Due Friday, November 18, 2013

Thread Creation and Destruction

The object of this program is to examine the time it takes to create and destroy an individual thread based on how many other threads are being created. There timer starts right before the first thread is created, and ends after the call to `pthread_join` returns. Each of the threads does no work and returns immediately, so we can reasonably assume that each one takes no time to execute. Each timing was done 1,000 for every given number of threads, so the times shown are pretty good averages. The results of the timings versus thread size are shown below.

Number of Threads	Total Time(s)	Average Time(s)
1	0.000563336	0.000563336
2	0.000636100	0.000318050
3	0.000736959	0.000245653
4	0.000819026	0.000204757
5	0.000800666	0.000160133
6	0.000886653	0.000147776
7	0.000940261	0.000134323
8	0.000936455	0.000117057
9	0.000943541	0.000104838
10	0.000969264	0.000096926
11	0.001045427	0.000095039
12	0.001070345	0.000089195
13	0.001128440	0.000086803
14	0.001048252	0.000074875
15	0.001119614	0.000074641
16	0.001221000	0.000076313
17	0.001218128	0.000071655
18	0.001260898	0.000070050
19	0.001311076	0.000069004
20	0.001612391	0.000080620
25	0.001597676	0.000063907
30	0.001741411	0.000058047
35	0.001947602	0.000055646
40	0.002097166	0.000052429
45	0.002143373	0.000047631
50	0.002897715	0.000057954

Table 1: The results of the total and average creation time of 1 to 50 threads

While this data is great, there's a lot of it, and it isn't all that illuminating at first glance. Basically, the cost of starting and stopping a thread is high enough so that for a few threads, it is easily seen. However, as the number of threads increases, the creation/destruction time begins to converge to about 0.00005 s, or 50 μs. There are anomalies in the data; the trend line isn't a completely smooth curve. However, there is enough data to show that over time, the values do approach a constant.

The plot below shows how even though there is a high cost for small numbers of threads but the average

cost converges suprisingly quickly.

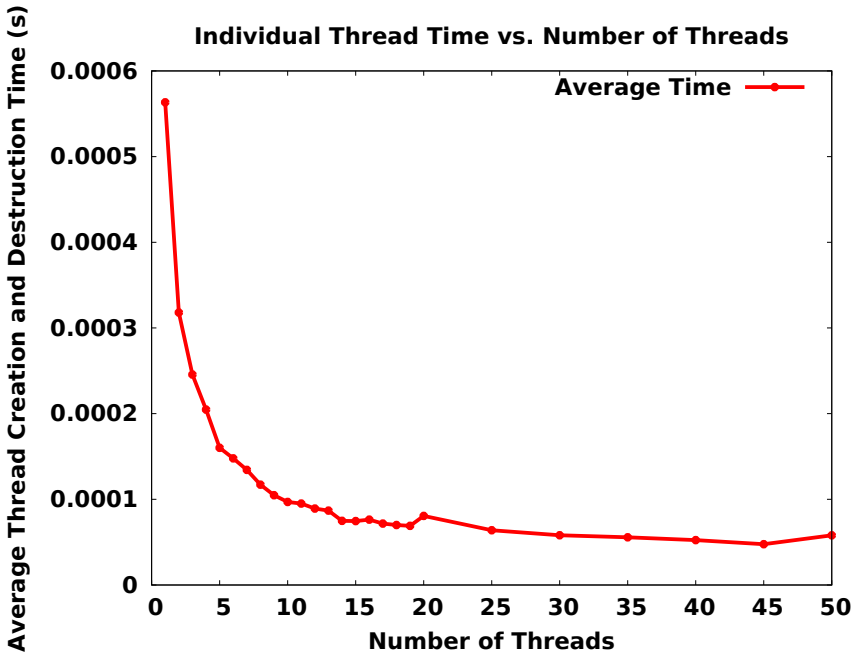

Figure 1: The plot showing the same data as Table 1.

The code that creates, destroys, and measures the time of each thread is included with this submission as `q1.c`. If you wish to run the program for yourself and collect your own data, I have a script that does so, outputs the results, and makes plots and tables for you. I will also include this with the submission as `proj2.sh`.

Histogram problem

We started with a serial implementation of a histogram problem provided to us by Dr. Ribbens. The goal was to use the `pthreads` library to parallelize the code. Further requirements were that the code had to run in $O(n/t + b)$ time where n is the number of data elements, t is the number of threads, and b is the number of bins to sort the data into.

There were two main aspects to parallelizing the code. The first had to do with breaking the initial data array up into n/t parts, and having each thread do some bin sorting on each part. Once that is accomplished, all threads wait until all sorting has been done. Then they work to consolidate all the bins into one giant count of bins, each thread summing up an index of each local bin.

That's the basic idea. The source code might be more illuminating. Describing what the code does at a higher level doesn't quite do it justice, because there's a lot of small details that went into making sure edge cases were handled correctly, and also making sure that the runtime fit the requirements for full credit. THe source code for part 2 is included with the submission as `histogram.c`. You will also need to link the math library (`-lm` flag) in order to compile it. If you run the `proj2.sh` script that I wrote, it should compile it for you so you won't have to worry about anything. I suggest you do that.