Homework 3 Analysis Report

Sunday March 13th, 2016

Christina Mosnick

PseudoCode & Techniques:

For this homework, I chose to divide the input data using the row-interleave method.

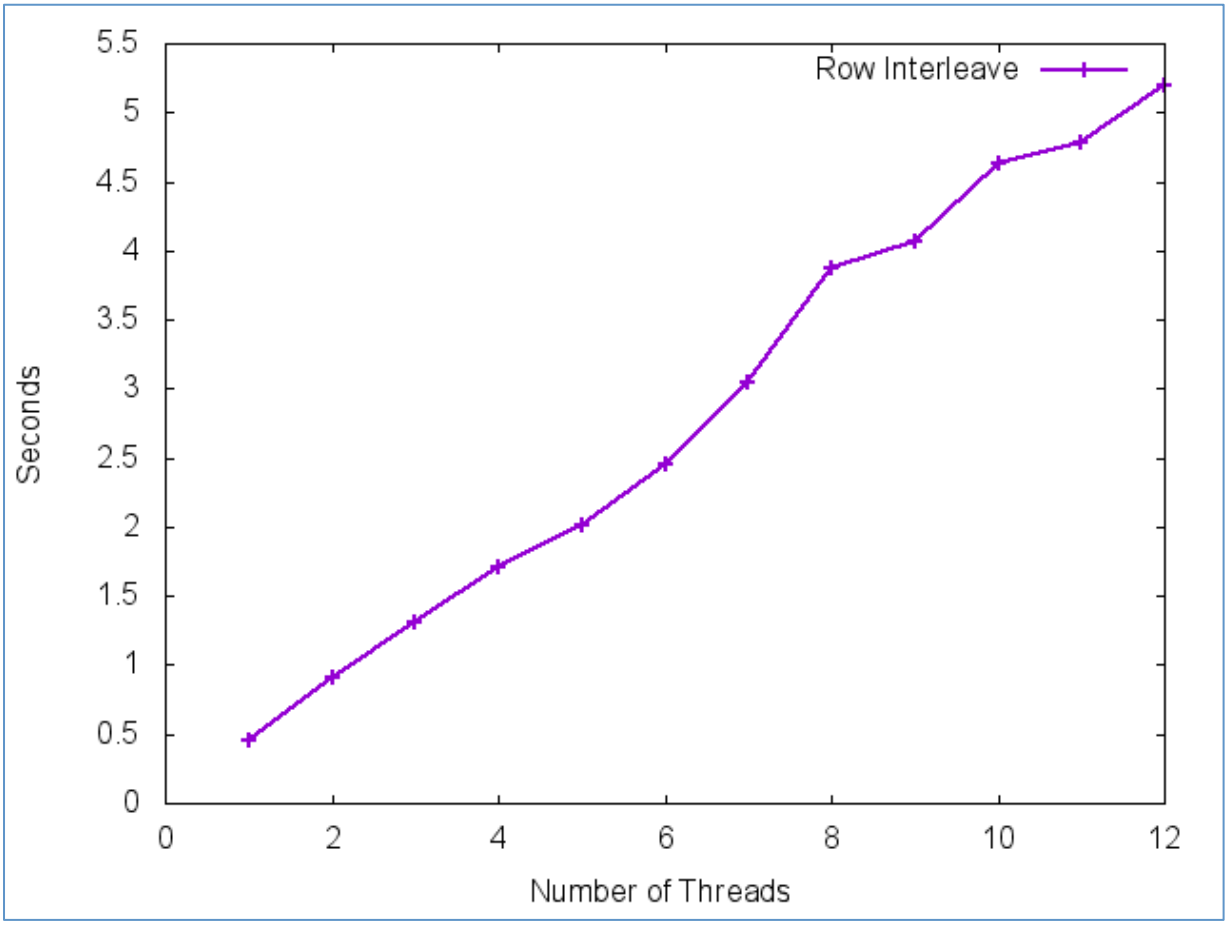
Pseudocode:

1. Read in file
    a. Read into map of filename => lineNumber
    b. As well as map of lineNumber=>float vector
2. Process data:
    a. Get number of threads, number of results desired, and the filename to query against in the data
    b. For the number of threads specified:
        i. Instantiate a thread object
        ii. Create a thread, pass it the object's row interleave function
        iii. In the row interleave function:
            1. For each line:
                a. Process line number, store in a vector of maps
                b. Skip to next line, which is (current + num threads)
            2. Partial sort vector of values using std::partial sort
            3. Cut off at the numResults index in vector
            4. Return from thread
    c. Gather all threads
    d. Gather all threads results
        i. Copy results form each thread object into new result
        ii. std::partial sort aggregated result
        iii. cut off at numResults index
    e. Match results which is in lineNumber => distance format to filenames
        i. For each lineNumber=>distance result
            1. Iterate through map filename=>linenumber until match
            2. Print result

Test 1: Varying Number of Threads

Goal: keep the number of results and the file size the same, but vary the number of threads. I used up to 12 threads. I tested each thread size for 100 iterations.

Results: I was unable to reduce processing time by utilizing mutli-threading. As the thread count increases, so does processing time, as is shown in the following graph. This test utilized the file with 8400 entries, generating 10 results:



The times for this test were:

Time(s)	# Threads
0.415135	1
0.781062	2
1.15937	3
1.45684	4
1.84899	5
2.06575	6
2.44778	7
2.81878	8
3.1207	9
3.68822	10
4.12924	11
4.71855	12

Comments:

It seems there is significant overhead in my algorithm each time a thread is used and managed. This overhead is clearly much larger than the benefit concurrent processing provides.

Part of the slowdown could be the partial sort at the end of the interleave function. This partial sorting of the result vector is certainly an improvement over a standard sort, which would be a minimum of O(n lg(n)) in complexity. A partial sort performs O(n lg(numResults)) where numResults is the distance between the first index and middle index (the cutoff point). So as the number of results increases, the partial sort complexity approaches the regular sort's complexity. If a constant min heap were executed, each result would either be added to the heap, or thrown out altogether. That decision would be a simple O(1)

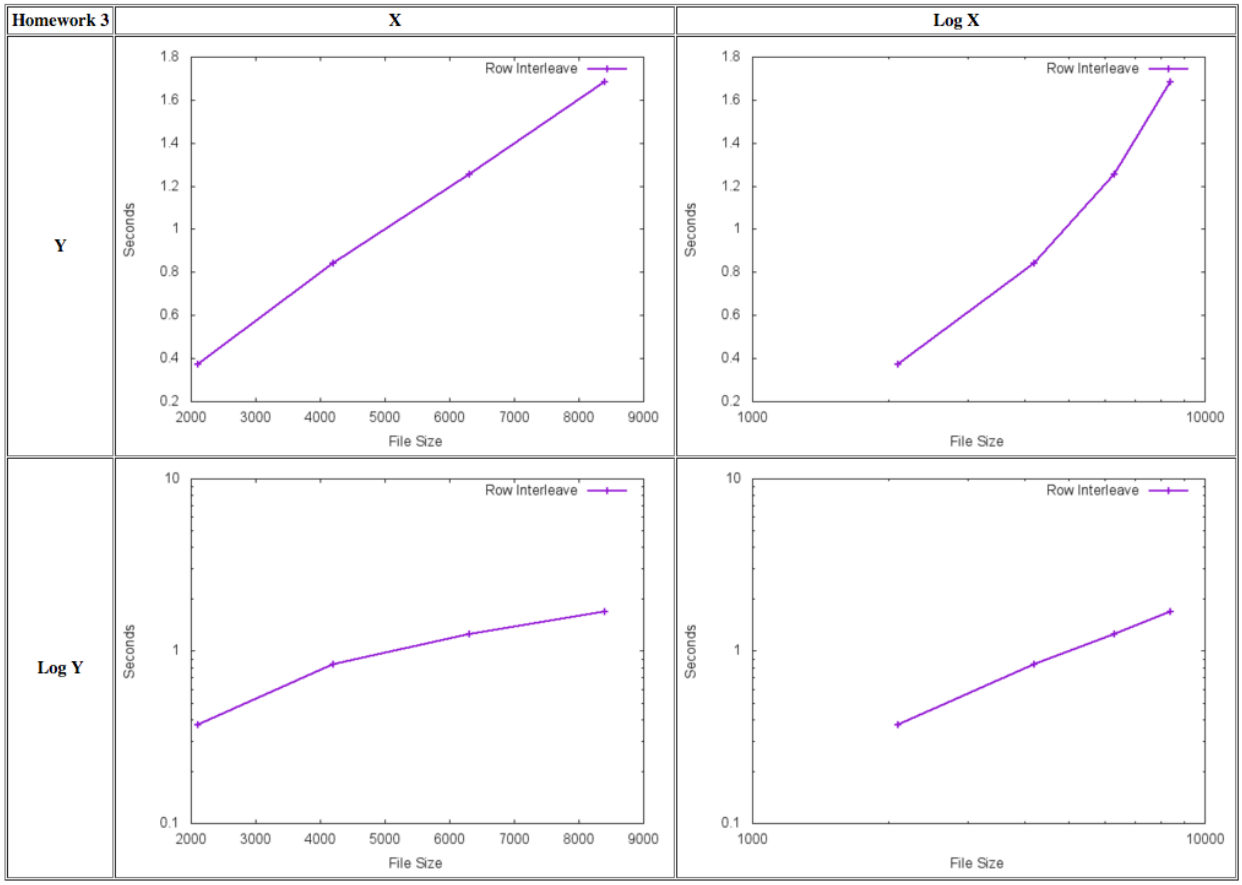
lookup, comparing the result to the top value on the heap. If a result is added, the min heap automatically sorts itself, leaving a pre-sorted heap at the end of processing all lines in the thread. Each insertion of a result would be a worst case O(lg(numResults)). Usually, the heap size (the number of results) is most likely minimal compared to the size of the data set the thread is working on. Worst case, each result in the data set is inserted in the heap only once (n insertions total), making the worst case for a min-heap application in this problem O(n lg(numResults)), which is better than standard sort in all cases except when n = numResults. This is similar to the partial sort's complexity.

Since a min-heap and partial sort have theoretically the same complexity, I would choose a min-heap since it takes less memory overhead. It rids of the need to store unneeded values.

Test 2: Varying the File Size

Goal: Test functionality with a constant number of results and threads, while increasing file and data size.

Results: The trend is mostly linear with increasing file size. This result was produced using 4 threads and 10 results:



The times for this test were:

Time (s)	File Size
0.371066	2100
0.840351	4200
1.25292	6300
1.68232	8400

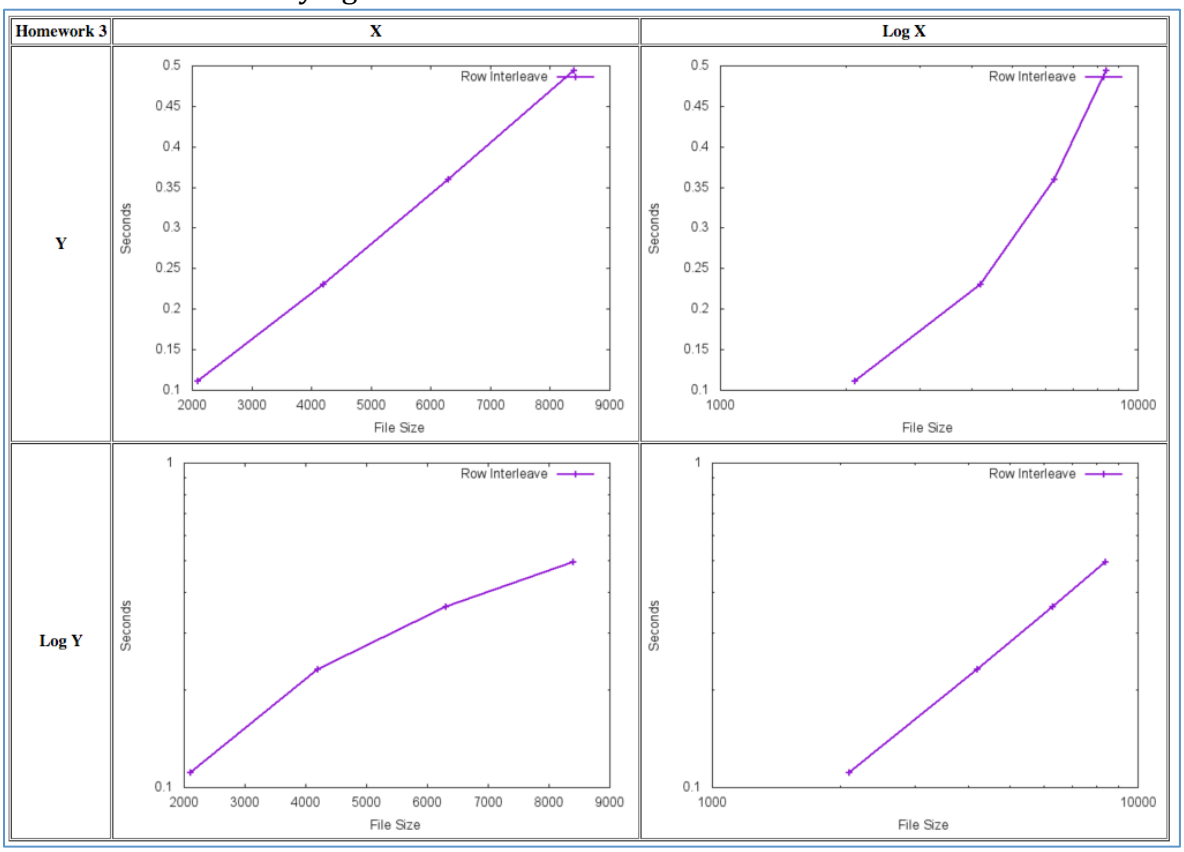
Comments:
Note that these times do not include file reading, only data processing. The charts for 1 and 8 threads processing varying files look about the same as this does.

Test 3: Vary Number of Results

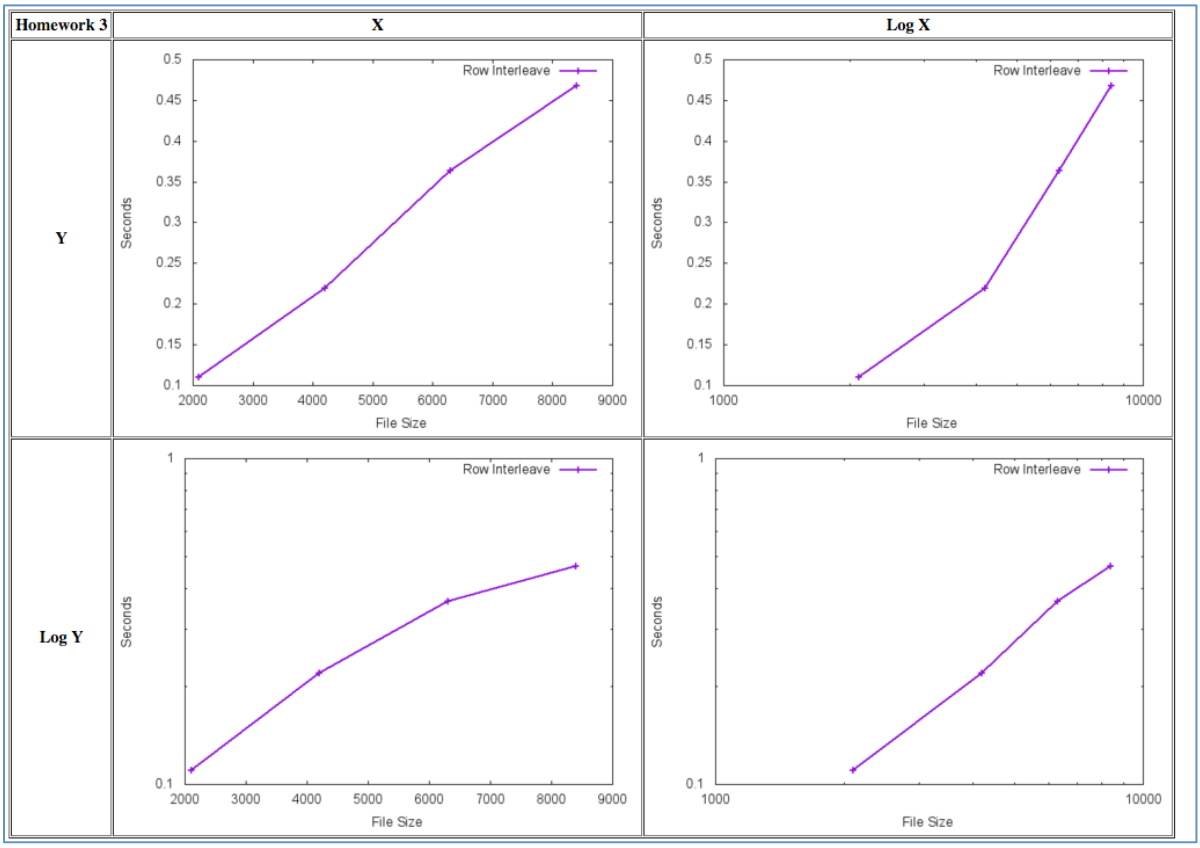
Goal: Keep a constant number of threads, while varying the number of results to be produced and the file sizes.

Results:

1000 Results with varying file sizes:



100 Results with vary file sizes:



Comments:
This visualization of 100 results does see less of a linear growth in some places. I imagine there is some sort of a "sweet spot" between the amount of data to process and the number of results to produce.

Other Conclusions:

The process of matching line numbers to filenames at the end is extremely inefficient. That operation is worst-case O(numResults * n). Worst case, numResults = n, so the operation would be $O(n^2)$. It could be good to turn the filename=>lineNumber map into a search tree by lineNumber value. This would decrease the complexity to O(lg(n)) for each lookup, making the whole operation O(numResults lg(n)), worst case O(n lg(n)) when n = numResults.

Backlog:

- Compare Row interleave in each graph to block-partitioning performance.
- Look at more comparisons for increasing number of results requested.
- Make sure threads are actually running concurrently, not sequentially. My results seem to reflect that they are not running in parallel.