$$\begin{bmatrix} 2 & 3 & 4 \\ 0 & 8 & 1 \end{bmatrix} \xrightarrow{R_1+R_2\to R_2} \begin{bmatrix} 2 & 3 & 4 \\ 2 & 11 & 5 \end{bmatrix}$$

Report on assignment 3

Roetta Marco – Mat. 876884

Christian Bernabe Cabrera – Mat. 843382

Francesco Busolin – Mat. 851884

Project Overview

The project is composed by the files provided as Assignment 2. We modified those files to implement asynchronous and parallel computations in sum and multiplications of matrices.

The main editing has been made on the *operations.h* file, while the new tests are implemented in the *example4.cc* file.

How to use

On a Linux computer, extract the provided ".zip" file into an empty folder, open a terminal and execute the *compile.sh* script.

One file will be created, called *es4*. Execute this file on a terminal and follow the text instructions to select and run the tests.

On a Windows computer, extract the provided ".zip" file into an empty folder and open the VS Project file to compile and run the program.

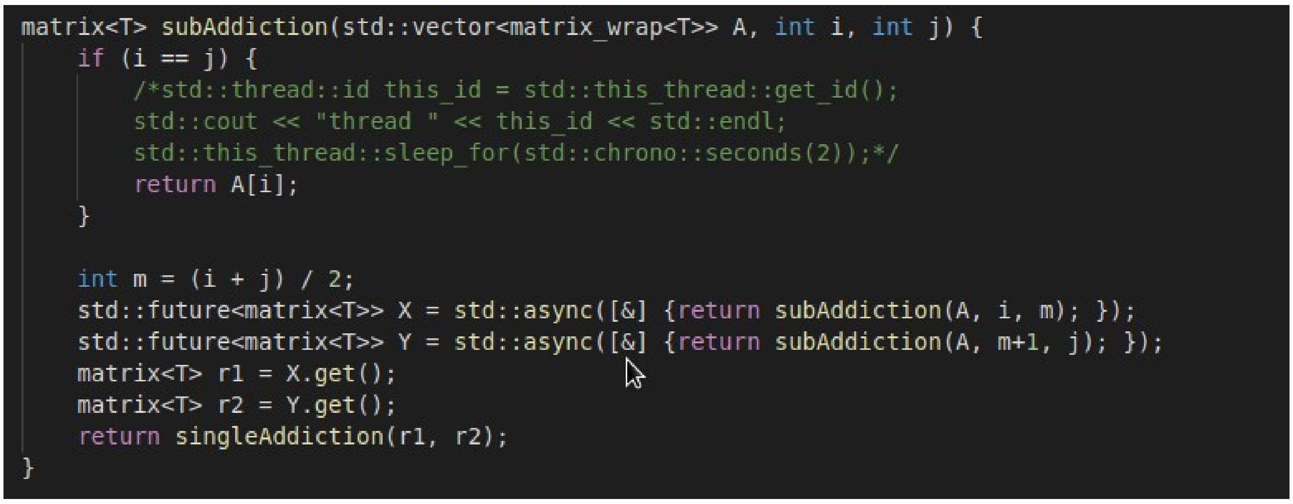
Matrix Sum

Asynchronous processing has been implemented to allow the parallel executions of sums of matrices, both in compile-time size mode and run-time size mode.

For example, we want to perform the sum (A+B+C+D) where A, B, C, D are all matrices of the same size.

Using a divide and conquer algorithm we split the computation in A+B and C+D and compute them in parallel using dedicated threads.

This means that we create a list of matrices to be added together through the overloading of + operators and use a future object connected to an async operation for every matrix couple in the list.



```
matrix<T> subAddiction(std::vector<matrix_wrap<T>> A, int i, int j) {
    if (i == j) {
        /*std::thread::id this_id = std::this_thread::get_id();
        std::cout << "thread " << this_id << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(2));*/
        return A[i];
    }

    int m = (i + j) / 2;
    std::future<matrix<T>> X = std::async([&] {return subAddiction(A, i, m); });
    std::future<matrix<T>> Y = std::async([&] {return subAddiction(A, m+1, j); });
    matrix<T> r1 = X.get();
    matrix<T> r2 = Y.get();
    return singleAddiction(r1, r2);
}
```

We do this while calling recursively the function for every couple of matrices available until the end of the list.

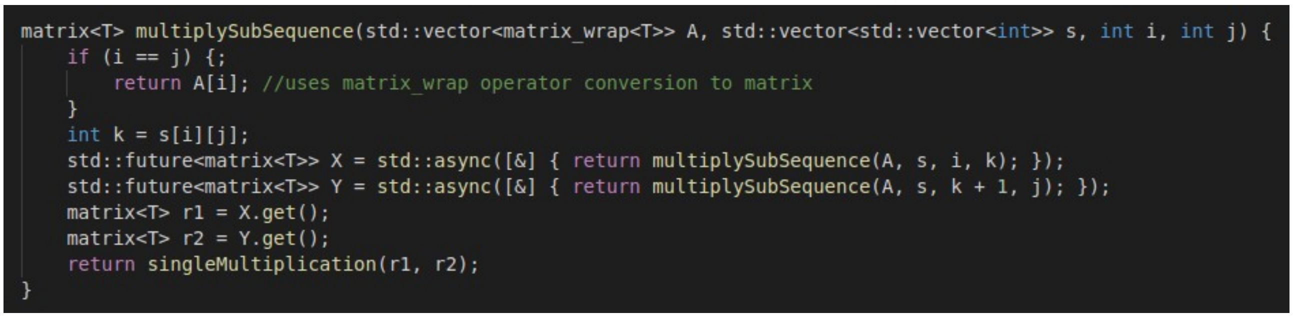
To get the results at every recursive call we use the get() method available from the std::future objects.

Inside the singleAddiction() function, we compute the sum of two matrices using multi-threading as described in the parallel processing chapter.

Matrix Product

As for the sums, when we perform operations like (A*B*C*D) on matrices, we want to calculate in parallel the matrices and maintain the multiplication order as provided by the matrix chain multiplication algorithm developed for the assignment 2.

To do so, we used our algorithm developed for the Assignment 2 and modified it by adding the std::future objects for asynchronous computation combined with a recursive method as visible in the code below:

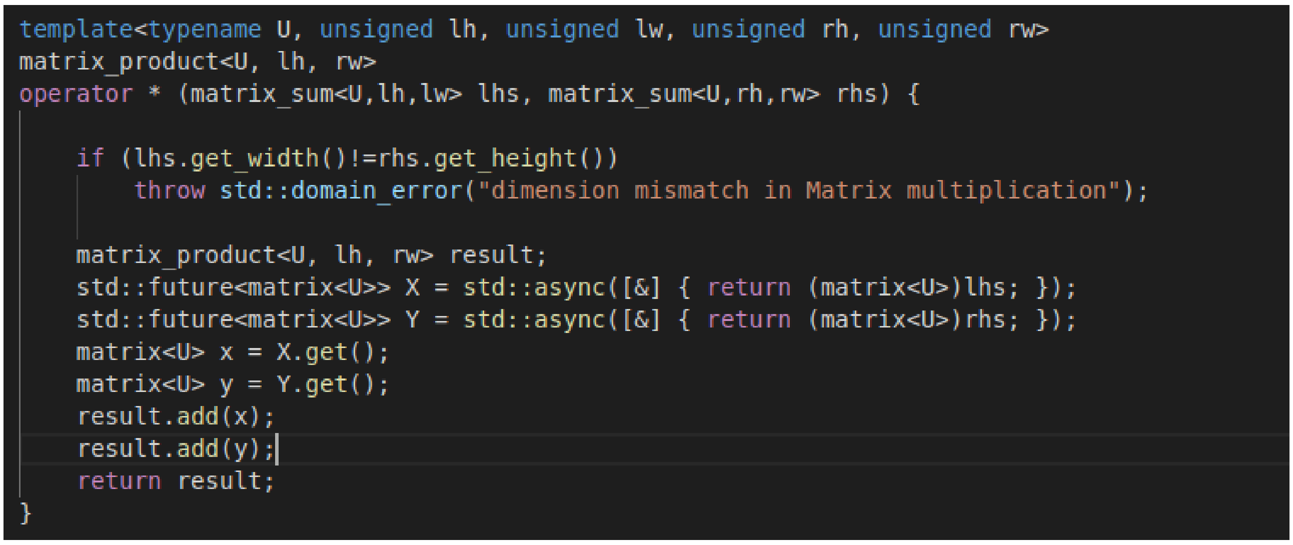


```cpp
matrix<T> multiplySubSequence(std::vector<matrix_wrap<T>> A, std::vector<std::vector<int>> s, int i, int j) {
    if (i == j) {;
        return A[i]; //uses matrix_wrap operator conversion to matrix
    }
    int k = s[i][j];
    std::future<matrix<T>> X = std::async([&] { return multiplySubSequence(A, s, i, k); });
    std::future<matrix<T>> Y = std::async([&] { return multiplySubSequence(A, s, k + 1, j); });
    matrix<T> r1 = X.get();
    matrix<T> r2 = Y.get();
    return singleMultiplication(r1, r2);
}
```

The recursive behavior is exactly the same as described in the report of the previous assignment.

Handling combined results

To allow operations in the form of (A+B+...)*(C+D+...) we added a new operator * override in the operations file.



```cpp
template<typename U, unsigned lh, unsigned lw, unsigned rh, unsigned rw>
matrix_product<U, lh, rw>
operator * (matrix_sum<U,lh,lw> lhs, matrix_sum<U,rh,rw> rhs) {

    if (lhs.get_width()!=rhs.get_height())
        throw std::domain_error("dimension mismatch in Matrix multiplication");

    matrix_product<U, lh, rw> result;
    std::future<matrix<U>> X = std::async([&] { return (matrix<U>)lhs; });
    std::future<matrix<U>> Y = std::async([&] { return (matrix<U>)rhs; });
    matrix<U> x = X.get();
    matrix<U> y = Y.get();
    result.add(x);
    result.add(y);
    return result;
}
```

This method allow both the operation itself and the async handling of the two sum parts in an optimal way. The technique used is the same as before, with the addition of the *result* handling where we call the matrix multiplication class to perform the final multiplication.

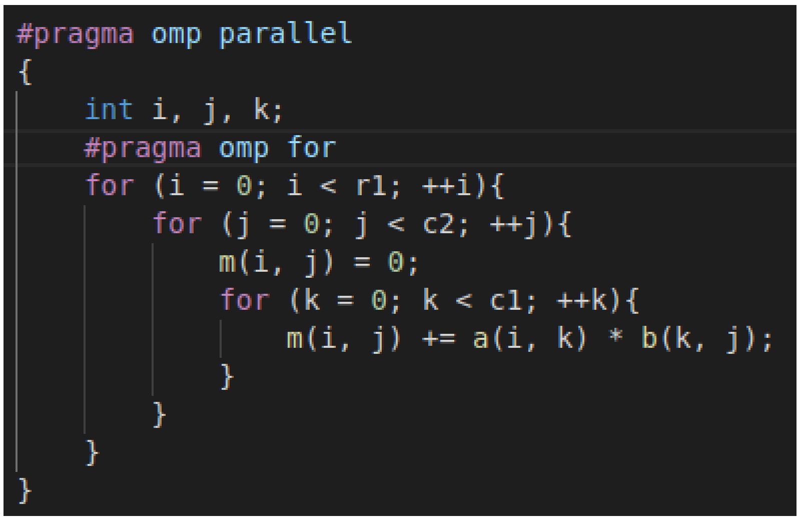
Design Decisions for parallel processing

We implemented the parallel processing of sums and products using the OpenMP framework, available since G++ 4.8.5 and particularly suited for the efficient parallel executions of for cycles, even if they are nested.

OpenMP provide a very simple and easy to read syntax to parallel parts of code without having to declare manually the partitioning of the work and the thread join and synchronization.

We used the standard approach suggested by the OpenMP documentation:

- Initialize a “omp parallel” section with threads private variables;
- Initialize a nested “omp for” section where we define the job to be done in parallel;



```
#pragma omp parallel
{
    int i, j, k;
    #pragma omp for
    for (i = 0; i < r1; ++i){
        for (j = 0; j < c2; ++j){
            m(i, j) = 0;
            for (k = 0; k < c1; ++k){
                m(i, j) += a(i, k) * b(k, j);
            }
        }
    }
}
```

OpenMP will automatically set the correct number of threads and split the load evenly on them.

To allow the linking of the additional libraries needed for OpenMP an additional command must be provided to g++ (and to VS compiler/linker in the project options):

g++ example4.cc ***-fopenmp*** *-o es4 -march=native -O3*

-fopenmp allow the linking of OpenMP;

-march=native allow the use of the vector instructions if the CPU used supports them;

-O3 allow the usage of all available optimization provided by the compiler.

It is also required to include the omp library as we did in the *operations.h* file.

Performance Analysis

To conclude the report we show some performance comparisons between the all possible combinations of executions with and without the use of openMP and/or async.

In order to ensure fairness all the executions were performed on the same machine and with the same matrices and chain of operations,

Here we run the test 6 with 512x512 matrices of integers

Addition	With async	Without async
With openMP	0.0401	0.0234
Without openMP	0.0177	0.0161

Multiplication	With async	Without async
With openMP	0.8178	0.5582
Without openMP	2.9357	3.0447

As we can see, additions are pretty much instantaneous no matter the strategy used, while on multiplication the use of openMP really makes a difference with the asyncs be mostly irrelevant.

The addition also show how the thread creation and join has a small impact on performance when using small matrices. Performing the test with square matrices of size 1024 or 2048 will show an advantage of openMP.