Report Template: Basic operations with programming threads

Student Name: Olesia Mykhailyshyn
Group: 1
Chosen Variant: 10
Github Link: https://github.com/VY-Assignments/parallel-assignment-1-olesia-mykhailyshyn.git

Task

Analyze the speed of work execution without using threads, with them, with different numbers of them, and understand what the optimal number of threads is, as well as what affects the performance of the program.

The analysis was carried out by implementing this task:
Fill the square matrix with random numbers. Place the maximum column element on the main diagonal.

Different matrix dimensions, the number of kernels, and the number of trades were taken into account.

PC Specifications

The following characteristics affect the efficiency and speed of parallel programs:

- Number of processor cores - The Intel Core i5-1240P processor has 12 physical cores, 4 of which are performance cores and support Hyper-Threading technology, which allows you to use up to 16 logical threads for parallel execution of tasks.
- RAM. The type and speed of RAM affect performance in tasks that require intensive work with data.

Execution time calculation mechanism

I used the std::chrono library to measure and calculate the program execution time. The std::chrono provides us with three clocks with varying accuracy. The high_resolution_clock is the most accurate and hence it is used to measure execution time.

I started measuring the time after I created the matrix and filled it with random numbers, since these are time and energy-consuming processes, and in this case we only needed to get the execution time of the task itself, namely, to change the diagonal elements of the matrix.

In the first part of the task, I calculated the time for each matrix size and then got the average of 5 calculations. And then, in the second part of the task, I fixed the size of the matrix and made 5 measurements for each other number of trades and got the average.

Problem Solved without Parallelization

```
5 0.000001
50 0.000016
100 0.000052
500 0.001635
1000 0.007273
10000 0.967362
```

Column 1 represents the dimension of the matrix, and column 5 represents the average time it takes to complete a task for that dimension.
The approach to this task is to create a matrix, fill it with random values, determine the maximum elements in each column and write them to the maximum value vector, replace the diagonal elements of the matrix with the values from the maximum value vector.

Problem Solved with Parallelization

The approach to this stage was similar to the previous one, but now I used std::thread and thread.join to divide the work on a large data set into several parts. I used this approach in the methods to determine the largest value in each column of the matrix, and then to change the diagonal element for each column of the matrix.

Choice of Parallelization Algorithm

For my task, the best option was to divide the matrix by columns and then work with each column separately, first looking for the maximum value in it, and then changing the diagonal element of the column to this maximum value. This is the best option because it allows you to work with each column independently.

Effect of Thread Count on Performance

Processor Architecture of Intel Core i5-1240P

Types of Cores:

1. **Performance Cores (P-cores)**:
   - 4 physical cores that support Hyper-Threading, meaning each core can process 2 threads simultaneously.
     - Total: 4 P-cores×2=8 logical cores
2. **Efficiency Cores (E-cores)**:
   - 8 physical cores that do not support Hyper-Threading. Each core processes only **1 thread simultaneously**.
     - Total: 8 E-cores×1=8 logical cores

Total Logical Cores:

- 8 (from P-cores)+8 (from E-cores)=16 logical cores
- This means your processor can process 16 threads simultaneously without context switching.

What Does `std::thread::hardware_concurrency()` Return?

- The function `std::thread::hardware_concurrency()` returns the number of logical cores available to the operating system for thread execution.
- In your case, it will return 16, as your processor has 16 logical cores (8 from P-cores and 8 from E-cores).

Why Can 32 Threads Perform Faster than 16?

1. **Idle Time Due to Memory Delays**:
    - Logical cores may be idle while waiting for data from memory, cache, or I/O operations.
    - By using 32 threads, other threads can perform work while some threads are waiting for memory operations to complete, reducing the overall idle time of the processor.
2. **Hyper-Threading**:
    - Each P-core can process 2 threads simultaneously using Hyper-Threading. With 32 threads, the first 16 threads will always be "busy," and the operating system will schedule the remaining 16 threads to execute efficiently.
3. **Context Switching**:
    - When the number of threads exceeds the number of logical cores (e.g., 32 > 16), the operating system schedules their execution through context switching.
    - While context switching introduces some overhead, it is often outweighed by the reduced idle time of the processor.
4. **Load Balancing Across Cores**:
    - The operating system distributes 32 threads evenly across 16 logical cores, ensuring the processor is fully utilized.
5. **Optimal Load for Large Tasks**:
    - For highly parallelizable tasks (e.g., processing large matrices or independent data blocks), using more threads ensures better utilization of the processor.

Why Are Numbers That Are Powers of Two (2, 4, 8, 16, 32) Important?

1. **Cache Efficiency**:
    - Numbers that are powers of two align well with the cache structure of the processor (e.g., cache line size is usually 64 bytes).
    - Evenly distributing data across threads minimizes cache misses.
2. **Task Distribution**:
    - If the number of threads is a power of two, tasks can be evenly divided into equal parts.
    - For example, a matrix with 32 columns can be distributed evenly across 32 threads (1 column per thread).
3. **Hardware Optimization**:
    - Modern processor architectures (especially cache and memory controllers) are optimized for numbers that are powers of two.

How to Determine the Optimal Number of Threads?

Programmatically Determine Available Logical Cores:

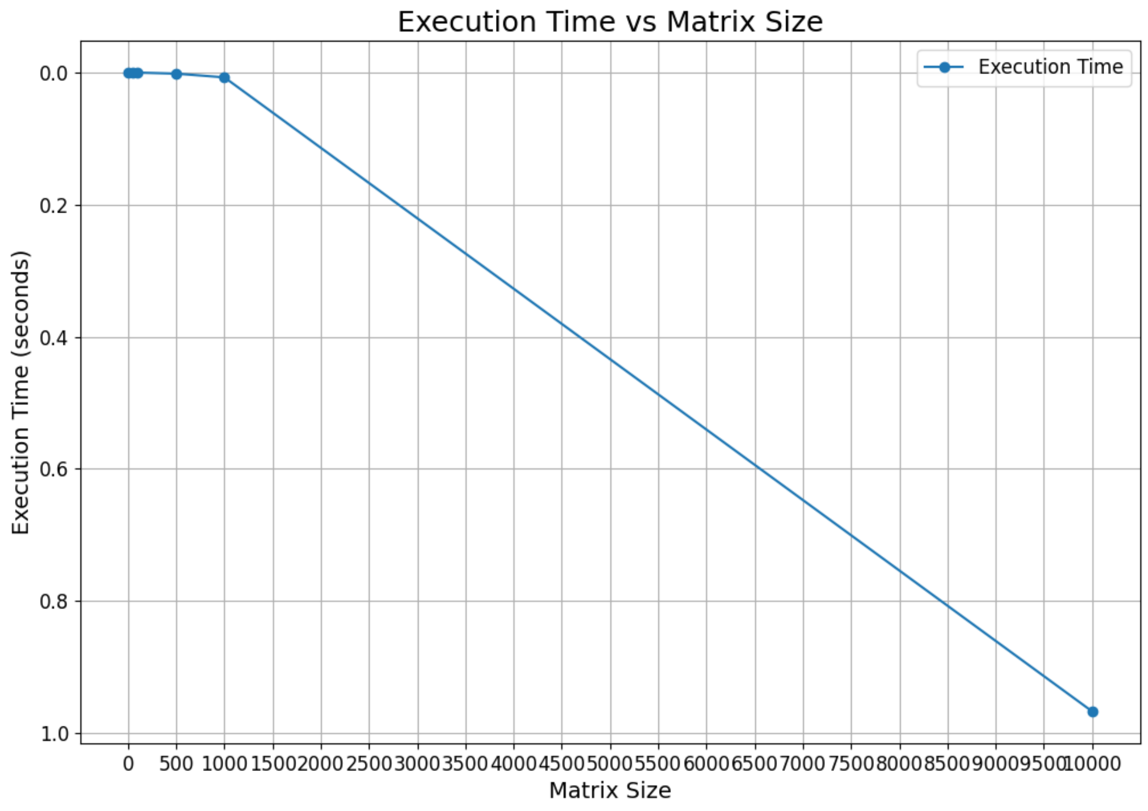
```
#include <thread>
#include <iostream>

int main() {
    unsigned int threads = std::thread::hardware_concurrency();
    std::cout << "Available logical cores: " << threads << std::endl;
    return 0;
}
```

Effect of Data Size on Performance

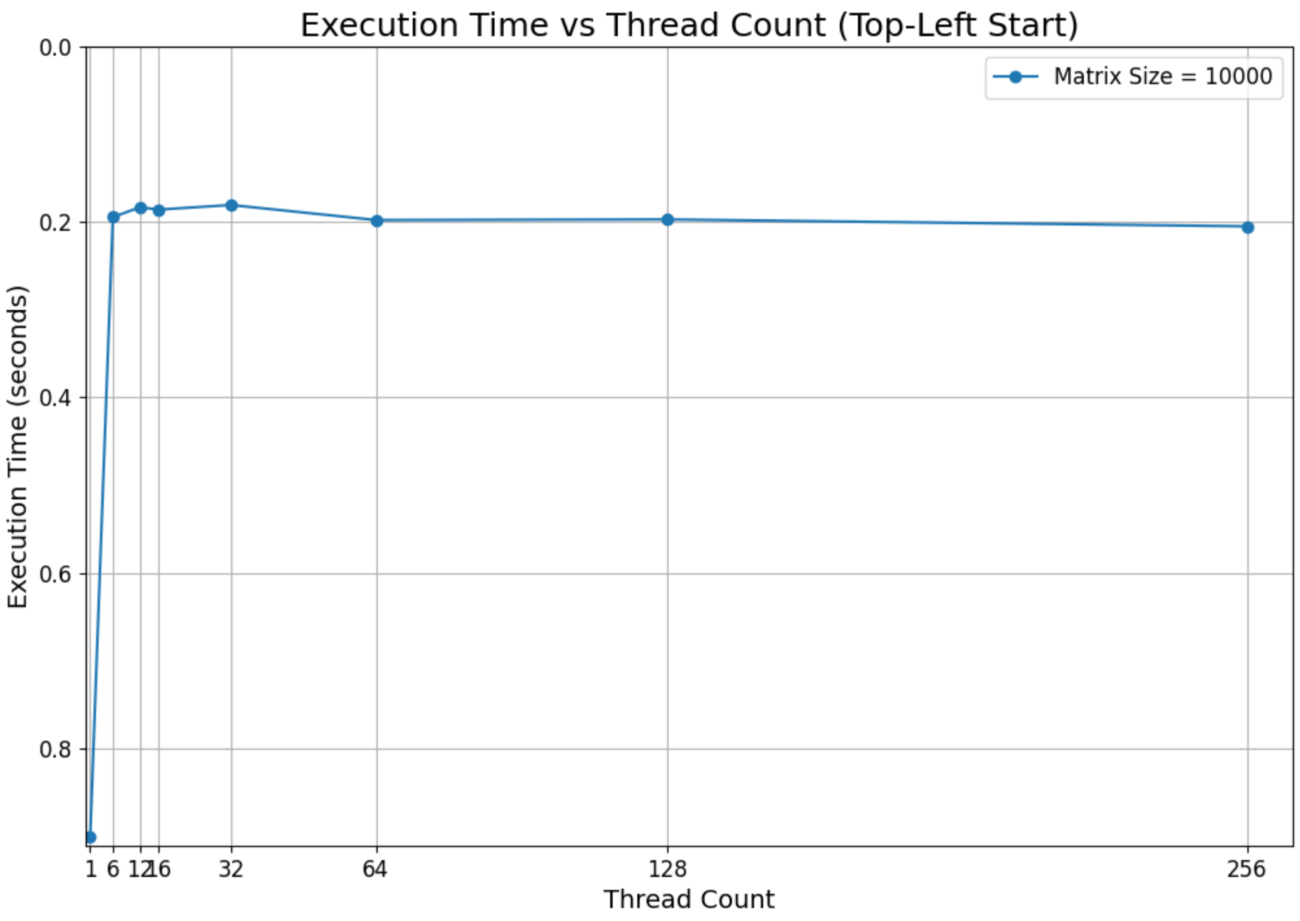
As the matrix size increases, multithreading becomes more efficient until performance is limited by memory access. The optimal number of threads depends on the size of the problem, and performance can be improved by optimizing cache utilization and evenly distributing work. If you create a matrix using an array, for example, and not a vector, then an overflow stack will occur.

- Small matrices: Multithreading may not significantly reduce execution time due to threading overhead.
- Moderate matrices: Multithreading provides substantial performance gains as computation outweighs overhead.
- Large matrices: Performance becomes limited by memory access patterns and bandwidth, requiring optimization techniques like blocking and cache-efficient algorithms.
- Scalability: Both strong and weak scaling depend on task size, cache utilization, and efficient thread management.

Conclusion



Sequential visualisation



Parallel visualisation