CSE3033 Project 3 Report

Eren Duyuk – 150120509

Ufuk Acar – 150121071

Furkan Acar - 150121534

Introduction:

This project centers on exploring the impact of mutex usage in multithreaded applications by implementing three distinct methods for computing the sum of square roots within a user defined range. The program takes four input parameters: the range limits (a and b), the number of threads (c), and the method identifier (d).

Method 1 involves concurrent updates to a global sum variable without any mutex usage. This method exposes the challenges of simultaneous thread operations without synchronization.

Method 2 introduces a shared mutex, ensuring serialized updates to the global sum. The goal is to evaluate the trade-offs between preventing race conditions and the associated synchronization overhead.

Method 3 employs local variables for partial sum calculations, using a mutex-protected global sum variable for aggregation. This method strikes a balance between concurrency and synchronization.

The project focuses on assessing how mutexes influence performance in varying multithreaded scenarios. By comparing the sum value, user time, system time, and total time across the methods, the study aims to draw insights into the practical implications of mutex usage in different thread contexts. This exploration provides valuable lessons for optimizing concurrent program performance by understanding the nuances of synchronization mechanisms.

Thread - Method	Sum Value	User Time	System Time	Total Time
1 1	4.05346e+16	151,82s	0,20s	2:34,57
2 1	2.03578e+16	159,45s	0,13s	1:20,42
4 1	1.05924e+16	170,87s	0,41s	0:43,887
8 1	8.68311e+15	297,16s	1,06s	0:43,646
1 2	4.05346e+16	380,45s	0,23s	6:23,89
2 2	4.05346e+16	537,30s	461,48s	11:11,47
4 2	4.05346e+16	891,21s	2432,41s	19:50,38
8 2	4.05346e+16	982,11s	3474,95s	20:06,67
16 2	4.05346e+16	1251,58s	7997,69s	24:50,41
32 2	4.05346e+16	1280,07s	9072,23s	25:13,16
1 3	4.05346e+16	151,69s	0,07s	2:32,81
2 3	4.05346e+16	154,82s	0,11s	1:18,19
4 3	4.05346e+16	161,60s	0,27s	0:41,103
8 3	4.05346e+16	178,10s	0,71s	0:26,244
16 3	4.05346e+16	178,07s	0,85s	0:26,163
32 3	4.05346e+16	179,03s	1,01s	0:27,202

Questions

1) Which method(s) provide the correct result and why?

The second and third methods reach the correct result because the global_sqrt_sum variable is protected by mutex. In other words, while a thread is updating the global_sqrt_sum variable, other threads cannot access this variable. The reason why the first method sums incorrectly when run with more than one thread is because the global_sqrt_sum variable is not protected by mutex and a race condition occurs between threads.

2) Among the method(s) providing the correct result, which method is the fastest?

The third method is the fastest among the working methods. The reason for this is that each thread creates a variable and updates this variable, and then adds the value of this variable to the global_sqrt_sum variable when the summation is finished. While performing these operations, it uses mutex only once while changing the global_sqrt_sum variable. In the second method, the opposite is seen and the lock is opened and closed every time the global value is added, so a lot of mutex locks are opened and closed. We can clearly see that mutex locking and unlocking is costly when we compare the running times of the second and third methods.

3) Among the method(s) providing the correct result, does increasing the number of threads always result in smaller total time? Discuss this considering the number of CPU cores available in your computer (in Linux, lscpu command provides the number of CPU cores available in your computer).

In the second and third methods that give correct results, when you run them on a computer with 8 cores, it is expected that the running time will decrease up to 8 threads, and that the threads after 8 will run at times close to the running time of 8 threads. That's exactly what happened in the third method, but in the second method, the running time did not change much after 4 threads. The reason for this was that as the number of threads using the mutex increased, the system time also increased. When we look at lscpu, as the number of threads increases up to 8 threads, the number of cores running the application also increases, but in 8 or more threads, all 8 cores are used.

4) Are there any differences in user time/system time ratio of the processes as the number of threads increases? What could be the cause of these differences?

The ratio of user time/system time decreases as the number of threads increases, because number of threads increases, the work that the system has to do increases as the operating system provides work sharing and communication between threads, but meanwhile, user time does not increase at this rate.