Report Template: Basic operations with programming threads

Student Name: Filipovych Marharyta

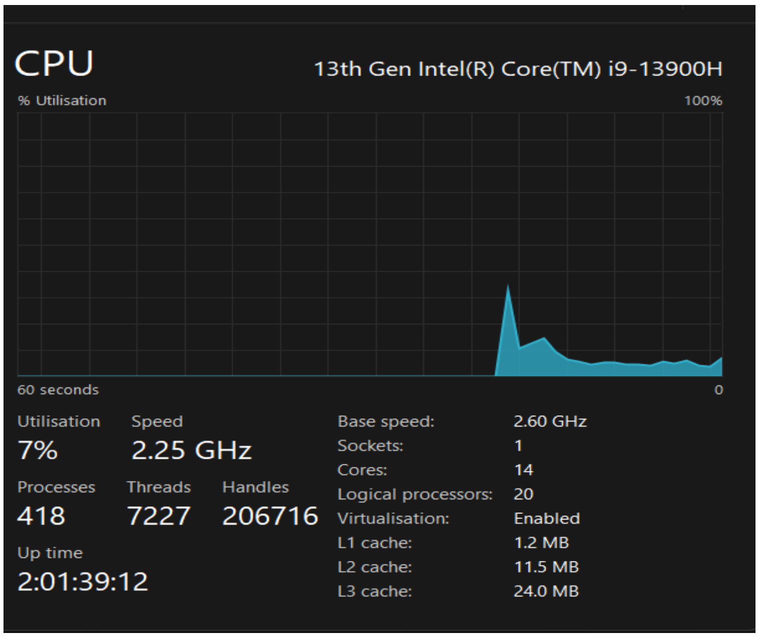
Group: ПКСП 1

Chosen Variant: 10

Github Link: https://github.com/VY-Assignments/parallel-assignment-1-MarharytaFilipovych

Task: Fill the square matrix with random numbers. Place the maximum column element on the main diagonal. Solve the problem using parallelization and without it. Measure the time to solve the problem. Analyze the results.

PC Specifications:



13th Gen Intel(R) Core(TM) i9-13900H

Utilisation	7%
Speed	2.25 GHz
Processes	418
Threads	7227
Handles	206716
Up time	2:01:39:12
Base speed:	2.60 GHz
Sockets:	1
Cores:	14
Logical processors:	20
Virtualisation:	Enabled
L1 cache:	1.2 MB
L2 cache:	11.5 MB
L3 cache:	24.0 MB

13th Gen Intel(R) Core(TM) i9-13900H 2.60 GHz

Installed RAM: 16.0 GB (15.6 GB usable)

System type: 64-bit operating system, x64-based processor

Total Cores: 14

- Performance cores: 6
- efficient cores: 8

Total Threads: 20

The main characteristics of a PC that affect the efficiency of performing parallel calculations:

- core efficiency
- numbers of cores
- clock speed
- memory (capacity, speed)
- hyperthreading
- operating system

Execution time calculation mechanism:

I used the std::chrono library to measure the execution time. I placed the clock after generating the matrix with random values and just before starting the main problem-solving process using threads. The measurement ended after all threads completed their tasks and were joined back into the main thread. I decided to measure the time in milliseconds.

Problem Solved without Parallelization:

I created a function called findMaxAndChange. Inside, I sequentially iterate over all columns of the matrix. For each column, I find the maximum element by iterating over all row values in that column. The maximum element is the element in the first row (0 index). Then, I go to the next row and compare the next value. If this value is bigger, I update the maximum element. After processing all row values in that column, I assign this maximum element to the diagonal place in this column (matrix[c][c]). I do it right away to save time. Then, I go to the next column and repeat all processing.

Time complexity: O(n^2)

When the data size was relatively small (50, 100, 500, even 1000), it took only 0-1 milliseconds to execute.

However, with the data sizes 5000 and 1000, executing was significantly longer – 154.7 and 865.3, respectively.

Problem Solved with Parallelization:

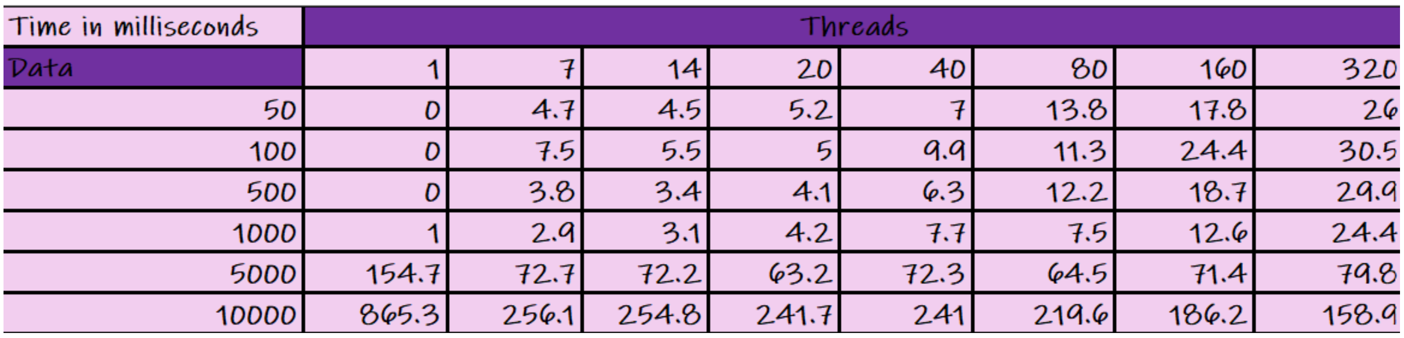
I divided matrix columns into equal-sized chunks based on the number of threads. Each thread was responsible for processing a range of columns, where it should calculate the maximum element and update the matrix.

How do I perform the division? I decided to compute the number of columns per thread based on the N of data and the number of threads. I divide the number of data N by the number of threads. The last thread gets the remainder of the columns. However, there are some issues: if the number of threads is bigger than the number of data (for instance, N=50 and THREADS=320), the division results in 0 (50/320=0) columns per every thread except the last thread, which has to proceed all the columns due to the condition specified later (size_t endColumn = (i == THREADS - 1) ? N : startColumn + columnsToProcess;). Hence, it is better not to use large numbers of threads to calculate a small amount of data. I thought about handling such cases separately, for example, by equally distributing data between the first N threads, but that way, I would process fewer threads, at most N. Hence, I would not be able to truly see such cases as N=100 and TREADS=160.

Choice of Parallelization Algorithm:

Columns are processed independently of each other. I chose a chunk-based approach.

Effect of Thread Count on Performance:



Time in milliseconds	Threads							
Data	1	7	14	20	40	80	160	320
50	0	4.7	4.5	5.2	7	13.8	17.8	26
100	0	7.5	5.5	5	9.9	11.3	24.4	30.5
500	0	3.8	3.4	4.1	6.3	12.2	18.7	29.9
1000	1	2.9	3.1	4.2	7.7	7.5	12.6	24.4
5000	154.7	72.7	72.2	63.2	72.3	64.5	71.4	79.8
10000	865.3	256.1	254.8	241.7	241	219.6	186.2	158.9

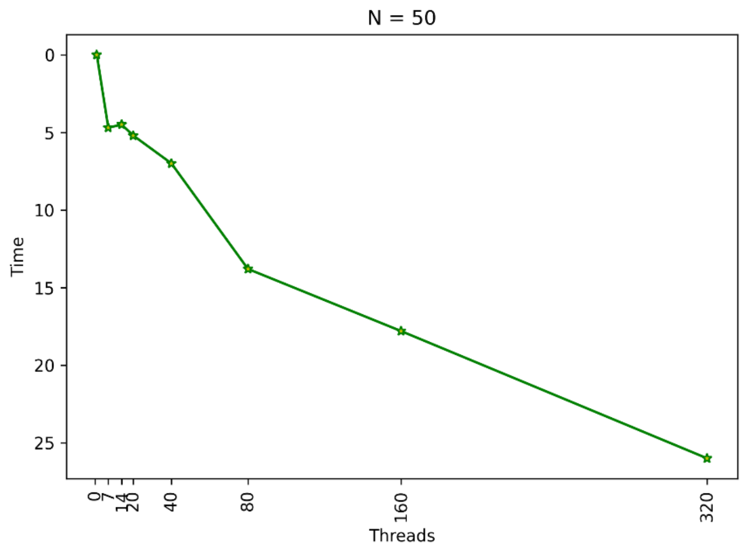
Small Data Sizes (N = 50, 100, 500, 1000)

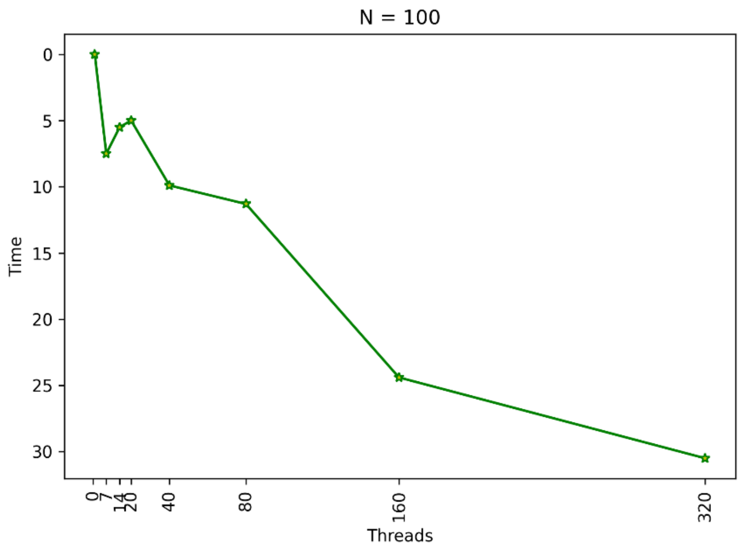
The program performs best for smaller datasets with a small number of threads or even a single-threaded implementation.

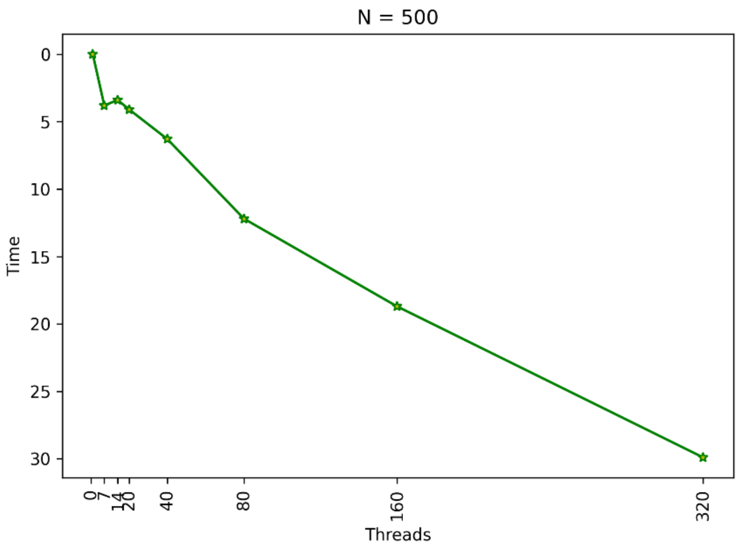
- N = 50, 100, 500, 1000 THREADS = 1: Execution time is **0, 0, 0, 1 ms respectively**.
- N = 50, 100, 500, 1000, THREADS = 7: Execution time is **4.7, 7.5, 3.8, 2.9 ms respectively.**
- N = 50, 100, 500, 1000, THREADS = 14: Execution time is **4.5, 5.5, 3.4, 3.1 ms respectively.**

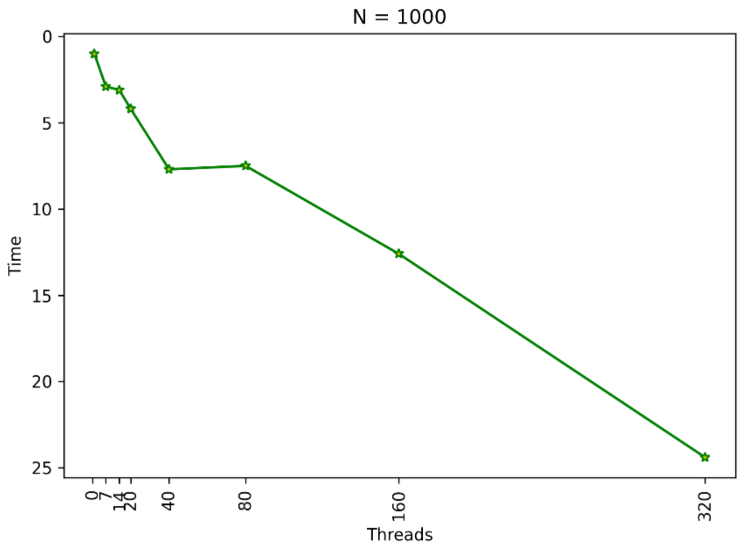
Increasing the thread count to higher values, such as 40, 80, 160, or 320, causes the program to slow down due to:

- thread time management overhead: creation and destruction of threads takes time.
- insufficient workload per thread. Especially when the data size is smaller than the thread count (100 < 160), the last thread has to process all data, but all other threads process 0 data.









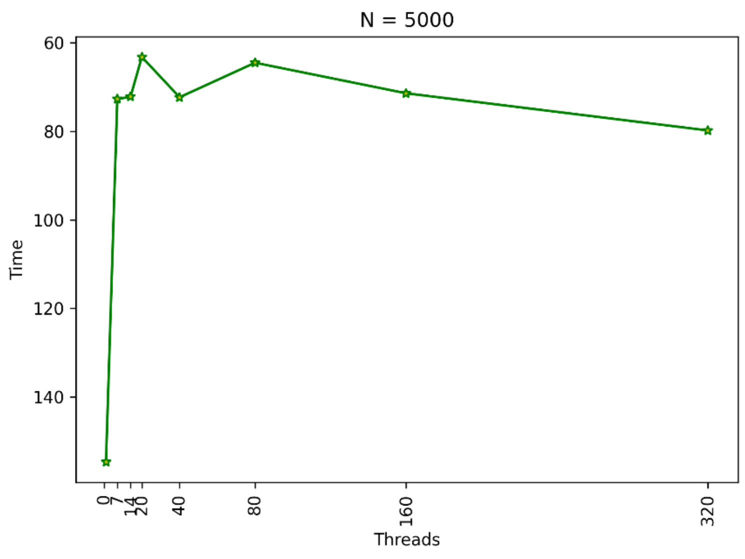
Larger Data Sizes (N = 5000, 10000)

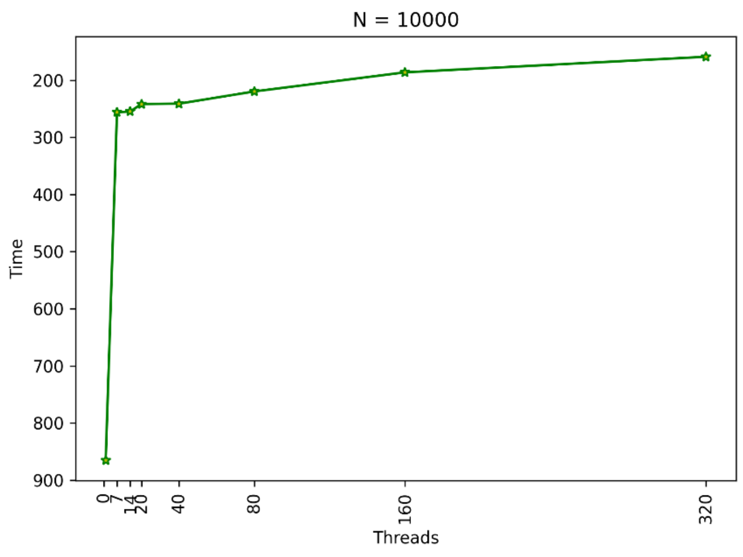
For larger datasets, parallelization becomes effective due to the significant workload that can be evenly distributed across threads.

- N = 5000, 10000 THREADS = 1: 154.7 and 865.3 ms respectively.
- N = 5000, 10000 THREADS = 7: execution time drops to 72.7 and 256.1 ms, respectively.
- N = 5000, 10000 THREADS = 20: execution time further reduces to 63.2 and 241.7 ms, respectively.
- N = 5000, 10000 THREADS = 40: execution time is 72.3 and 241 ms, respectively.

As the thread count increases, execution time grows gently for N=5000 but improves for N=10000 (maybe due to the 'equal' distribution of columns between threads).

- N = 5000, THREADS = 320: 79.8 ms.
- For N = 10000, THREADS = 320: threads: 158.9 ms.





Effect of Data Size on Performance:

The square matrix has of size N x N. As N increases, the number of elements to be processed grows twice. The total amount of elements to process is N^2.

The larger the matrix, the more time needed to complete the task, especially when only one thread exists. I noticed it when N was 5000 and 10000.

It is worth mentioning that the range of values inside the matrix generated at the beginning of the program also matters. If it is smaller (1-100), the execution time is faster. In the experiments detailed in this document, a range of 1-1000 was used, which led to longer execution times.

Conclusion:

- For small matrices, single-threaded execution may perform best because thread management's overhead outweighs parallelism's benefits.

- For larger matrices, parallelism becomes highly beneficial because the workload is distributed across multiple threads. The execution time reduces noticeably when the workload is divided effectively.
- The optimal number of threads depends on the matrix size and available CPU cores. Based on my PC, this number lies between 14 and 20. I would choose the number of 20. This will utilize hyperthreading and seems reasonable for most cases based on graphs and table data.