Mary-Rose Tracy

03/30/2023

ID#:1001852753

CSE3320-001

Report on Multithreading Program:

brief executive summary/ overview:

In my Operating Systems class, we are assigned a program that is incomplete. What we essentially had to do was to implement in the coding language of C “p threads” to make the program compile. We also had to time how long it takes for the program to compile utilizing “-t.” Not to mention we had to somewhat alter the original code itself in order to implement these changes. The code cannot compile with any errors, race conditions, nor should it deadlock. Our goal from the program was to compute the distance of angular stars within the program with the threads. The lesson of this assignment is to see what is the optimal number of threads to compile the 3000 records. In the end, we had to calculate the time it took in seconds for 1,2,4,10,25,100, and 1000 threads.

description of your effort to include which libraries, if any, you used, and the reasons you chose your timing methods:

1. At first, I did typedef struct, but then my program became extremely complicated. So, I decided to start over, and fix the program as simple as possible.

2. We have to do "#include <pthread.h>", along with using the variable "pthread_mutex_t Mutex;." We then used the mutex to in between the "determineAverageAngularDistance" function, as well as "ThreadFuncOMain" method. The first function we needed to lock and unlock the threads like a key, if the thread is already locked because of another thread. Then the mutex lock will unlock it. In other words, release the thread so we can change it. The same is kind of going on for the "ThreadFuncOMain," but we dereference it.
3. I noticed that some variables where being individually used in each method. To make it easier I decided to make them global variables.
4. The original code the "determineAverageAngularDistance" array had 3 j's instead of 2 j's, and 2 i's.
5. The code already had the function that if the user types help then it'll print out the necessary stuff for the program. However, the program could not read -t so I implemented that in the method using the similar code from the help function. Only making the Number of Threads or (NOThread) increment. Thus, changing the threads. If the program is less than 2 then the program has to read it as one thread because if it's less then one it's still 1 thread. Kind of like the theoretical graph for an empty set. Tried n++, but for some reason when I ran the time was going everywhere. I decided on n+1, and it somehow worked. That n+1 gave me a lot of trouble.
6. Printing out the time it took by having #include <time.h>. Now usually you would have to do clock_t start & end with a function. I tried that, and I kept getting weird numbers. Since I know the function clock() is how it essentially is the full time I subtract it from

the clock_t time to get the All of The Time (AOTTime). Then in another function I make the integer a double then I divide it by the primitive function of CLOCKS_PER_SEC.

7. Synthesized, commented, and tried to make my code work more efficient. However, those changed made the timing a little bit longer, but it was the exact same result.

Results in both graph and table form:

Graph:

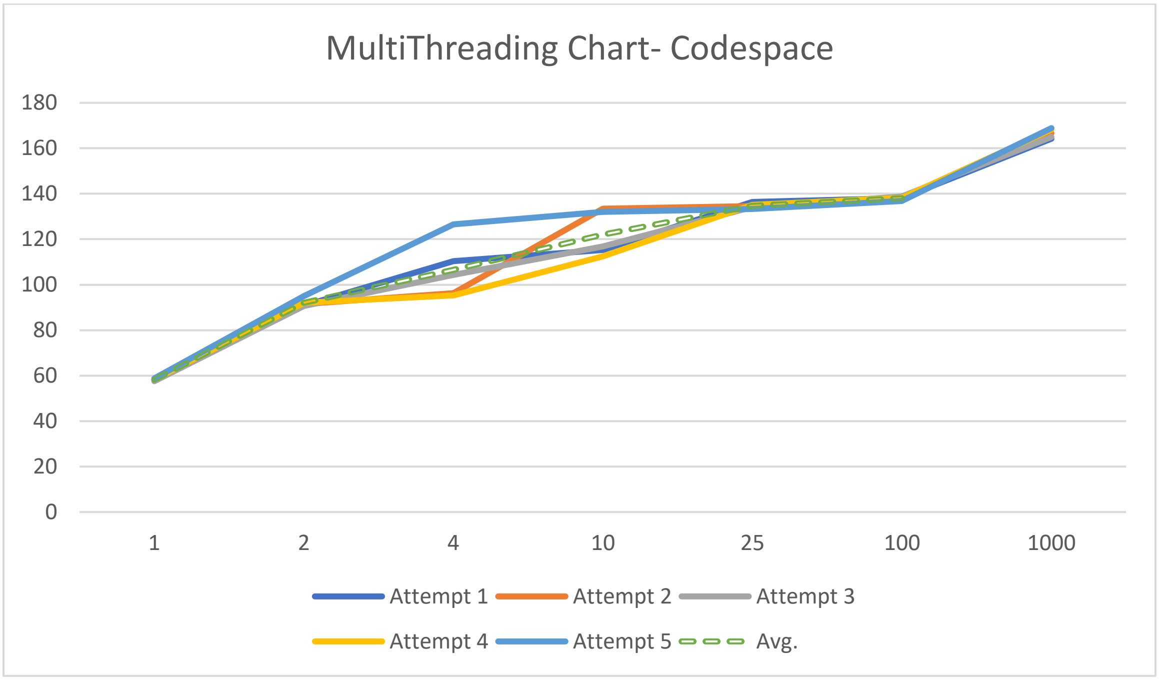

Table:

Thread -t	Attempt 1	Attempt 2	Attempt 3	Attempt 4	Attempt 5	Avg. of Attempts:
1	58.247	58.093	57.556	58.331	58.798	58.205
2	90.826	91.531	90.713	92.092	94.998	92.032
4	110.342	96.239	104.253	95.339	126.607	106.556
10	115.222	133.489	116.806	112.516	131.973	122.001
25	136.327	134.549	134.174	135.096	133.237	134.676
100	138.151	138.363	138.744	138.185	136.868	138.062
1000	164.205	166.619	164.980	168.252	168.877	166.586

discussion of any anomalies you found in the resulting data:

From the resulting data from code space, as the thread increases the amount of time increases. However, when it comes to thread two it dips a little. This means that it is the most efficient number of threads. Conversely, as the threads increase after thread two it increases more and more. This is similar to graphs in algorithms and data structures. Where if you solve a certain set of files, it's efficient in terms of time, but if the files are slightly bigger then at that certain point it becomes more time consuming for the algorithm or vice versa. Although this is depended on what algorithm you utilize. In the local computer it is more apparent that two threads are the most efficient for the overall goal. Being that there is big dip between one to four.

Conclusion explaining the optimal number of threads:

In conclusion, if there's a choice between compiling in multiple files in Code space or in your local computer. You should compile in your local machine because Code space has a not so apparent output. The optimal number of threads is two. Although, I would also like to argue that four threads are the second most efficient thread compared to the other threads that were used in this given assignment. In spite of the given data from specifically Code Space. I believe that I acquired these certain results for a plethora of reasons. For starters, it may be due to the fact that an even number that is small enough divides the given files enough for the stars to be calculated. However, if its more than two to four then the program has so many threads to solve that it takes more time. Maybe that's also due to the fact that in binary it's solving in 0's and 1's. That may be the reason why in the cloud of Code Space it doesn't show an apparent deviance in the timing of threads. Whereas, a local machine is more deterministic compared to the cloud which is nondeterministic. In the end, if you want to calculate anything by threads in a file. You have to keep it short like two threads, and also utilize an actual machine.