Jacob Smith
CS 361 Assignment 1 Report

Implementation

This program was implemented with the idea that it should be unnecessary to read the entire file into memory so that it can scale for larger and larger datasets. The input data is in the form of XX{space} for temperatures 0 to 99 degrees celsius. (Or if you prefer balanced around zero so that all livable temperatures can be taken into account). A standard apache program is used to parse all the command line args, so this is trivial. The bin class merely holds n many bins and can easily be incremented by 1 or added to another bin.

The main thrust of the program is to read as little in as possible in order to not make the heap unnecessarily big. It is fairly easy to find out exactly how much we need to read in for each iteration as it will be NUMBINS * LEN_ELEMENT. My implementation, my elements were three long and usually we used 7 bins for each day of the week. Granted, this can change, but for this example we can easily we that each set of data is 21 characters long. Therefore, instead of reading the files in, we can just read 21 characters until we are done with the file. The DATASIZE parameter does allow for using bigger files to look at smaller sets, as it can be calculated how many sets we need to read before hitting the end.

Since we know exactly how long each set is and how many sets we have, we can then in the threaded programs apply this to have the java program skip ahead past the sets that will be read by other threads. There is some fairly complicated algebra to partition the sets in the comments of P2, however in the end an offset is calculated (then adjusted) so that the correct number of characters are skipped. Each thread skips an equal amount and then the last takes care of what is leftover, careful to not exceed the max number of sets for the data size.

Problems / Failures

The instructions were to increase the maximum heap size in order to read the entire file in, however; I wanted my program to run on commodity hardware as well. The total memory size of my program even with 8 threads doesn't seem to exceed 4 megabytes. I believe that this is the correct way to code as throwing increasing large amounts of RAM won't solve a problem if it can be solved a much simpler, resource friendly way. In the sequential program, I was able to parse out file IO, from the overall system time (yes, there is overhead, but it is much reduced), however I have since learned that you can't have multiple threads accessing the same file simultaneously even if they each have their own file handle. This seems rather silly as I was not attempting to write the file and I frequently got locked out of the file. Sidenote: I was told this by my peers, but I am not actually sure if locking is the issue, the operations I do with my data aren't the most trivial, as I split them into bins and then compare seven strings with each other (I don't even convert to ints as it yield the same result, and I believed it to be slower). Either way, the next paragraph elaborates on my challenge of computing the IO time and parsing in a threaded program.

Another difficulty I had with the threaded problem is that it is tough to calculate the total time of file IO access in a parallel program. Since there is plenty of locking in the program for the file handles, the program has many threads stuck attempting to access the files. This

makes the program take much longer, and trying to subtract out those accesses is not calculable easy enough. One of those problems is that these are happening simultaneously so taking the total time for IO in each thread and adding them up, subtracting it from the total program time will actually yield negative program times in some cases (large numbers of threads make it appear).

Analysis

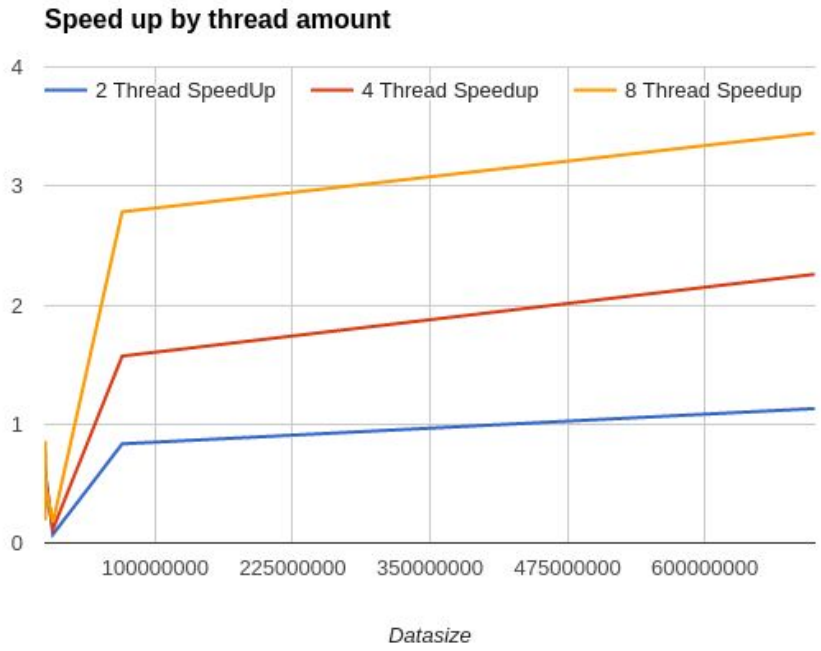
The winner of the timing went predictably to the eight threaded program, even with the challenges of either the locks or the parsing and comparing of strings. I am tempted to give it to the string parsing side of the argument because with 20 threads this program runs in 4000 MS (700M datasize), about half the time of 8 threads. If locking was an issue, the performance would only get worse. Without further ado here is the timing results:

Datasize and Thread Time in Milliseconds				
Datasize	1 Thread	2 Threads	4 Threads	8 Threads
7	2.23	5.153	6.007	11.704
70	2.546	8.591	7.809	12.743
700	5.351	19.188	16.743	14.074
7000	13.745	20.734	23.634	18.969
70000	41.838	68.194	72.0106	48.733
700000	74.316	135.073	150.497	161.247
7000000	33.0923	459.0887	284.631	192.63
70000000	2744.59	3295.351	1747.7	986.21
700000000	32638.792	28900.08	14453.66	9474.4

There doesn't seem to be much improvement of 2 threads over 1, however there is much more noticeable improvement from 2 to 4. 8 also has a slight improvement over 4. The next area to look at would be speedup for additional threads:

Speedup across by number of threads			
Datasize	2 Thread SpeedUp	4 Thread Speedup	8 Thread Speedup
7	0.4327576169	0.3712335608	0.1905331511
70	0.2963566523	0.3260340633	0.1997959664
700	0.2788722118	0.3195962492	0.3802046327
7000	0.6629208064	0.5815773885	0.7246033001
70000	0.6135143854	0.5809977975	0.8585147641
700000	0.550191378	0.4938038632	0.4608829932
7000000	0.07208258448	0.1162638644	0.1717920365
70000000	0.8328672727	1.570401099	2.782967117
700000000	1.129366839	2.258167966	3.444945537

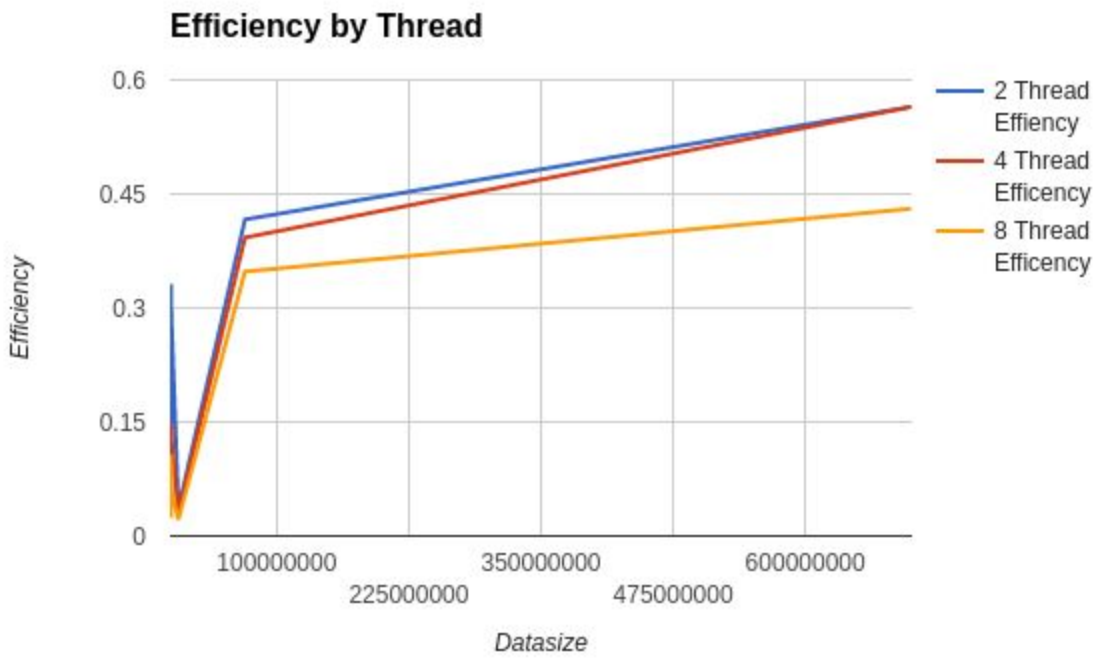
The formula for this is the sequential time / thread time. A graph would be a better representation as it is tough to read so many numbers:



Once again there is no surprise that the eight threaded program has a large margin over the other threads as it is given many more resources than the other threads, however, it is perhaps viewed better through the lens of efficiency which is the speedup amount divided by the number of threads.

Datasize	2 Thread Effiency	4 Thread Efficency	8 Thread Efficency
7	0.2163788085	0.09280839021	0.02381664388
70	0.1481783262	0.08150851582	0.0249744958
700	0.1394361059	0.07989906229	0.04752557908
7000	0.3314604032	0.1453943471	0.09057541252
70000	0.3067571927	0.1452494494	0.1073143455
700000	0.275095689	0.1234509658	0.05761037415
7000000	0.03604129224	0.02906596611	0.02147400457
70000000	0.4164336364	0.3926002746	0.3478708896
700000000	0.5646834196	0.5645419914	0.4306181922

Once again these efficiencies are better viewed on a graph so that they can better compared to one another.



Although the eight threads are the quickest to finish, they have the most overhead. The two and four threaded programs are about the same efficiency-wise as they have half the thread overhead that the eight threaded one has. Overall, I believe it is likely to only get more efficient as more threads are added, however do to all the IO that wasn't apart of the sequential program, it will never be all that efficient.

Conclusion

I suppose that this ended up being a learning experience for me in that I should perhaps abstract the IO even further away from my program. I believed that I would see a great deal of performance improvement from reading the file gradually, and in terms of memory usage, I succeeded. However, this class is about time and not memory usage so perhaps I should have redesigned my program to be as timely as possible. For my approach, I think I reached the limit of what I could do in terms of speed. I think that I could have gotten some more performance if I could have opened the files read-only so that they did not lock (or did they). Obviously keeping everything in RAM would speed it up, but I don't view that as a realistic approach.