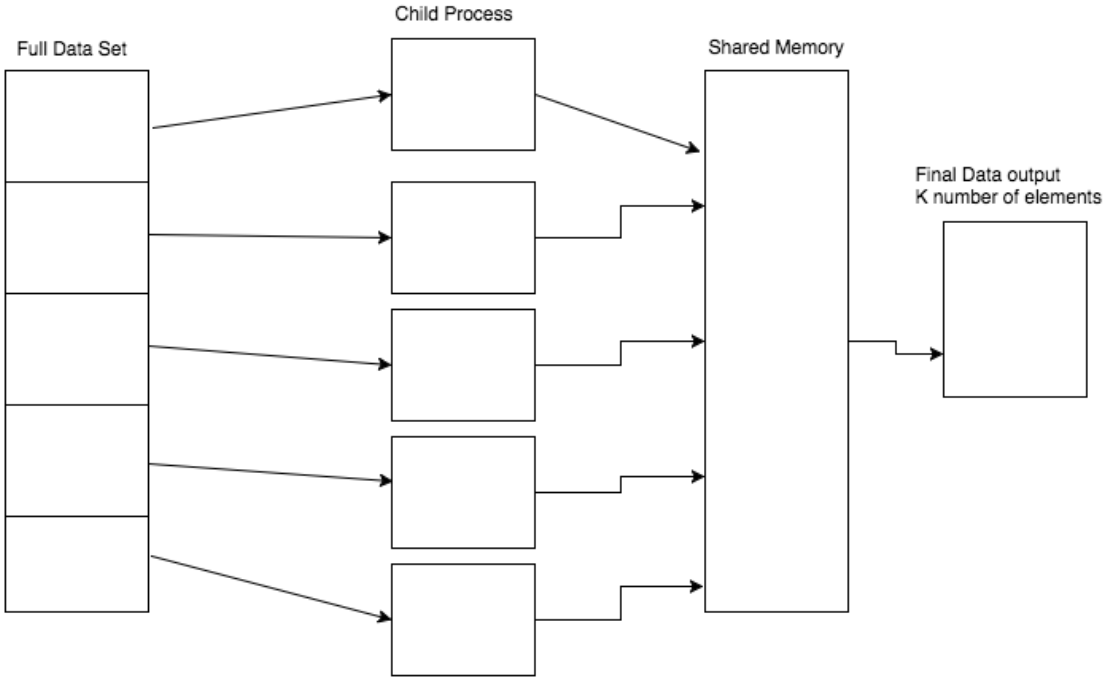
1. Initial Plan for the Algorithm



The initial plan for the algorithm was to take the chunking approach to splitting the data across multiple processes. The parent process would parse the dataset and load it into a data struct, then fork off the specified number of process and evenly chunk the data set. Then each proc would work in its own heap space running the distance algorithm and then when all processing was done load it's results into its own division of shared memory. Then once the parent proc has collected all of its children, it would read through the shared memory, re order and write the k results to a csv.

Why Chunking vs Interleaving

I decided to go with the chucking approach to simplify the shared memory accessing. I could easily figure out the size of each chunk based off how many children procs and the size of the dataset. Then I could allocate the correct size of memory knowing I would only store k elements from each proc. This also allowed me to give each proc their own section of the shared memory meaning I didn't have to mutex the writes allowing for parallel writes speeding up my algorithm's processing time.

2. First round implementations – (Final time analysis for file sizes are below)

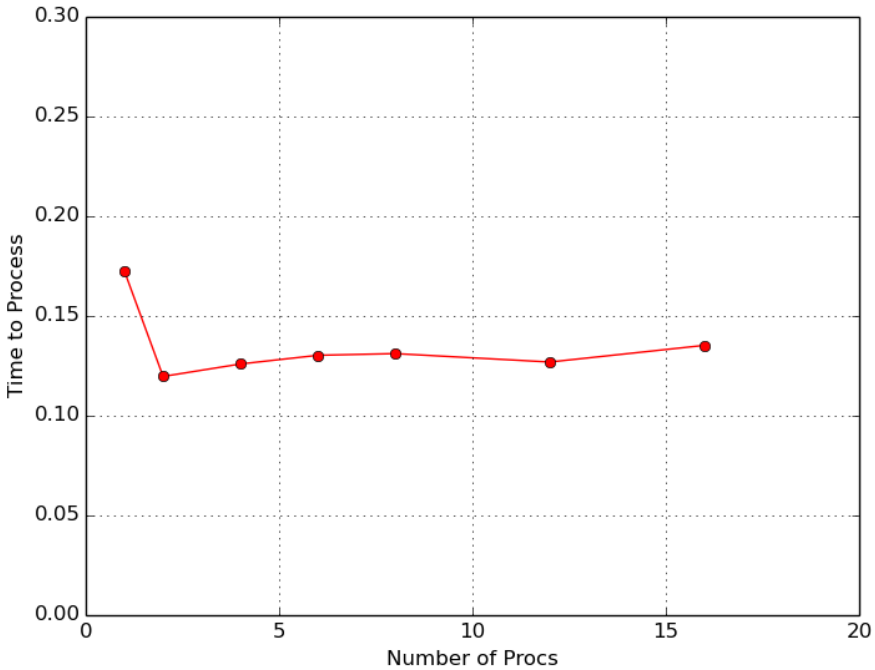
Parser: For the parser class I used getline to get each line, store the filename as the key in an ordered map then a stringstream to convert the comma separated values and store them into a vector of floats and store that as the value for the maps. This is a huge bottle neck since I'm make reallocations for every insert along with the function call to insert for every single value in the file.

Algorithm: First I used `int divNum = dataMap->size() / p;` to find the block size for each of the processes. Then setting using a for loop for the forking I used the I variable to find which

process I was in order then multiply to find the chunk to process. The problem I ran into was random access into a map. My solution was to use the std::advance to advance the iterator of the map to the correct position. Then just step through the block for each child proc. I now relize not only is that very inefficient. For finding the distance I just passed the vector from the map iterator the a find distance function. I also found using gprof that I was calling the std::vector::copy a lot. I notice that I was passing the vector of floats in each of the maps to the distance function by value instead of reference. This caused a huge space and time problem since it was copying every vector from the dataset multiple times.

Once the children procs found the distance I would store the result into a new map with floats (distance) for the key and long (line number) for the value. Because the c++ maps are implemented with a red black tree it sorted the distances during insertion. I then added the k top elements into the child's section of shared memory with a struct of distance:linenumber.

Once all process where collected, using a vector of pids and waitpid(vector.at(i),&status) to collect all of the children), used the vector constructor to just pass the beginning and end of the shared memory space to construct a vector from and sorted it based off of the distances. Then I created a map from line numbers to filenames and used that to find the distance and name of each of the top k elements and wrote them to the output csv. All of these data structure conversions was very costly.

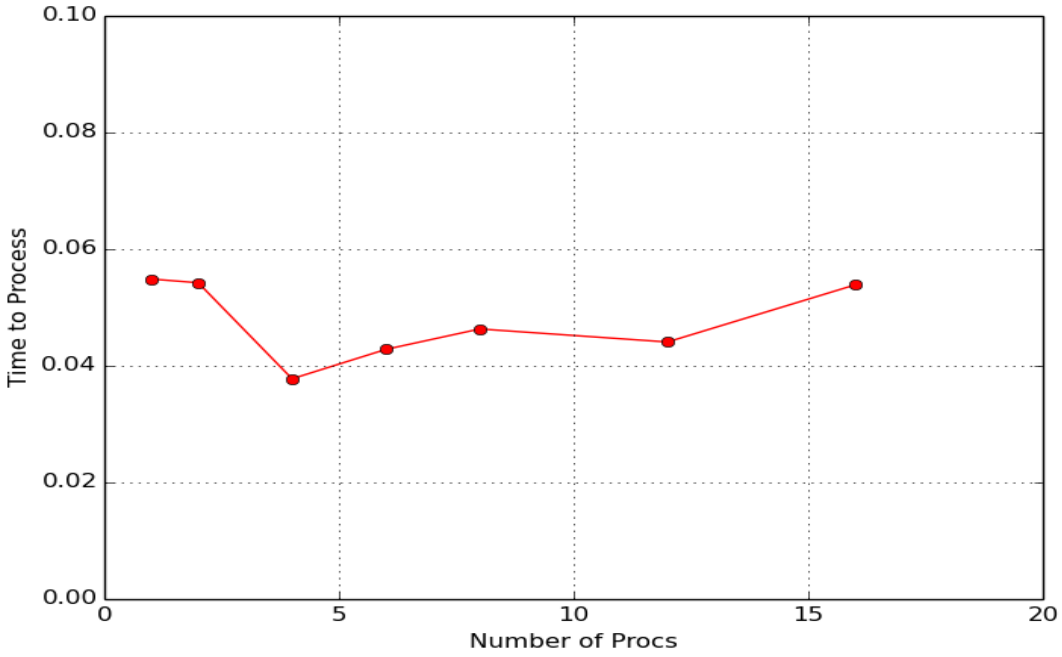


Showing the multiproc with the 6300 size file, We see the expected results. The addition of procs causes a speed up until we hit 2 procs since I have a 3 core virtual machine running, the resources and context switching of multiple procs starts to overcome the speedup of using them. This is without the opt flags since they were causing a lot of weird results for the timing.

3. Optimization 1

After using gprof I notice I was making an incredibly high amount of calls inserting and looking up the float values in the vectors. I was always using the function calls to do so which was a high overhead since I was doing so many calculations. To fix this I moved from a map of filenames and vectors of floats to a map of filenames to indexes and stored the float data into a 1D array of floats. The indexes would be the index of the first element in the line. Also during constructing this I counted how big each line was to make all the calculations in the macro for indexing. Now for finding the distance instead of passing a vector I just passed a starting index and used pointer math to access the data.

From this I saw almost a double in speed for parsing and loading the file and a huge increase in the processing time for the files.



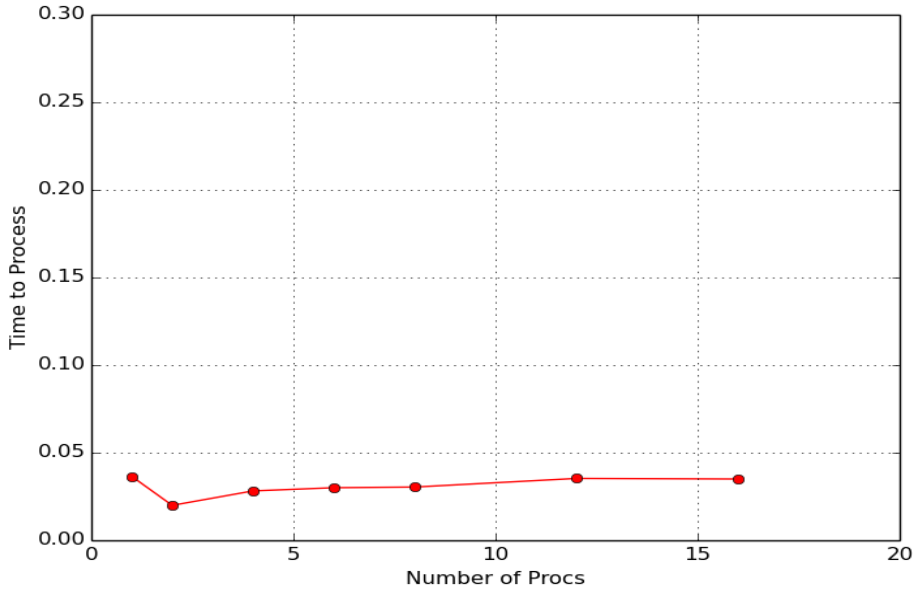
That caused a major speed up in the processing times for all of the number of procs. There is also around the same trend where there is a speed up for more procs the the uprise when the overhead of the page swaps and context switching gets to high for the benefits of the parallelizing.

4. Optimization 2

For the next optimization I looking into how the child procs were being collected. In he base algorithm I was populating a vector with all the pids then during the collection, I would use waitpid to collect the pids in order. In this way if one proc finished first they could only be collected in order of being forked. To fix this I changed from a vector to a set and from waitpid

to wait. Then just taking whatever pid was collected from the wait function and removing that from the set. I would loop until the set was empty.

This showed a small increase in speed especial for higher numbers of procs. I didn't show a large amount since most of the data is equally distributed over the procs.

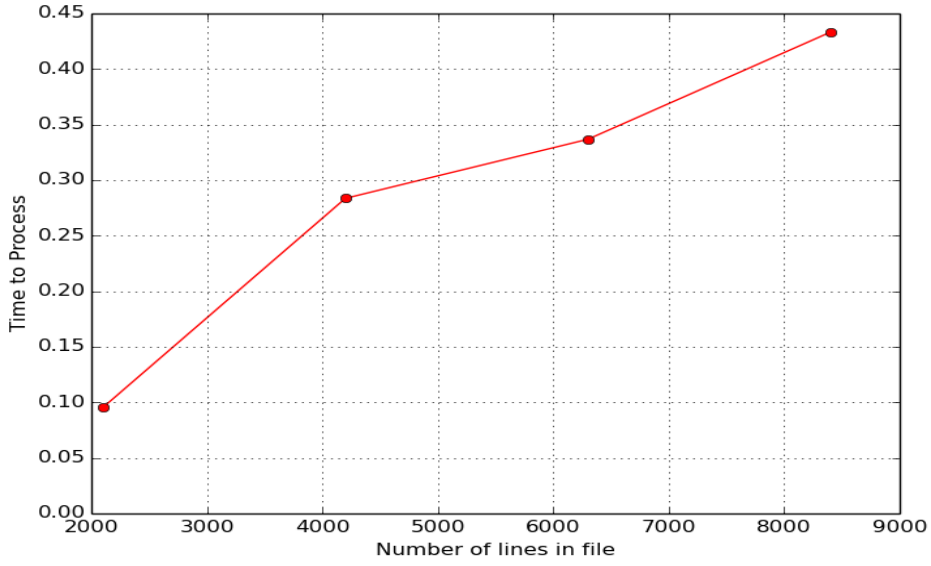


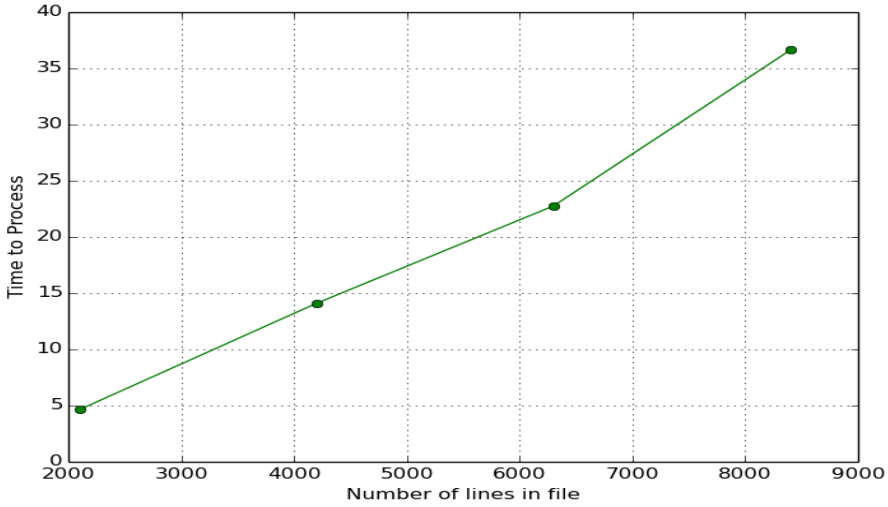
Close to same results just better times for the different procs

5. File Size Analysis

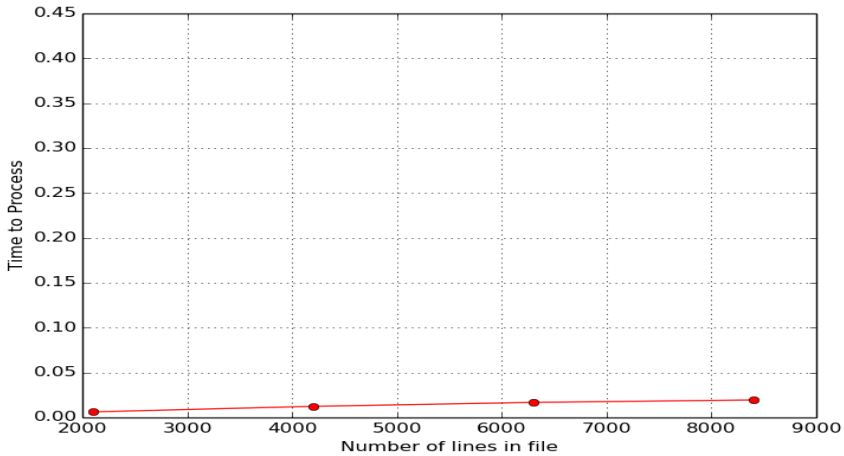
***Red lines are multiple file time do compute k dist, the greens are load times**

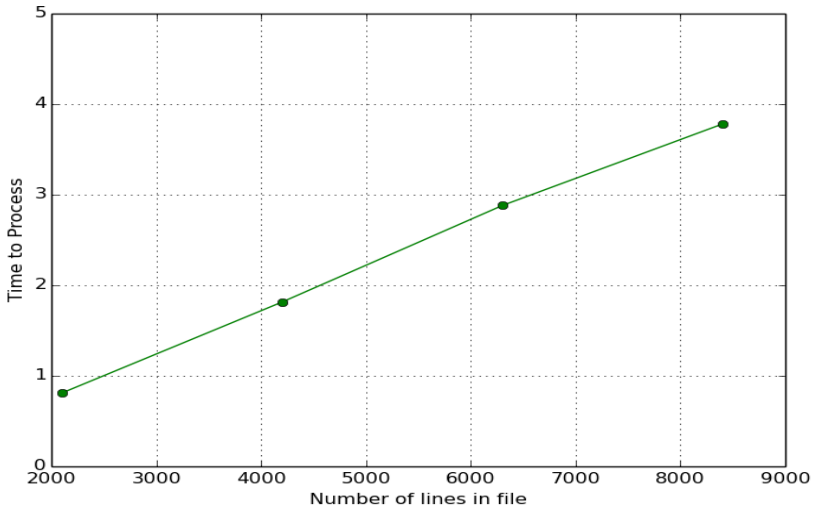
Round 1:





Optimization 1, with the O2 flag





6. Lessons Learned

The lesson that I learned the hardest on this assignment was that when a child process seg faults it quietly dies and never really says anything. It also wont print any output to the screen for debugging. After many hours of debugging, finally doing the right thing and printing out the status returned from the child proc I found that they were all seg faulting and was able to quickly fix the issue.

Anything problem I ran into was that I was not getting complete answers on my very first run, just all zeroes. The problem was that I was only using a for loop for collecting the process and did not check what status they sent. Because of this I would stop collecting and have the parent process continue without the children finishing. Since the shared memory segment is initialized to 0, I kept just getting all zeroes in my final calculations. To fix this I changed to a vector that collected the ids and checked to make sure it was the right exit status before collecting the next process.

Also, I learned that the Mac OSX process management is very sporadic. I was getting very weird results with different number of processes when running on a Mac vs running on my Linux machine. I found a lot of the same results using the O3 flag on Linux vs using the O2 flag.

The overall takeaway I got was to look closer at the data structures I choose for storing data and really understand the under lying implementation of it. Also using raw data chunks is always faster and with a few macros is almost the same to access.