided *P3.py* is run. Looking at the difference between these two cases, we get the table in Figure 3c. This table can be produced by calculating $max(rank, rem)$. The result of this operation will be added to $n * rank$ to get the correct start index. Similarly $(n + 1) * rank + min(rank + 1, rem)$ gives us the correct end index. The output of the fixed program is shown below.

Output when P = 5:
Serial Time: 9.466945 secs
Parallel Time: 2.712090 secs
Parallel Result = 3144543.496083
Serial Result = 3144543.496082
Relative Error = 2.572245e-13

Improvements:

Instead of Sending the data from root to other processes, each process can read/generate its own data. This would drastically reduce the amount of communication between processes.