Comp Systems Project 1

Sean Olejar: solejar236@gmail.com

Mhammed Alhayek: almoalhayek@gmail.com

Brian Monticello: b.monticello23@gmail.com

For Part A of the project, we set up the framework for the future parts by writing modular code that made use of functions. Timing showed that this serial code is very fast for the smaller data sets, but the timings exhibit a large rate of increase moving from the 10k to 100k data set. This means the code is not scalable.

For Part B, we decided to approach the issue of processes only spawning one child by using a recursive solution. We felt that this approach elegantly allows children to keep instantiating new children as necessary. It is worth noting that while the solution is elegant, we found it conceptually difficult at first. As far as process count, we knew that we wanted to strike a balance. On one end of the spectrum, a single serial process avoids the overhead costs of parallelism, but under-utilizes system resources. On the other end of the spectrum, an completely parallelized code where every elem is handled by the process would have extreme I/O overhead. We wanted to strike a balance, where our resources where being effectively utilized, while overhead costs were minimized. After running some tests we decided to go with 4 processes. Data was split evenly between all processes. Though slower than the serial code, it was still fairly fast. Going from the 10k to the 100k, the rate of increase in the timing was less steep than in the serial code. This leads us to believe that this multi-processed solution is more scalable than the serial version.

For Part C, we decided an iterative solution was the most intuitive. Compared to Part B, the iterative solution was much simpler to implement in our opinions. We used the same process as mentioned above to select 4 total processes for our program. Data was split evenly between all processes. Timings were similar to the recursive solution. Though initially slower, the rate of increase in timing was much less steep than both the recursive solution and the serial solution. So while it's slower than the recursive solution for smaller data sets, we believe it might be more scalable than the recursive solution as the size of data increases

For Part D, we decided to combine the iterative and recursive functions used in Part B and C. Each of the 4 processes of the recursive function in part B calls the iterative function of Part C, which spawns 4 processes to calculate its statistics. This results in 16 total processes. Data was split evenly between all processes. The timings were slower than part B and C. This could be because our additional function calls introduced greater I/O overheads. Our solution might also not be completely optimized. This method was slower for smaller data sets, and seems to exhibit a slower rate of change than the serial code, making it more scalable. With these data sets, this solution seems more scalable than the recursive solution, yet slightly less scalable than the iterative solution.

Spark script:

Upon reading the project description, our first thought was that this sort of problem would be well-suited to the Map-Reduce paradigm. As a result, we decided to write a script in the Apache Spark environment that would automatically split the work over several processes, and collate the results. Looking at the timings for different files, you see that the Spark script actually performed the worst out of all the options. This is explained by several factors. One is that Spark is designed to easily abstract away the passing of information between parent and child processes. The cost of this is that the code, while significantly easier to write and understand, is not necessarily as optimized as the direct C code. Additionally, there are certain start-up and overhead costs that are implicit in instantiating the Spark environment and running a program on it. Because our problem is of relatively small scale, these overhead costs completely dominate the timings and outweigh the actual computation times (especially for the file of size 10). If the job was much larger, however, we expect that Spark might demonstrate greater scalability than our handwritten code, and these overhead costs would become relatively smaller.

A couple of observations:

* The multi-processed solutions were not necessarily faster than the serial code, but they appear to be much more scalable.
* While we were able to extract some general trends about our solution quality, it is difficult to completely characterize the solution with such small input data sets. As further exploration, we would be interested to see how the solution timings scale with very large input files.
* While handwritten multi-processed code may be very fast, it is difficult to write and debug. This is one major advantage of Apache Spark or Hadoop, for example.