Studia i Materiały Informatyki Stosowanej, Tom 18, Nr 1, 2026
str. 35-38

ZALEŻNOŚĆ MIĘDZY ILOŚCIĄ CZASU POTRZEBNĄ DO WYKONANIA ZADANIA A LICZBĄ WĄTKÓW W JĘZYKU PROGRAMOWANIA C++

Aleksandra Mreła

Uniwersytet Kazimierza Wielkiego
Wydział Informatyki
Ul. Kopernika 1, 85-064 Bydgoszcz
e-mail: a.mrela@ukw.edu.pl

Streszczenie: *Czas wykonania programu jest bardzo istotnym elementem procesu akceptacji programu komputerowego do jego komercjalizacji. Paradymat programowania równoległego powinien spowodować przyspieszenie działania programu w przypadku posiadania większej liczby rdzeni. W pracy przebadano szybkość wykonania programu w zależności od liczby wątków. W różnych środowiskach programistycznych dla różnej liczby uruchomionych wątków uzyskano różne czas wykonania programu.*

Słowa kluczowe: *Programowanie równoległe, C++, wątek, czas działania programu*

The relationship between the amount of time required to complete a task and the number of threads in the C++ programming language

Abstract*: Program execution time is a crucial element in the process of accepting a computer program for commercialization. The parallel programming paradigm should result in program acceleration when using a larger number of cores. This work examines program execution speed, which depends on the number of threads running. In various programming environments, various program execution times were achieved for different numbers of running threads.*

Keywords: *Parallel programming, C++, thread, program runtime*

1. INTRODUCTION

A concurrent program consists of multiple computational tasks executed in parallel. A programmer must encode how the computational tasks interact with each other, how they use the results obtained by other tasks, and how the tasks are synchronized. If a computer has more than two processors (cores), individual tasks can be performed in parallel, which speeds up program execution time [1].

If a computer has more processors, individual tasks can be executed in parallel, which speeds up program execution time. If more tasks are run than the computer has physical processing units (CPU), the operating system uses interleaving, which is inefficient and time-consuming [2]. When transitioning to the next process, the processor performs context switching, which requires storing necessary data regarding the current process state. The processor stores the contents of arithmetic registers, register contents, and the instruction counter [3].

2. THREADS

Parallel programming utilises multithreading, a technique that divides a program into smaller execution units called threads. Each thread operates independently but shares resources (such as memory), allowing for the simultaneous execution of tasks. Executing tasks in parallel increases

performance because processor cores are used more efficiently [4,5].

Beyond the benefits of using threads, there are, of course, challenges, primarily caused by the sharing of resources (including memory) by threads running in parallel [6]. The most important problem is race conditions, a situation in which the program's outcome is determined not by program logic but by the order in which threads execute (their execution time), which can result in data inconsistency [7].

Another problem to solve is deadlock, a situation where threads block each other's access to resources, causing all threads to wait indefinitely for each other. Another problem is starvation, a situation in which a lower-priority thread never receives enough processor time or resources because other, higher-priority threads constantly occupy them, so the task it is supposed to perform never completes [8].

Furthermore, a livelock can occur, which is similar to a deadlock, except that threads are not blocked. They actively change their states in response to each other but do not perform any useful work [9]. Memory visibility issues can also arise if, in modern processors, threads can store data in their own caches instead of the main RAM. If one thread changes data, other threads may not immediately see the change. Finally, over- or under-synchronization can occur, where excessive use of locks causes threads to wait for each other, thereby reducing performance. In undersynchronization, the lack of appropriate locks leads to race conditions and data corruption (Nayak & Basu, 2024).

3. IMPLEMENTATION OF THE APPLICATION IN C++

To analyze the relationship between the time spent executing the program and the number of threads applied, the two versions of the algorithm are written in the C++ programming language and implemented in two programming environments: Visual Studio 2022 and OnlineGDB. The application calculated the sum of a hundred values, presented all intermediate sums, and measured the time spent executing the program. The application ran the number of threads between 2 and 10.

The first version defined the chosen number of threads, which are started with a function that sums the hundred values divided by the number of threads; however, if 100 divided by the number of threads is not an integer, the result is corrected.

The second version used Open Multi-Processing (OpenMP), an open-source Application Programming Interface (API) for shared-memory parallel programming in C++. In this application the pragma parallel was defined:

```
#pragma omp parallel for
    num_threads(MAX)
```

Pragma `omp parallel` creates a parallel region in which a block of code is executed concurrently by multiple threads, the number of which is specified using the `num_threads()` function. This is the basic directive for parallelization in the C++ programming language when creating child threads. In applications, it is used with the pragma for (`#pragma omp parallel for`) to divide work within loops. To measure time the `omp_get_wtime()` is applied.

The OnlineGDB environment does not specify the number of processors installed; however, Visual Studio 2022 is installed on the computer with 4 CPUs.

4. RESULTS

The OnlineGDB online programming environment runs both versions of the application, and in each case, 10 program runtime measurements were taken, varying the number of threads. The results are presented in Table 1.

Table 1. The first version program runtime measurement for a chosen number of threads (OnlineGDB)

Nr	The number of threads			
	2	3	5	10
1	0.0001	0.0001	0.0004	0.0006
2	0.000086	0.000191	0.000511	0.000483
3	0.000086	0.000186	0.000181	0.00077
4	0.000101	0.000133	0.00051	0.000987
5	0.000081	0.000328	0.000222	0.000588
6	0.000092	0.000217	0.000376	0.001365
7	0.000117	0.000816	0.000216	0.000252
8	0.000079	0.000195	0.000379	0.000904
9	0.000097	0.000246	0.0003	0.000855
10	0.0001	0.000171	0.000735	0.000242

Then the second version of the program was applied in OnlineGDB, and the times were measured and presented in Table 2.

Table 2. The second version program runtime measurement for a chosen number of threads (OnlineGDB)

Nr	The number of threads			
	2	3	5	10
1	0.01935	0.00704	0.0004	0.00116
2	0.02222	0.01323	0.0134	0.01412
3	0.00673	0.0029	0.00262	0.00273
4	0.00469	0.00477	0.00194	0.00136
5	0.00561	0.00472	0.00088	0.00102
6	0.0257	0.00583	0.00136	0.00159
7	0.00143	0.02257	0.00079	0.01456
8	0.00814	0.0049	0.00178	0.00246
9	0.00345	0.00483	0.00135	0.0062
10	0.00176	0.02388	0.00065	0.00138

Next, the first and second versions of the program were applied in Visual Studio 2022, and the times were measured and presented in Tables 3 and 4, respectively.

Table 3. The first version program runtime measurement for a chosen number of threads (Visual Studio 2022)

Nr	The number of threads			
	2	3	5	10
1	0.001	0.001	0	0.003
2	1.00E-03	0.001	0.002	0.004
3	0.00E+00	0.001	0.002	0.005
4	0.001	0.001	0.002	0.004
5	1.00E-03	0.002	0.002	0
6	1.00E-03	0.001	0.002	0.004
7	0.001	0.001	0.002	0.004
8	1.00E-03	0.001	0.002	0.004
9	1.00E-03	0.001	0.002	0.005
10	0.001	0.001	0.003	0.004

Table 4. The second version program runtime measurement for a chosen number of threads (Visual Studio 2022)

Nr	The number of threads			
	2	3	5	10
1	0.13436	0.12046	0.12923	0.14047
2	0.1257	0.14171	0.1478	0.14256
3	0.11085	0.13872	0.13592	0.16635
4	0.11589	0.14045	0.1376	0.1494
5	0.13977	0.19136	0.13432	0.14705
6	0.13148	0.13175	0.15152	0.15052
7	0.12742	0.12409	0.12818	0.15091
	0.1316	0.15592	0.15241	0.15394
9	0.13385	0.13976	0.12075	0.15168
10	0.13039	0.14615	0.14996	0.15453

Based on the data presented in Tables 1 – 4, average measurement times were determined; Fig. 1 illustrates the first version program, and Fig. 2 presents the second one.

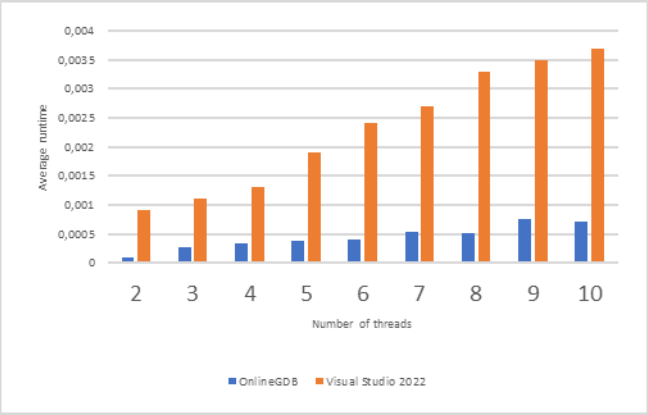

Fig. 1. Average task execution times of the first version program

In the case of the first version program, the fewer the number of threads running, the shorter the execution time is. However, OnlineGDB does not specify the number of CPUs; running two threads is the most effective. Visual Studio 2022 could utilize four available cores; however, running two threads was less time-consuming.

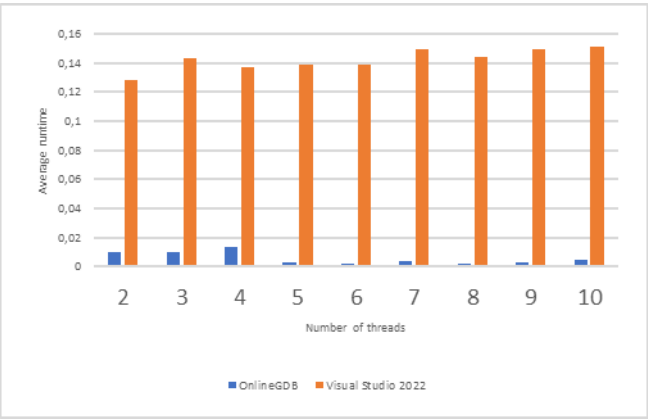

Fig. 2. Average task execution times of the second version program

In the case of the second version program and Visual Studio 2022, the required execution time was not affected by the chosen number of threads; however, two and four threads required the shortest amount. However, in the case of OnlineGDB, the smallest amount of time is observed for five threads, then seven and six threads.

5. SUMMARY AND CONCLUSION

Program execution time is a very important consideration for programmers. When deciding whether to use multithreaded or sequential programming, it's important to measure the application's execution time. The same applies to verifying the number of threads to choose.

Research has shown that the number of threads has a significant impact on program execution time. For OpenMP, 4-6 threads were the right choice for achieving the shortest execution time. For non-OpenMP programs, two threads were the best choice.

Therefore, when writing a multithreaded program, it's essential to verify the program's execution time with different numbers of threads in different C++ programming environments.

Bibliography

1. Tanenbaum A.S., Bos H., “Modern operating systems”, Pearson Education, Inc., 2015.
2. Williams A., “C++ concurrency in action”, Wyd. Simon and Schuster, 2019.
3. Czech Z., „Wprowadzenie do obliczeń równoległych”, Wyd. 2. Warszawa: Wydawnictwo Naukowe PWN, 2013.
4. Silberschatz A., Galvin P. B., Gagne G. “Operating system concepts essentials”, Wiley Publishing, 2013.
5. Ben-Ari M., “Principles of concurrent and distributed programming”, Pearson Education, 2006.
6. Chen G., Stenstrom P., “Critical lock analysis: Diagnosing critical section bottlenecks in multithreaded applications”, SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, USA, 2012, pp. 1-11, DOI: 10.1109/SC.2012.40.
7. Matuszek M., „Zjawisko wyścigu w programowaniu współbieżnym." Wydawnictwo Politechniki Gdańskiej, 2021, str. 261-270.
8. Zöbel, D. "The Deadlock problem: a classifying bibliography". ACM SIGOPS Operating Systems Review. 1983,17 (4): 6–15., doi:10.1145/850752.850753.
9. Ganai, M.K., “Dynamic Livelock Analysis of Multi-threaded Programs”. In: Qadeer, S., Tasiran, S. (eds) Runtime Verification. RV 2012. Lecture Notes in Computer Science, vol 7687. Springer, Berlin, Heidelberg, 2013, https://doi.org/10.1007/978-3-642-35632-2_3.
10. Nayak A., Basu, A., “Over-Synchronization in GPU Programs,", 57th IEEE/ACM International Symposium on Microarchitecture (MICRO), Austin, TX, USA, 2024, pp. 795-809, doi: 10.1109/MICRO61859.2024.00064, 2024.

OnlineGDB, https://www.onlinegdb.com/online_c++_compiler

Open Multi-Processing (OpenMP), https://www.openmp.org/