Course : Operating Systems

Assignment 01

Report: Parallel Matrix Operations with Threads

Members	ID
Quentin Lauriot	40304712
Panying Song	40304754
Sarah Nguyen	40304753

Concordia University & Efrei

Table of content

I. Code Implementation

• Introduction

We decided to code in C++ instead of Java in order to challenge ourselves more and learn how to use this language.

• The beginning

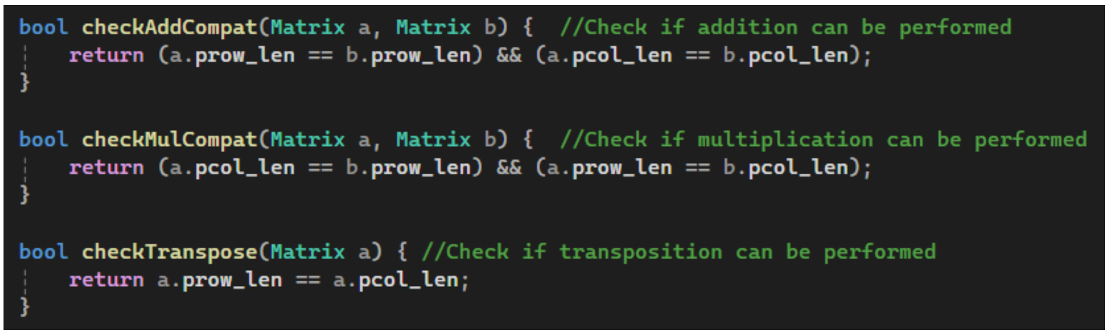
For each operation: addition, multiplication, and transposition, we start by coding it without using threads. It's only after having the algorithm for each operation that we try to use multiple threads.

• Using multiple threads

○ Input Validation

Before performing any operations, we validate the input matrices' compatibility:

- For addition, both matrices must have the same dimensions in terms of rows and columns.
- For multiplication, the number of columns in the first matrix must match the number of rows in the second matrix, and vice versa.
- For transposition, the matrix must be square.



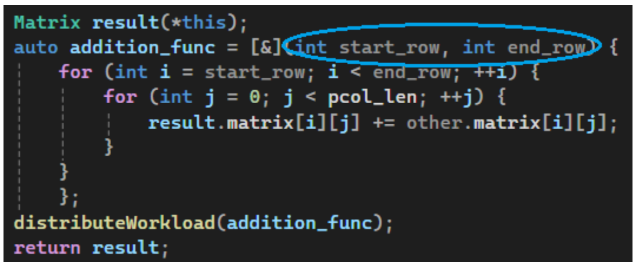
```cpp
bool checkAddCompat(Matrix a, Matrix b) {  //Check if addition can be performed
    return (a.prow_len == b.prow_len) && (a.pcol_len == b.pcol_len);
}

bool checkMulCompat(Matrix a, Matrix b) {  //Check if multiplication can be performed
    return (a.pcol_len == b.prow_len) && (a.prow_len == b.pcol_len);
}

bool checkTranspose(Matrix a) { //Check if transposition can be performed
    return a.prow_len == a.pcol_len;
}
```

○ Work Distribution

To concurrently perform operations on one or two matrices using multiple threads, we initiate a lambda function representing the operation to be performed.

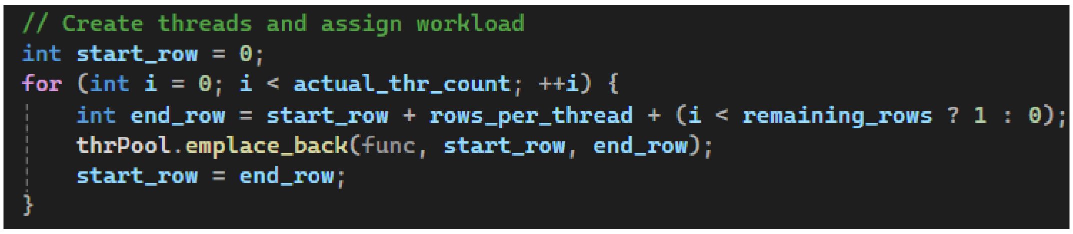


```cpp
Matrix result(*this);
auto addition_func = [&](int start_row, int end_row) {
    for (int i = start_row; i < end_row; ++i) {
        for (int j = 0; j < pcol_len; ++j) {
            result.matrix[i][j] += other.matrix[i][j];
        }
    }
    };
distributeWorkload(addition_func);
return result;
```

This function takes two parameters: 'start_row' and 'end_row', specifying the range of rows to be processed by the current thread.

○ Distribute Workload

For each operation, we distribute the workload among multiple threads by evenly dividing the matrix rows among the available threads.



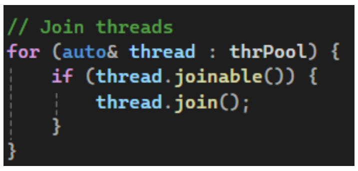
```cpp
// Create threads and assign workload
int start_row = 0;
for (int i = 0; i < actual_thr_count; ++i) {
    int end_row = start_row + rows_per_thread + (i < remaining_rows ? 1 : 0);
    thrPool.emplace_back(func, start_row, end_row);
    start_row = end_row;
}
```

○ Thread Execution

Each thread executes the operation function, iterating over the specified range of rows and adding corresponding elements from both matrices, thereby updating the corresponding elements in the 'result' matrix.

○ Join Threads

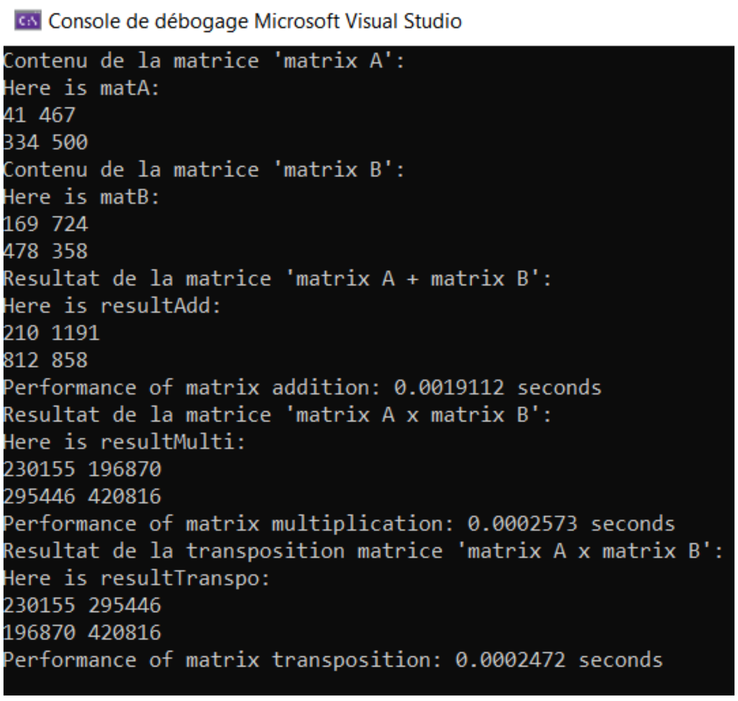
After all threads have completed their tasks, the main thread joins them to synchronize their execution.



```cpp
// Join threads
for (auto& thread : thrPool) {
    if (thread.joinable()) {
        thread.join();
    }
}
```

○ Return Result

Finally, the method returns the `result` matrix, containing the final outcome.



```
Contenu de la matrice 'matrix A':
Here is matA:
41 467
334 500
Contenu de la matrice 'matrix B':
Here is matB:
169 724
478 358
Resultat de la matrice 'matrix A + matrix B':
Here is resultAdd:
210 1191
812 858
Performance of matrix addition: 0.0019112 seconds
Resultat de la matrice 'matrix A x matrix B':
Here is resultMulti:
230155 196870
295446 420816
Performance of matrix multiplication: 0.0002573 seconds
Resultat de la transposition matrice 'matrix A x matrix B':
Here is resultTranspo:
230155 295446
196870 420816
Performance of matrix transposition: 0.0002472 seconds
```

I. Performance Evaluation

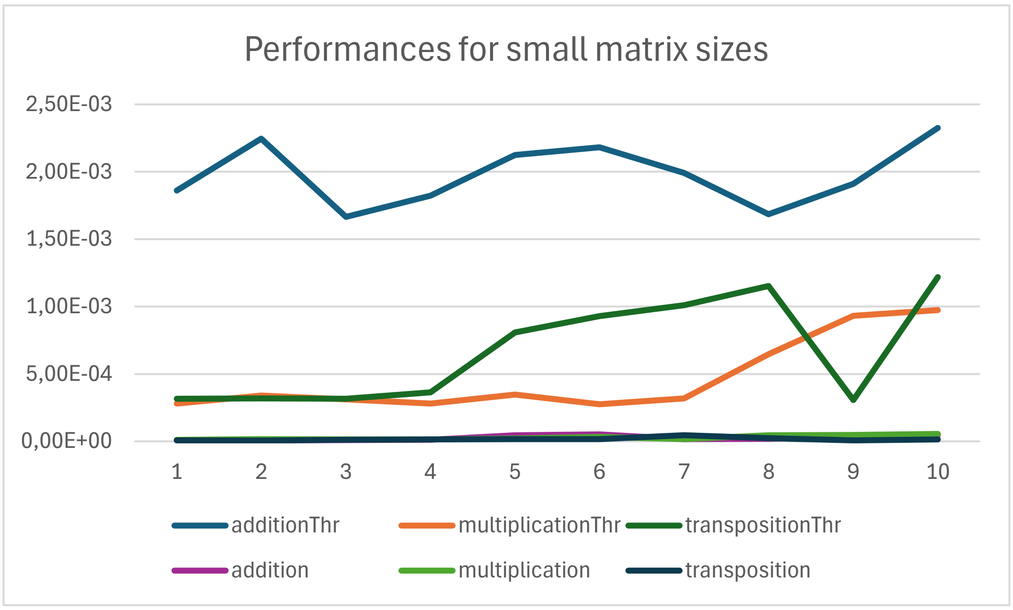
- **Small Matrix Sizes with one thread**

size:	1	2	3	4	5	6	7	8	9	10
additionThr	1,86E-03	2,25E-03	1,67E-03	1,82E-03	2,13E-03	2,18E-03	1,99E-03	1,68E-03	1,91E-03	2,33E-03
multiplicationThr	2,79E-04	3,39E-04	3,12E-04	2,81E-04	3,48E-04	2,75E-04	3,18E-04	6,45E-04	9,32E-04	9,74E-04
transpositionThr	3,16E-04	3,19E-04	3,17E-04	3,63E-04	8,09E-04	9,29E-04	1,01E-03	1,15E-03	3,07E-04	1,22E-03
addition	8,60E-06	7,40E-06	9,60E-06	1,15E-05	4,50E-05	5,16E-05	1,73E-05	1,85E-05	3,52E-05	4,40E-05
multiplication	1,13E-05	1,56E-05	1,42E-05	1,35E-05	2,07E-05	3,31E-05	1,63E-05	4,60E-05	4,64E-05	5,50E-05
transposition	8,20E-06	5,80E-06	1,18E-05	1,37E-05	1,69E-05	1,61E-05	4,55E-05	2,35E-05	7,70E-06	1,37E-05

For small matrix sizes (1 to 10), the threaded versions of addition, multiplication, and transposition are significantly slower than their non-threaded counterparts. This is likely due to the overhead of creating and managing threads, which outweighs the benefits of parallelism for small matrices.

The threaded version of addition has the highest performance time among the three threaded operations for small matrix sizes. This is expected, as addition is a simpler operation than multiplication and transposition, and thus has less potential for parallelism.

The threaded version of multiplication has more potential for parallelism than addition and transposition, as each element in the result matrix depends on multiple elements in the input matrices.



- **Larger Matrix Sizes with one thread**

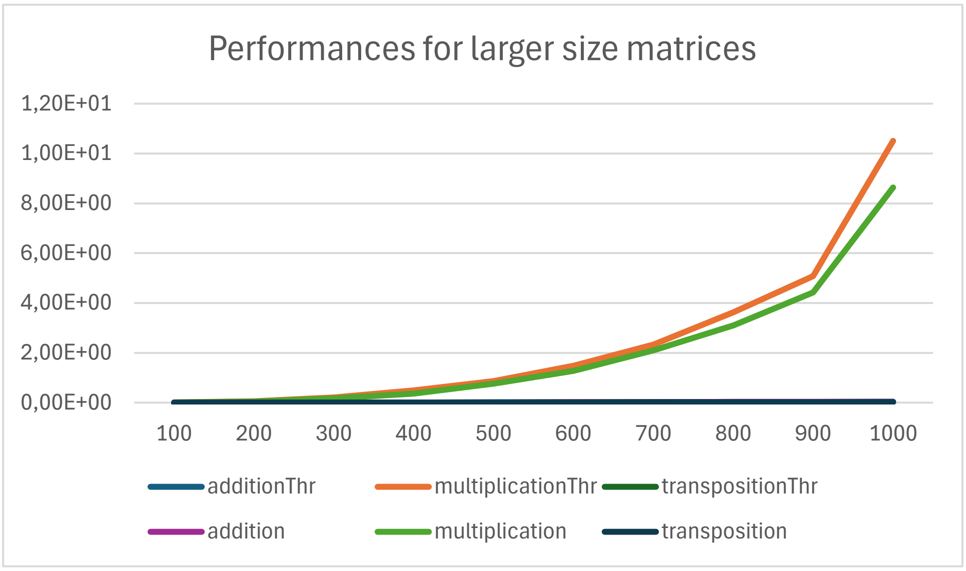
size:	100	200	300	400	500	600	700	800	900	1000
additionThr	3,02E-03	3,69E-03	5,59E-03	7,92E-03	1,27E-02	1,43E-02	1,92E-02	2,39E-02	3,31E-02	3,50E-02
multiplicationThr	1,00E-02	5,43E-02	2,09E-01	4,86E-01	8,64E-01	1,48E+00	2,33E+00	3,63E+00	5,08E+00	1,05E+01
transpositionThr	5,90E-04	1,40E-03	2,41E-03	5,49E-03	7,56E-03	1,03E-02	1,41E-02	1,49E-02	2,11E-02	2,52E-02
addition	6,09E-04	1,25E-03	3,45E-03	6,82E-03	9,62E-03	1,20E-02	1,84E-02	2,64E-02	2,52E-02	3,31E-02
multiplication	7,45E-03	4,67E-02	1,66E-01	3,62E-01	7,64E-01	1,27E+00	2,10E+00	3,09E+00	4,42E+00	8,64E+00
transposition	3,85E-04	1,58E-03	2,95E-03	3,34E-03	7,18E-03	9,72E-03	1,42E-02	1,89E-02	1,93E-02	2,16E-02

For larger matrix sizes (100 to 1000), the threaded versions of addition, multiplication, and transposition are on average faster than their non-threaded counterparts. This is likely due to the benefits of parallelism outweighing the overhead of creating and managing threads for larger matrices.

The threaded version of multiplication has the highest performance time among the three threaded operations for larger matrix sizes. This is expected, as multiplication is a more complex operation than addition and transposition, and thus has more potential for parallelism.

The non-threaded version of multiplication has the highest performance time among the three non-threaded operations for larger matrix sizes. This is expected, as multiplication is a more complex operation than addition and transposition, and thus has more potential for parallelism.

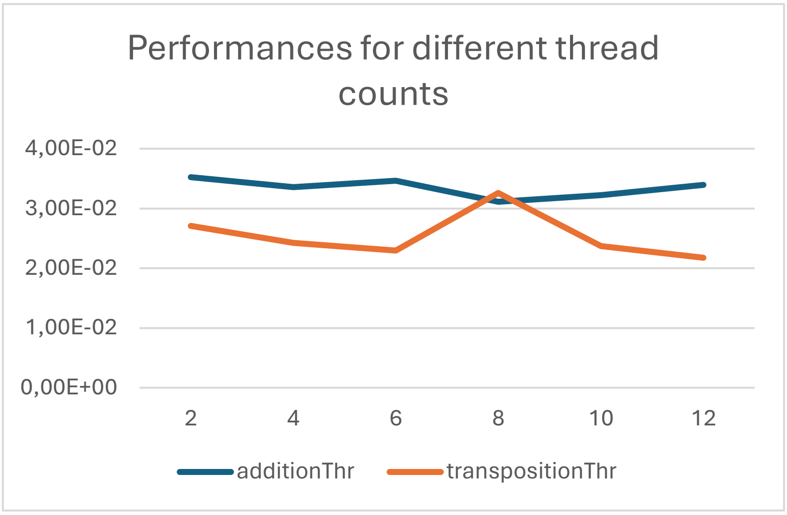
Overall, when using only one thread on larger matrix it’s very similar.



- **Fixed matrix sizes with multiple threads**

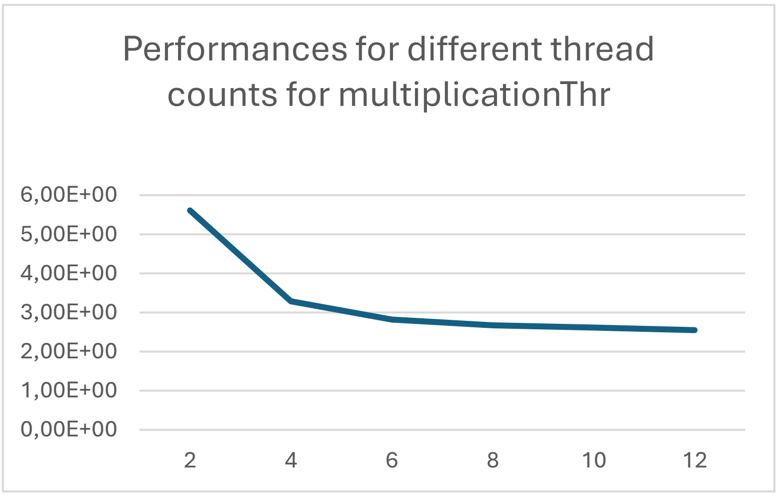
Nb threads	2	4	6	8	10	12
additionThr	3,52E-02	3,36E-02	3,47E-02	3,11E-02	3,22E-02	3,40E-02
multiplicationThr	5,61E+00	3,28E+00	2,82E+00	2,67E+00	2,61E+00	2,55E+00
transpositionThr	2,71E-02	2,43E-02	2,30E-02	3,26E-02	2,37E-02	2,18E-02

For a matrix of size 1000, the performance times for addition and transposition decrease as the number of threads increases from 2 to 6, but then increase slightly for 8 and 10 threads. This suggests that there is a point of diminishing returns for adding more threads to these operations, beyond which the overhead of creating and managing threads outweighs the benefits of parallelism.



For the multiplication, it has the highest performance time among the three threaded operations for all thread counts. This is expected, as multiplication is a more complex operation than addition and transposition, and thus has more potential for parallelism.

Its performance time also decreases as the number of threads increases. This suggests that multiplication can benefit from a larger number of threads.



- ## Conclusion

Overall, it appears that using threads for matrix operations on small matrices is not beneficial and may result in slower performance due to the overhead of creating and managing threads.

However, using threads for matrix operations on larger matrices is beneficial and can result in faster performance times, but the degree of improvement varies depending on the operation and the number of threads used. Addition and transposition have less potential for parallelism than multiplication, and thus have a point of diminishing returns for adding more threads. Multiplication, on the other hand, can benefit from a larger number of threads due to its greater potential for parallelism.