Daniel Angell

Implementations

I started my work on this assignment by developing the multi-threaded implementation of the Histogram code first. That way it would be trivial to translate the code into serialized code later on, rather than refactoring everything to be multi-threaded for part 2. Both implementations are unified by the Histogram interface, which requires that they both implement a `generateHistogram` method.

Parallelized Version

The parallelized code resides in both the `ParallelHistogram` class as well as the `HistogramThread` class. The `ParallelHistogram` class implements the Histogram interface and acts as the “main” thread which then delegates work to many `HistogramThreads`. This is done by chunking up the original data set such that each thread gets a roughly equal portion of the original data to work on. Once each thread has aggregated its data, it attempts to take control over a shared lock (which originates as an instance attribute of the `ParallelHistogram` object) before adding its data to the shared histogram.

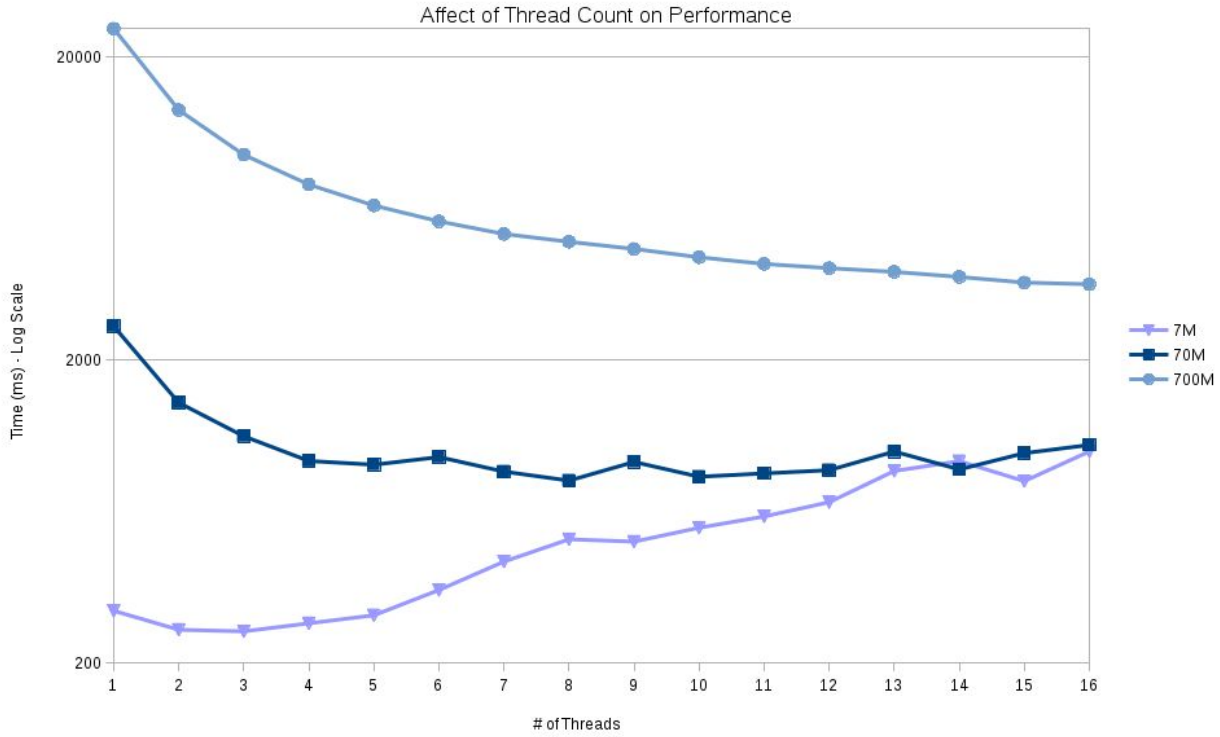
The `ParallelHistogram` class waits for all child threads to complete before returning the completed histogram.

Serial Version

The serial code resides solely in the `SerialHistogram` class. It takes a couple of methods from the `HistogramThread` class, and a couple from the `ParallelHistogram` class to provide the full functionality, but without parallelization.

Benchmark

The parallel histogram implementation was benchmarked on Tux with the script saved at docs/benchmark.rb. The results of the benchmark can be seen below, and the raw data is available at the end of this report.



Analysis

For all of the larger data set sizes, parallelism brought at least some performance benefit. With a data size of 7 million, there was a small reduction in runtime with 2 or 3 threads compared to the serial implementation. However, any more threads would quickly become less efficient than serial code. This is probably due to the overhead associated with multi-threading on the JVM. Each individual thread could be paused at any time, and may remain in the “runnable” state for many milliseconds before re-starting. In addition, any overhead that is shared by both the serial and parallel implementations would be most noticeable at smaller data set sizes.

Running the parallel implementation on a data set of 70 million integers yielded more interesting results. Moving to only 2 threads the program shows nearly half the runtime. The performance gains continue all the way to 8 threads. After 8 threads there appears to be a performance floor around 850ms. The 70 million integer benchmark shows a slight runtime increase towards the higher thread counts. This is similar to the more consistent upwards slope visible in the 7 million integer benchmark.

The 700 million integer benchmark shows the most encouraging results. It appears to decrease in runtime past the 8 thread floor seen with the 70 million benchmark. Testing performed outside of the automated benchmarking showed a decrease in runtime at even 32 threads.

Further testing performed outside of tux showed similar performance curves, but with much lower overage runtimes. For the 7 million integer benchmark it was more common to see a runtime around 50ms, rather than the 200+ms seen on Tux. This is likely due to the difference in per-core performance between AMD's CPUs and the Intel CPUs the software was benchmarked on.

Benchmark Data (Tux)

Data Size	Thread Count	Time (ms)
7M	6	349
7M	7	433
7M	8	513
7M	9	504
7M	10	560
7M	11	610
7M	12	680
7M	13	862
7M	14	929
7M	15	798
7M	16	1000
70M	1	2587
70M	2	1447
70M	3	1122
70M	4	930
70M	5	904
70M	6	958
70M	7	858
70M	8	801
70M	9	923
70M	10	825
70M	11	846

70M	12	867
70M	13	999
70M	14	872
70M	15	987
70M	16	1050
700M	1	24808
700M	2	13369
700M	3	9517
700M	4	7591
700M	5	6475
700M	6	5740
700M	7	5215
700M	8	4917
700M	9	4657
700M	10	4371
700M	11	4154
700M	12	4024
700M	13	3912
700M	14	3765
700M	15	3607
700M	16	3560