Introduction:

Aim of this project is to analyze and compare three methods (as unsynchronized access, mutex-based synchronization, and a combination of local computation with global synchronization) in terms of accuracy and efficiency by doing a simple operation like finding the sum of the square root of numbers in a range. With the help of this project, we will have a better understanding on the multithreading.

Implementation:

Project is made by C programming language in Linux operating system with especially help of the pthread library’s facilities on multithreading and time command of the Linux.

Three different method is asked for to calculate the summation of two numbers in a given range which is given by user. These methods can be listed as like:

Method 1:

In method 1 there is no mutex so that each thread calculates a segment of the operation and directly updates the global sum, due to direct access of these threads on global sum and not having a blocking system to these threads, the system is unsynchronized and the result is up to race conditions so that result is not always correct.

Method 2:

In method 2, each tread calculates a portion of the summation that is controlled by one mutex, this one mutex creates a synchronized update on global sum by letting only one thread update the global sum at time and prevents race conditions like in method 1 but also due to mutex locking, method 2 increases the time of the execution.

Method 3:

In method 3, each thread has a local sum and threads make calculations independently from other threads. After threads compute the their local\_sums, they lock their shared mutex and updates the global sum, thanks to this approach the system is synchronized just like method 2 but also reduces the mutex locks and the system is much faster than the method 2.

Results:

Method 1 Table 1

Example use of program

gcc -o project3.out project3.c -pthread -lm

time ./ project3.out 880130203012 922823372203 4 1

|  |  |  |  |
| --- | --- | --- | --- |
| Threads | 1 | 2 | 4 |
| Sum | 40534565385993456.000000 | 22210918099018148.000000 | 11164891402448928.000000 |
| Total | 1m38,973s | 2m56,158s | 3m17,627s |
| User | 1m38,865s | 5m37,221s | 12m38,462s |
| System | 0m0,076s | 0m5,363s | 0m11,889s |

|  |
| --- |
| 8 |
| 7463088639029211.000000 |
| 2m33,823s |
| 12m5,080s |
| 0m26,934s |

Method 2 Table 2:

Example use of program

gcc -o project3.out project3.c -pthread -lm

time ./ project3.out 880130203012 922823372203 4 2

|  |  |  |  |
| --- | --- | --- | --- |
| Threads | 1 | 2 | 4 |
| Sum | 40534565385993456.000000 | 40534565376248608.000000 | 40534565390450616.000000 |
| Total | 11m10,837s | 47m8,659s | 46m25,873s |
| User | 11m6,657s | 55m29,648s | 52m0,516s |
| System | 0m4,028s | 34m21,939s | 50m58,272s |

|  |  |  |
| --- | --- | --- |
| 8 | 16 | 32 |
| 40534565408639736.000000 | 40534565353047832.000000 | 40534565417417896.000000 |
| 23m52,957s | 16m58,346s | 23m52,957s |
| 27m16,309s | 18m4,742s | 16m16,733s |
| 47m55,786s | 23m46,033s | 20m24,477s |

Method 3 Table 3:

Example use of program

gcc -o project3.out project3.c -pthread -lm

time ./ project3.out 880130203012 922823372203 4 3

|  |  |  |  |
| --- | --- | --- | --- |
| Threads | 1 | 2 | 4 |
| Sum | 40534565385993456.000000 | 40534565391145360.000000 | 40534565390259656.000000 |
| Total | 1m36,022s | 0m51,425s | 0m28,513s |
| User | 1m35,696s | 1m42,511s | 1m50,559s |
| System | 0m0,279s | 0m0,174s | 0m0,801s |

|  |  |  |
| --- | --- | --- |
| 8 | 16 | 32 |
| 40534565389973904.000000 | 40534565390592728.000000 | 40534565390499128.000000 |
| 0m23,176s | 0m24,303s | 0m22,756s |
| 1m44,956s | 1m41,054s | 1m37,519s |
| 0m4,381s | 0m10,151s | 0m12,458s |

Analysis:

As we can see from these tables, we can make these assumptions:

In method 1, due to not using any mutex and letting(non-blocking) threads update the global\_sum the result from the same inputs are different even though we only change the number of threads. It is fast but also wrong so that method 1 is not a reliable option to choose when we work on higher numbers like in the inputs.

In method 2 thanks to mutex synchronization, we get much similar results when we change the number of threads and much closer to the correct than method 1 but due to frequently locking the mutex, it impacts on the time. Lastly, we clearly see that from this method, the more threads are not always means the faster program.

In method 3, by the help of the mutex locking and synchronization method 3 has much more accuracy than method 1 method 3 is reliable like method 2 but also and with the reduction in the frequency of the mutex locking method 3 is much faster than method 2. We can clearly see from the table that, method 3 and method 2 almost have the same accuracy but although we use more treads in method 2 than method 3. For example, table 3 4 threads and table 2 32 threads.

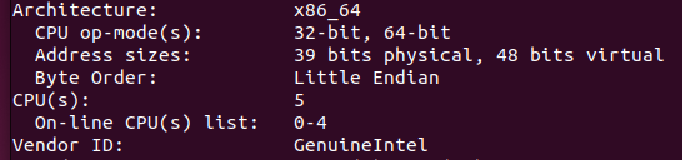
Conclusion:

In this project, we see the different multithreading implementations’ different effects on a simple calculation such as not synchronizing the threads is a much faster way to get result but we lose the accuracy. On the other hand, when we use synchronization but with a frequent mutex locking, we see that the accuracy is much more reliable than method 1 but even though we increase the thread numbers the performance does not completely change and it is so much slower but accurate. Last but not least, we see that balance between multithreading means much more efficient, accurate and also better performance of the program.

Direct Answers to The Questions:  
1. Which method(s) provide the correct result and why?

Methods 2 and 3 provides correct results because these methods implement mutex synchronization and that manages access to the shared variables and prevents race conditions like happens in method 1.

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

Method 3 is the fastest one because it both uses local calculations and reduced mutex lockings so 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). 

From the tables we can clearly see that more threads are not always means faster. We can have an assumption from the table 2, for the 5 CPUs having more than 8 threads does not mean better compile time and this is due to additional threads context switchs and additional threads does not always improve the time.

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?

If we look at the table 3, we can see that using more threads increases the system time although the user time is almost same. This happens because of the overhead from thread management and synchronization of the program.

Melih Sinan Çubukçuoğlu – 150122828

Şevval Alp – 150119815

Reyyan Rana İnan - 150199723