Homework 2 Analysis Report

Wednesday, February 24, 2016

Christina Mosnick

Introduction

The purpose of this program is to find the kth-nearest neighbors 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.

Algorithm:

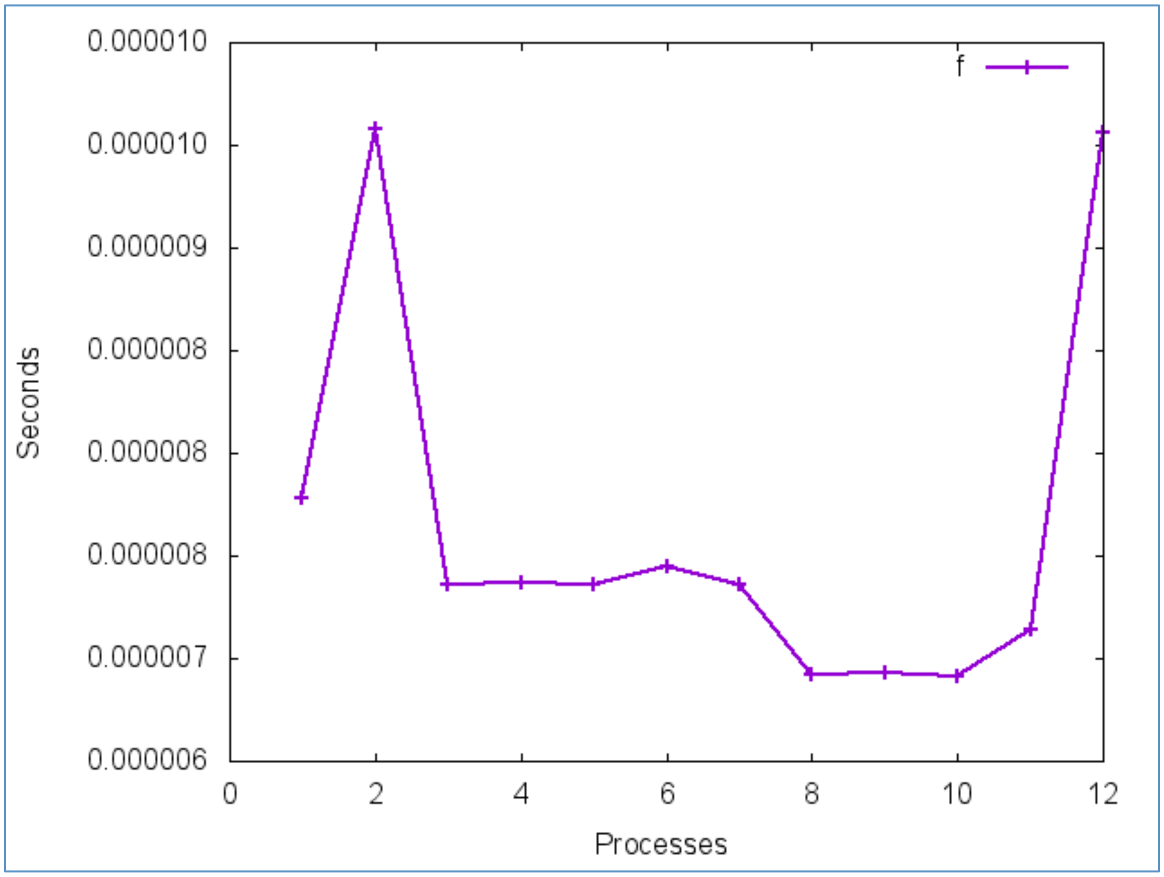
I started by reading the file into a map of filename->lineNumber, and a vector of lineNumber->vector of floats pairs.

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 reports its top k results to the shared memory. The results are sorted using std::sort, which is most likely not ideal 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:

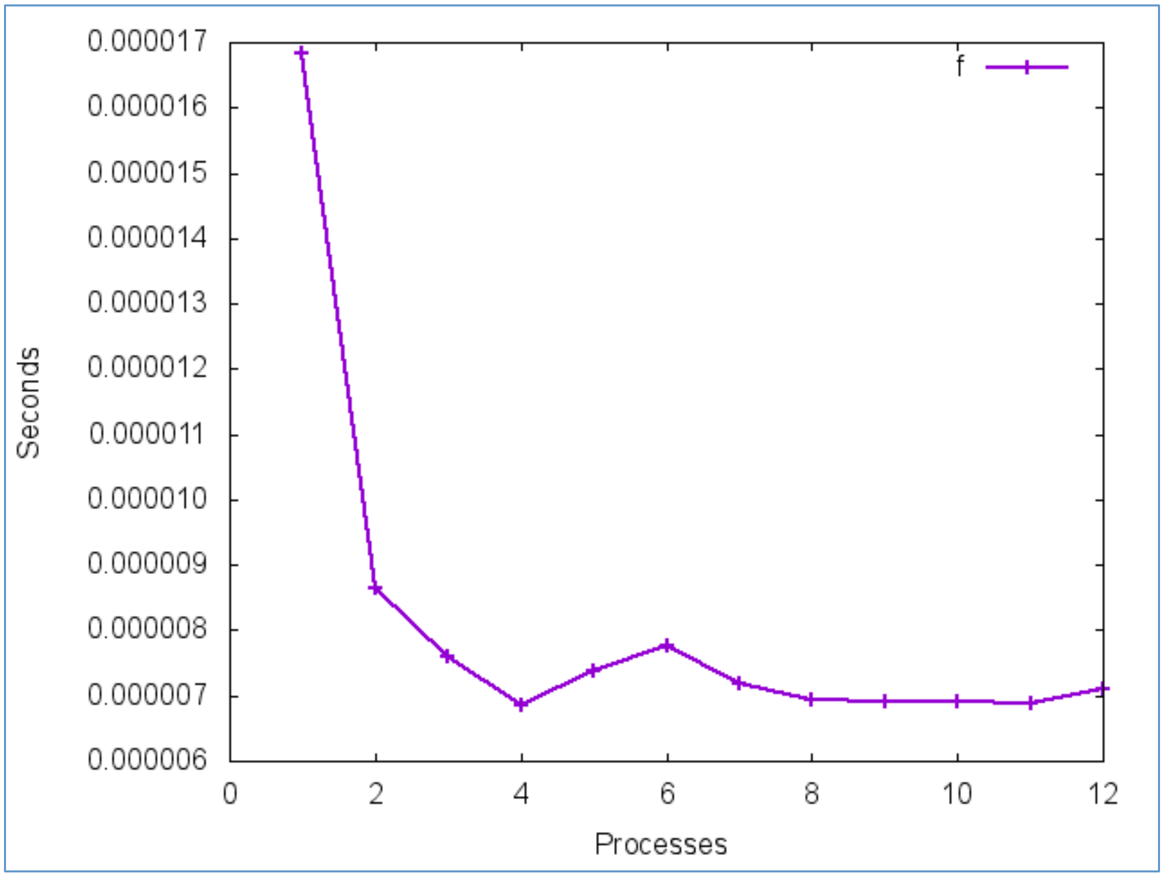
Not really much technique, starter code as described above in introduction section. The times per process as shown above are:

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.

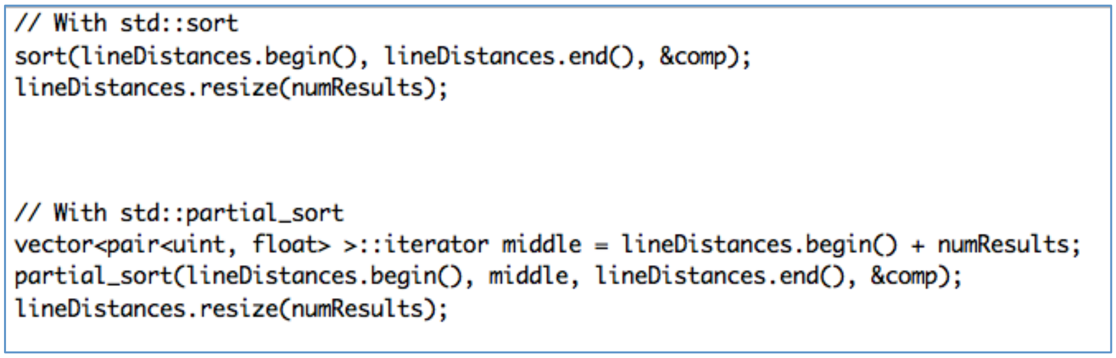
Analysis: unsure why the spike from 1 to 2 processes. Looking back on the times after multiple runs on this stage of development, the graphs are very 'spiky' everywhere. I'm unsure what is causing these inconsistencies. This could be a potential issue with the test itself. I tried increasing the number of iterations from 100 to 1000, but it yielded the same results.

Iteration #2:



Technique:

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



```
// With std::sort
sort(lineDistances.begin(), lineDistances.end(), &comp);
lineDistances.resize(numResults);



// With std::partial_sort
vector<pair<uint, float> >::iterator middle = lineDistances.begin() + numResults;
partial_sort(lineDistances.begin(), middle, lineDistances.end(), &comp);
lineDistances.resize(numResults);
```

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.

Revisiting the inconsistency issue with the 1st iteration, could the large inconsistencies have occurred as a result of the std::sort method? The trend definitely seems to smooth out for all run of the analysis when std::partial_sort is used.

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

Future Ideas / Backlog

- Use line interleaving for processing pattern, instead of partitioning data file into large chunks. This could alleviate some cache missing issues.
- Figure out better way to match filenames with line numbers at the end. Currently runs in O(n*k) time to search through map by value.
- Do more research on an optimal sorting method. Maybe std::partial_sort has some extra overhead. A min heap was mentioned in class.
- Look into difference in number of processes between iterations 1 and 2: why would ideal number go from 4 to 8 with that change?
- Todo: reduce file reading time for future. It is embarrassingly bad.
  - On a side note to that, I suggest the large file not be tested with my code. I have not tested it yet.
- Find more tools to analyze with such as cachegrind, gprof, etc.