Homework 2 Analysis Report

# Introduction

The purpose of this program is to find the kth-nearest vectors to a specified vector, using the L1 Norm calculation. K, or the number of matches is specified by the user, as well as the number of processes to do the processing.

## Algorithim:

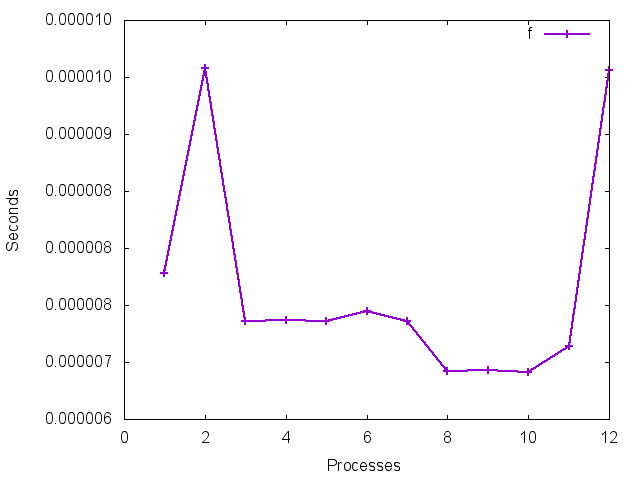
I started by reading the file into a map of filename->lineNumber, and a vector of lineNumber->vector of floats pairs.

I then sent the vector of line numbers and their floats to be processed, along with the line to be compared to.

In the processing function, shared memory is calculated and set up first. Then child processes are spawned and passed shared memory boundaries through a struct. The child iterates through its section of the database and reported its top k results to the shared memory. The results are sorted using std::sort, which is most likely not ideally efficient and performs a full sort on the data.

Once the child processes join again, the top k results are parsed out of the shared memory and reported using the same method as the child processes.

# Iteration #1



# Technique:

Not really much technique, starter code as described above in introduction section.

The times per process as shown above are:

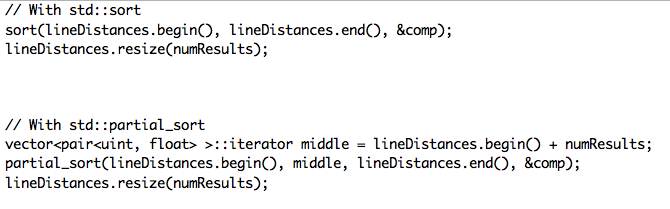
|  |  |
| --- | --- |
| 7.78E-06 | 1 |
| 9.58E-06 | 2 |
| 7.36E-06 | 3 |
| 7.37E-06 | 4 |
| 7.36E-06 | 5 |
| 7.45E-06 | 6 |
| 7.36E-06 | 7 |
| 6.92E-06 | 8 |
| 6.93E-06 | 9 |
| 6.91E-06 | 10 |
| 7.14E-06 | 11 |
| 9.56E-06 | 12 |

It looks as if 8 processes may be ideal for this algorithm as it stands now. I am working on 4 cores, 8 threads (from http://ark.intel.com/products/41316/Intel-Core-i7-860-Processor-8M-Cache-2\_80-GHz). I’m assuming the 8 threads works similar to having 8 cores.

# Iteration #2:

# Technique:

Implement std::partial sort where I used std::sort before:



Notice how the shape of this graph changes towards 12 processes. When std::sort was being used before, the number of unnecessary elements being sorted was quickly increased as the number of processes increased. By utilizing a partial sort, this number of unnecessary operations is decreased for each process. This causes the spike starting near 9 processes (in the previous iteration) to dramatically lower.

Now, it can be seen that 4 processes consistently produce the lowest time.

Times:

|  |  |
| --- | --- |
| 1.68E-05 | 1 |
| 8.65E-06 | 2 |
| 7.61E-06 | 3 |
| 6.86E-06 | 4 |
| 7.39E-06 | 5 |
| 7.77E-06 | 6 |
| 7.19E-06 | 7 |
| 6.94E-06 | 8 |
| 6.91E-06 | 9 |
| 6.91E-06 | 10 |
| 6.87E-06 | 11 |
| 7.10E-06 | 12 |

# Iteration #3