Edward Alkire - 1001722331 CSE 3320 – 003 March 26, 2023

MT Star Catalog Report

Summary:

In this report, we explore the performance characteristics of a CPU bound multi-threaded workload. The workload in question is computation of the average, min, and max angular distance of a 30k star catalog. The primary anomaly explored is the behavior under "overthreading" or the allocation of more threads to a multithreaded process than it has access to in hardware. Testing was performed on a "GitHub Codespaces" VM, and an M1 Mac for comparison. The core issue which overthreading improves may be the bottleneck at the thread-join loop related to thread balancing. Some plausible explanations for improvement with overthreading are related to thread variance reduction. Testing results reflected 1000 threads as optimal.

Libraries used:

- To measure time, timespec with clock_gettime() was used. Timespec utilizes the OS time/clock. This as opposed to clock_t variables with clock(), which measure time via CPU time, which multiplies the time measured in multithreaded contexts — meaning that the real time passed would no longer be the subject of measurement. Alternatively, timeval could have been used, as it operates similarly to timespec.
- Pthread was used for thread creation as it is the POSIX standard for multi-threading, meaning it is standard to UNIX based operating systems.

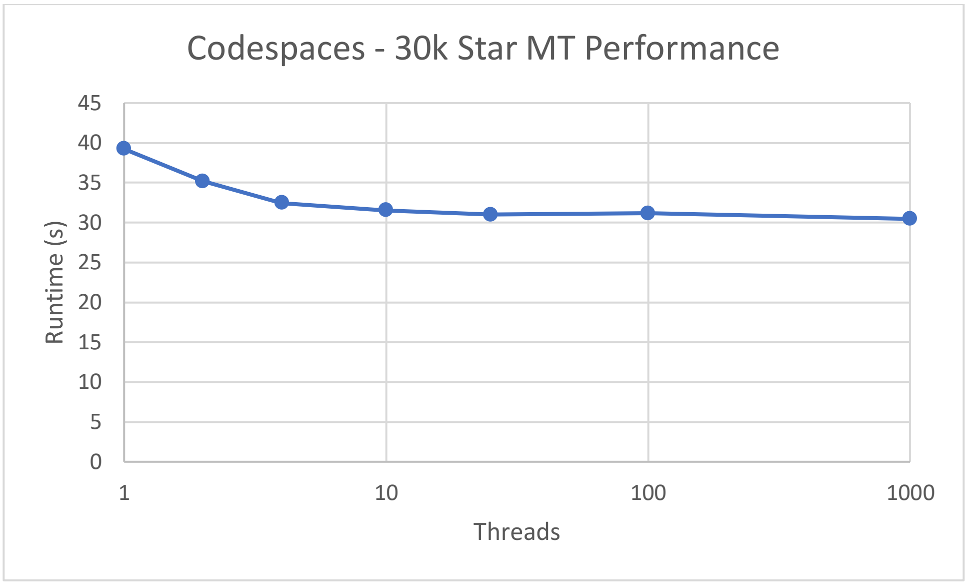
Steps to run:

1. Compile with: "gcc src/main.c src/utility.c -lpthread -lm -fPIC -mcmodel=large"
2. then run "./a.out -t <Num threads to use>"
3. If the program functions correctly, expect the following results no matter the number of threads used:
   a. Average Distance: 31.904231
   b. Minimum Distance: 0.000225
   c. Maximum Distance: 179.569720

Results:

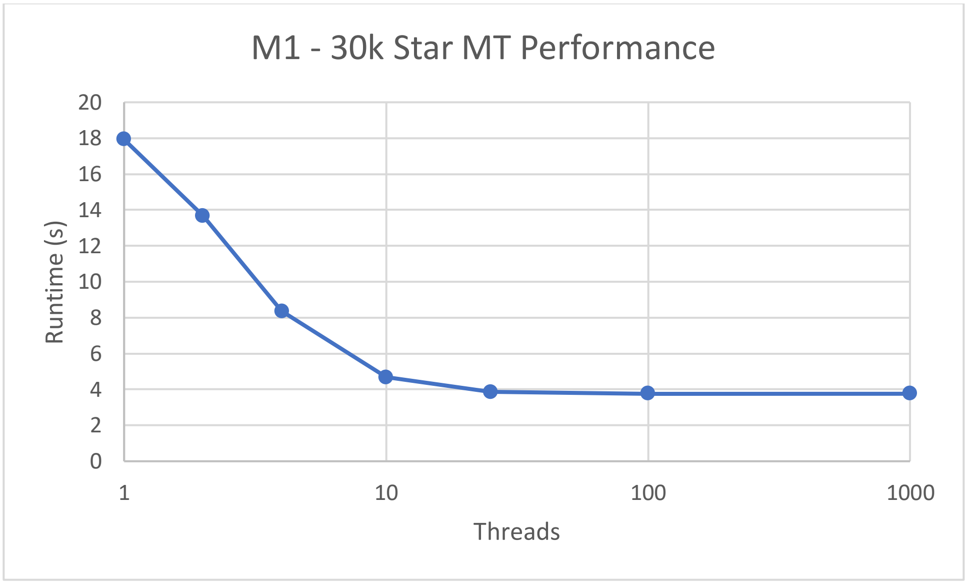
Codespaces (2c/4t) + 4GB RAM

Threads:	Time(s):
1	39.235158
2	35.182965
4	32.456121
10	31.536769
25	31.016687
100	31.154934
1000	30.465682



M1 Mac (8c/8t) + 8GB RAM

Threads:	Time (s)
1	17.924284
2	13.677046
4	8.349596
10	4.667984
25	3.859966
100	3.767594
1000	3.760357



Interpretation:

One apparent abnormality in the results is how the best performance was found when the number of threads created was far more than the number of threads the program had access to in hardware. That characteristic seems counterintuitive, given that creating more threads entails overhead cost, and that the hardware should be fully utilized once the number of threads used matched the amount in hardware — that is until you look at the code. An apparent bottleneck lies within the runThreads() function, in the pthread join loop. The join loop notably operates by joining threads particularly by their thread number (the order they are created), not simply by the order the threads finish. This means any variance in turnaround time between successively created threads becomes a form of overhead, with the process left waiting between joins.

We've identified where variance becomes a problem, but where is this variance created? The determineAverageAngularDistance() function may have an answer. The double for-loop is responsible for enabling each thread to calculate the distance of its allocated stars. Notice how the inner loop does not cover the entire star range but skips to the index of the outer loop. This saves cycles by skipping redundant indexes but also contributes to thread time variance since each successive thread will start at a higher j index, have a smaller workload. It seems the benefit observed with higher thread count is related to reduced variance with smaller workloads per thread, reduced size difference between successive threads.

Interestingly the M1 seems to benefit more from overthreading than the VM. Perhaps the varying behavior could be attributed to the difference in processor architecture. They use very different processors, a server grade intel x86 vs a laptop arm processor. The most notable difference might be how the intel chip is a typical symmetric processor, whereas the M1 chip is asymmetric, having half "performance" cores and half "efficiency" cores. Perhaps the unequally performant cores lend to unequal thread completion times. If this is the case, one might expect improvement from higher thread count, such that there are enough different threads for the chip's algorithm to effectively load balance between the greater and lesser performant cores.

The specific causes may have unclear relevancy to performance, but what is certain is the testing results which reflected 1000 threads as optimal, even across two dissimilar processors. This supports the conclusion that in large workloads (especially highly CPU bound ones), the optimal number of threads to use is not as much a factor of the number of cores available as is typically understood, but a factor of the total work size.