CSC2001F
Assignment 3

GDFLAW001
Lawrence Godfrey

7 September 2019

Introduction

Multithreading is an increasingly important part of computer science due to its ability to increase speed and responsiveness while still being more economical since multiple threads can share a processes recoucources.

Multithreading also takes advantage of the increasing number of physical cores in consumer level and commercial level computers. Normally threads will share time on a single processor, however, with multiple cores, multiple threads can run simultaneously in parallel.

To test the theory that parallelisation can cause speed up I will compare the run-times of the same java program using sequential code and threaded code.

Methods

In order to compare the relative speed of the threaded version to the unthreaded version I ran each version 6 times, then varied the sequential cutoff and ran the threaded version another 6 times. I varied the sequential cutoff between 17 values from 2 to 8,000,000. I always ignored the first run-time since it is much longer than the last 5 and tends to skew the data.

I only benchmarked using the largesample_input.txt as the input textfile, and compared my output to the sample output given using a site which allows you to compare text.

I used the System.currentTimeMillis() function to record the time.

I followed this same method on 3 different computers:

- A laptop with a 4-core Intel i5-7200U CPU @ 2.50GHz (x86_64)
- A Raspberry Pi with a 4-core Cortex-A53 @ 1.4 GHz (armv7l)
- A Desktop with a 4-core Intel i5-2500 CPU @ 3.30GHz (x86_64)

I then chose the optimal sequential cutoff for the desktop and compared the threaded and unthreaded version with a varying dataset size. I used another java script to generate files of varying sizes.

I plotted all the results below, using a logarithmic scale on the horizontal axis for the first 3 graphs.

Results

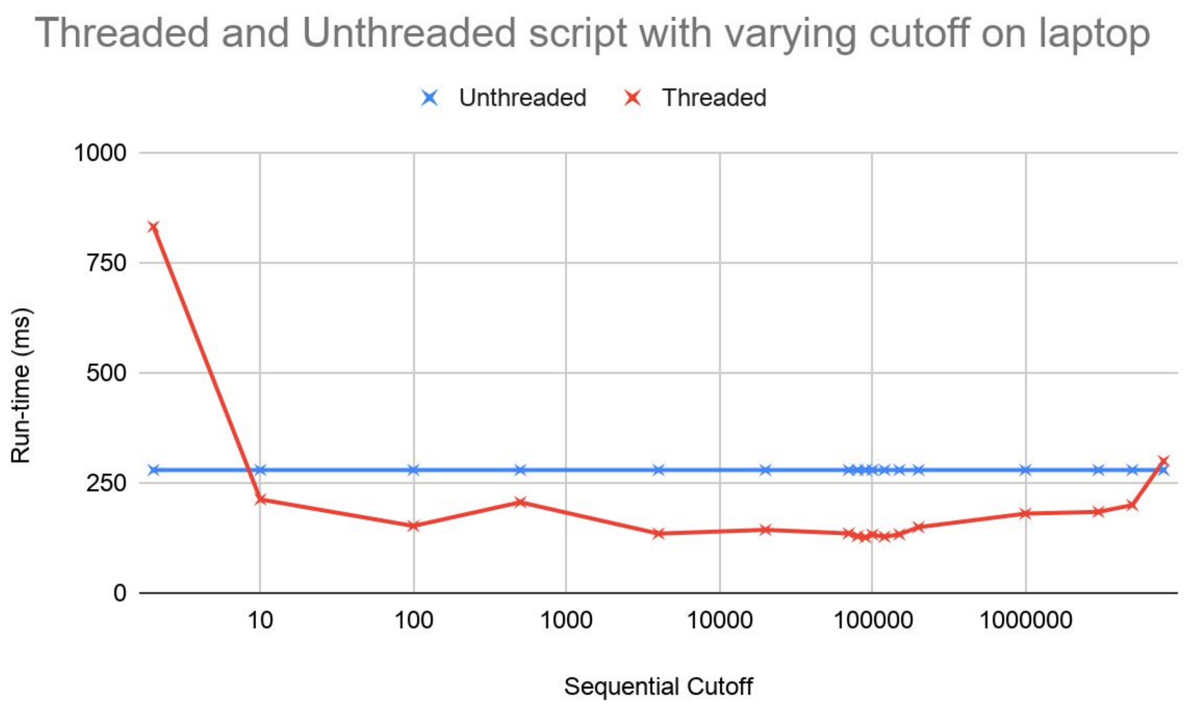

Figure 1: Threaded and Unthreaded on laptop

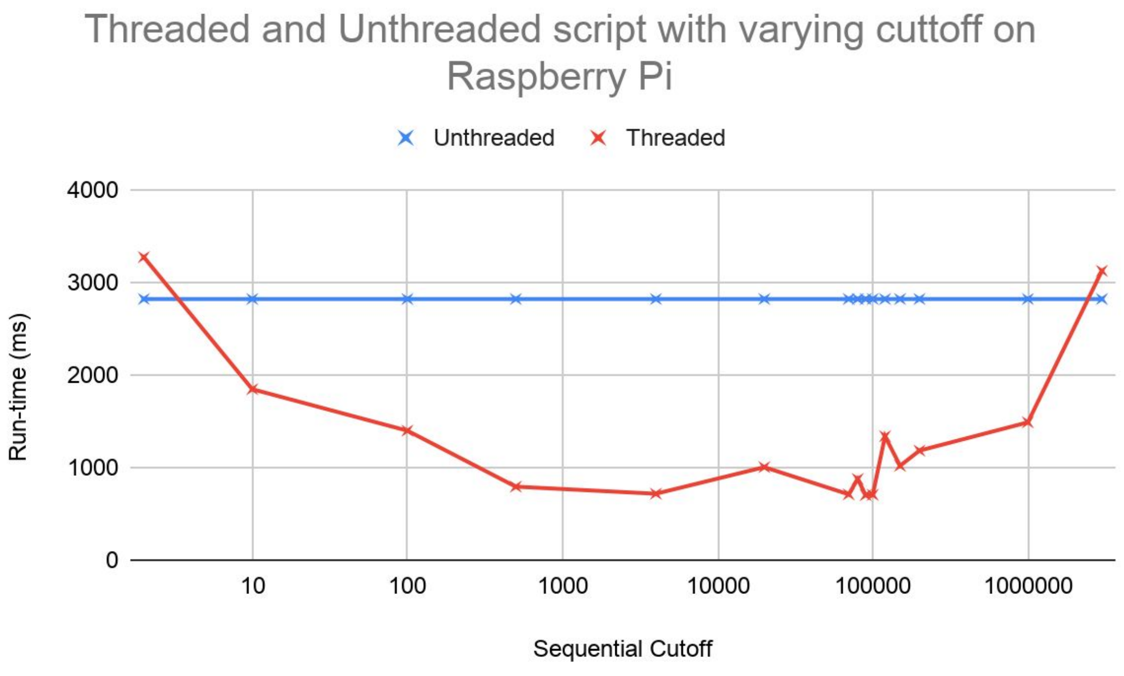

Figure 2: Threaded and Unthreaded on Raspberry Pi

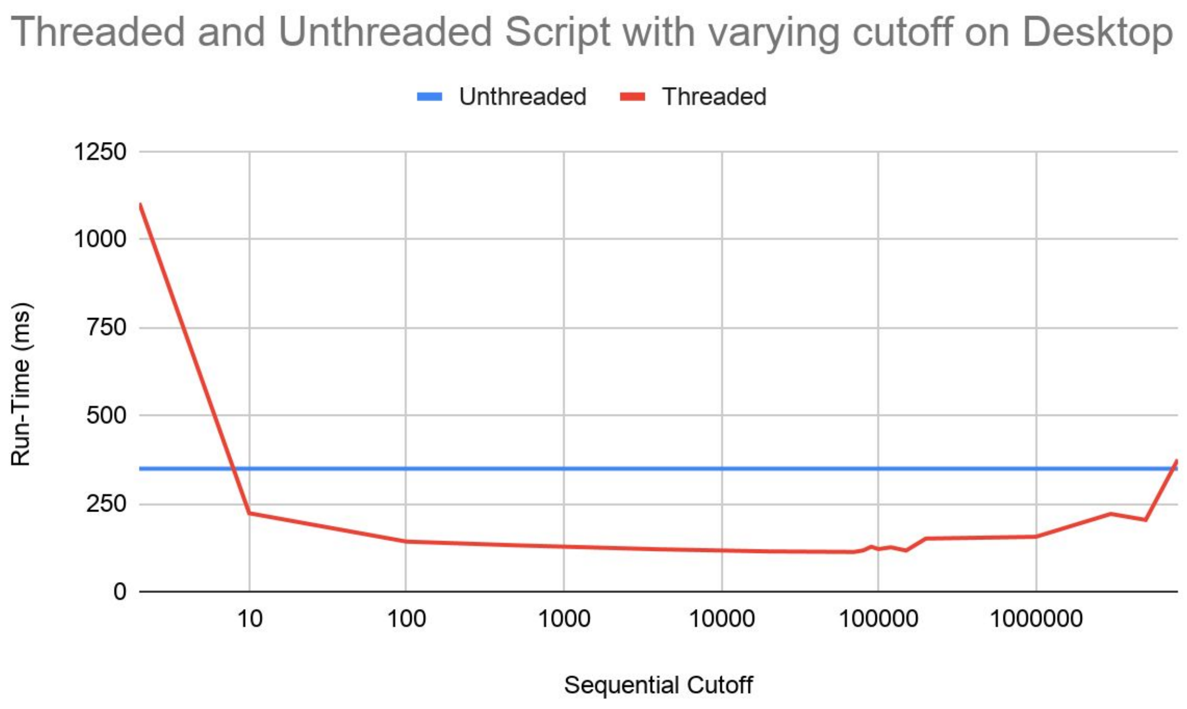

Figure 3: Threaded and Unthreaded on Desktop

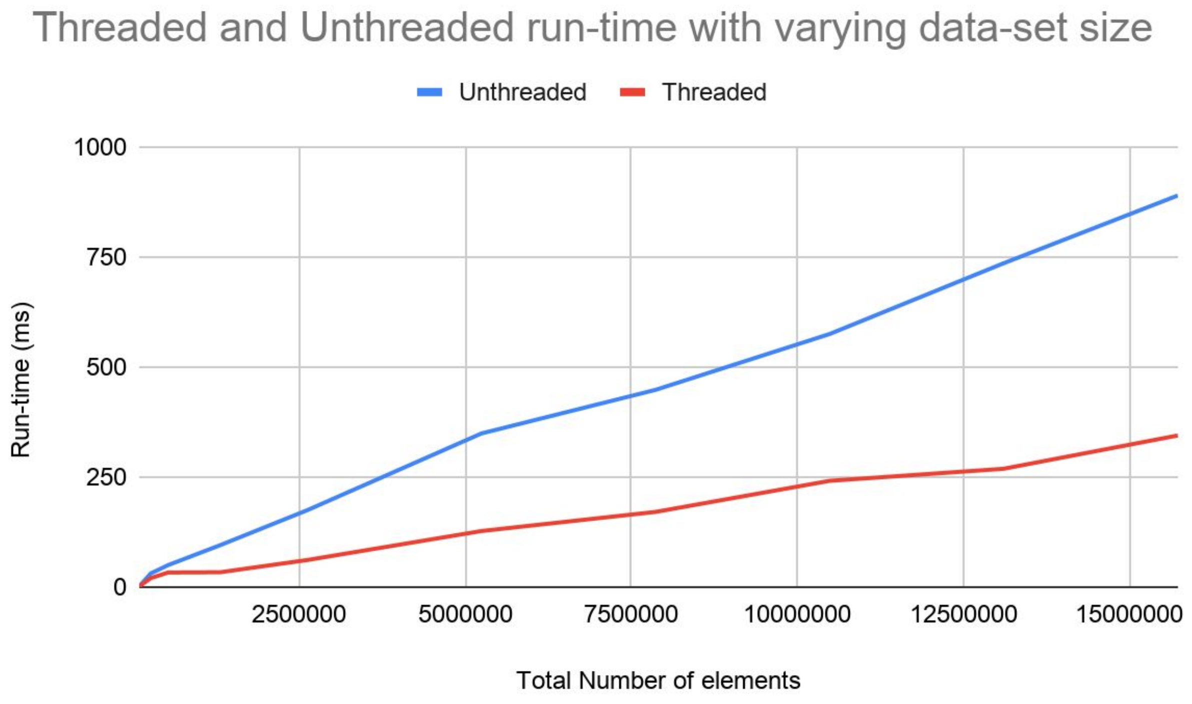

Figure 4: Threaded and Unthreaded with varying input data-set size

Discussion

Figure 1: Laptop
We can see that in Figure 1 when the sequential cutoff goes below 10 or above around 8 million, the threaded version starts becoming slower than the unthreaded version. This is because when there are too many threads, the overhead for creating the threads becomes larger than the speedup that parallelism provides. If we have too few threads, the workload isn't distributed evenly, and a core could be doing work while the other cores have already finished running their threads.

The ideal cut-off in this scenario is around 90,000, which equates to about 58 threads being used. We get a speedup of $\frac{279.2}{126.6} = 2.2$. This is definitely a good speed up when one takes into account the fact that this program realistically would be run on large datasets or would be run constantly. If the program ran at half the speed it could easily cause a backlog where new datasets are being sent in but the program hasn't finished computing the last dataset.

The run-time, however, doesn't change much between a cutoff of 2000 and 100,000.
Figure 2: Raspberry Pi
The run-times here are clearly much longer due to the slower processor speed, however, we do get a larger speedup compared Figure 1. Again the optimal cutoff is at around 90,000, but this time the run-time seems to vary quite drastically with cutoff. The speedup is $\frac{2819}{702} = 4$.

Figure 3: Desktop
The run-times here were the lowest, and varies the least with cutoff. Here the optimal cutoff was around 70,000, equating to about 75 threads being used. However, the run-times were fairly stable between cutoffs of 100 and 80,000, meaning using 75 threads or 52,000 threads doesn't impact the runtime much at this size.

The speedup here was $\frac{349}{112.6} = 3.1$.

Figure 3: Varying dataset
Here I varied the dataset from 90,000 total elements to 15,728,640 elements. This shows how at 90,000 elements there is no speedup due to the fact that the parallel version is essentially only creating one thread, and therefore running the program sequentially. It also shows how much better the parallel version performs at high dataset sizes.

The speedup at 90,000 elements is 0.89, while at 15,728,640 elements it is 2.6.

Conclusions

Parallelism definitely can have a large speedup even at relatively low dataset sizes, however, for trivial computations it ends up being as slow or even slower than a sequential program due to the overhead involved in creating the threads.

Sometimes there can be large variations in the number of threads being used without much variation in speedup. This might need to be taken into account if you don’t want to create unnecessary threads that aren’t contributing to speedup.

Too few or too many threads can also lead to a program running slower than its sequential version.