Optimal Multi-Threading
And
The Effect of Specific Quantities of Threads

By Eduardo Ramos

UTA ID: 1001834020

Overview:

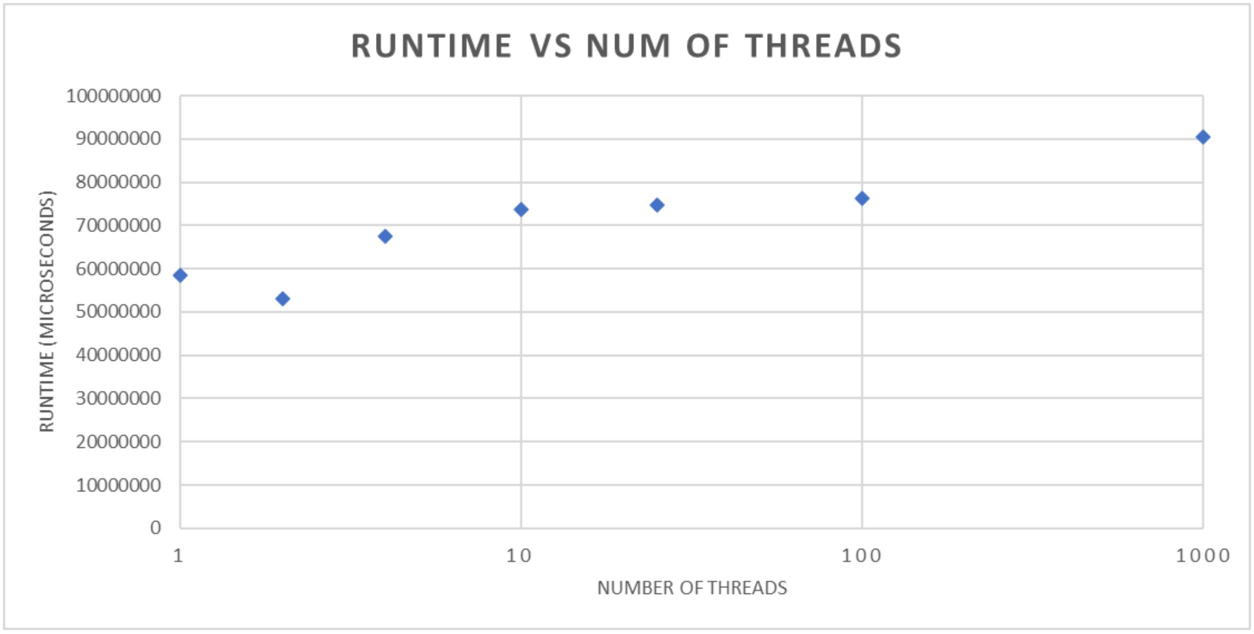
When it comes to running calculations on large sets of data, it is quite difficult to compute such operations when only using one process. Therefore, to facilitate the burden of these calculations, multithreading is regarded as a quick but delicate solution to deduce information from large data sets. How can threads be the solution? To begin with, threads are the smallest unit of a program that can be scheduled by the scheduler. But unlike processes, threads do not require their own address space, rather, they share it with the process from which they were created. Another advantage of threads is they require few resources from the system than a process would. Thus, making multiple threads would cost the OS little compared to making multiple processes. Hence, when used with the appropriate tools, threads could lower the time needed to process large amounts of information to a certain extent. Through the use of multi-threading, we were able to successfully calculate the angular distance of 30,000 stars in less time than one process would take.

Description:

To create the threads we will use to multi-thread the program for this assignment, we included the library file “pthread.h”. However, we also had to link our program with the pthreads library before compiling, which is done with the “-lpthread” command. Other libraries that were included in the program are the standard libraries “stdio.h” and “stdlib.h”, which give us the basic functionalities to help our code display the correct information. Moreover, we also included the library file “sys/time.h” to get an accurate reading of how much our code speed improved as the number of threads increased. The function we used for our timing method was the get time of day function. The reason why we choose this function is that it measured how long our threads took to compute in microseconds, which offers a degree of accuracy that other methods can not. Through this timing method, it was easier to observe the difference in time when we tested for the optimal number of threads.

Results:

Number of Threads	Runtime (Microseconds
1	58498608
2	53177232
4	67539024
10	73717341
25	74649486
100	76210654
1000	90406347



Discussion of Anomalies:

When computing our results, it was observed that our time increased linearly despite the number of threads. Although it was assumed our code was the source of the problem, it was later concluded that the machine we used to run our code was the source of the problem. Through some research, we found that the limitations of our codespace affected how our program performed. To minimize how much this anomaly affected our result, we ran the program multiple times at different times to gain reliable data.

Conclusion

Overall, through this assignment, we concluded that the optimal number of threads to reduce the amount of runtime is 2. Furthermore, we observed that any number above or below this number will result in a runtime higher than approximately 50 million microseconds. The explanation for this behavior is that our codespace has only 2 cores available. Thus, any number of threads that only use 1 core or overwhelm the number of cores will result in a high runtime. Furthermore, another reason the optimal number of threads is 2 is how the work is divided. If we were to use a number lower than 2 as the number of threads, it would compute the same results as if we were running the program in series. On the other hand, if the number of threads were to be bigger than 2, the amount of work each thread would have to do would be sub-optimal, which would increase the runtime.