Advanced algorithms and programming methods

Year: 2017/2018

Assignment 3

Concurrency and Parallelism

18th August 2018

Author:
Hibraj Feliks 854342

Index

1 Introduction

1.1 Project overview

Extend the matrix library so that the operations can be performed concurrently.

There are two forms of concurrency to be developed:

- **Concurent operations**: In multiple matrix operations like $(A+B)*(C+D)$ the addition $A+B$ and $C+D$ can be performed asynchronously in any order (or in parallel) before the final multiplication. Same goes for sequence of multiplications like $A*B*C*D$ if the optimized multiplication order happens to be $(A*B)*(C*D)$.
- **Parallel matrix multiplication**: With large matrix multiplication the access times through the polymorphic wrapper can induce an sizable overhead. one way around this is to access big fixed-size submatrices with each call. With this optimization matrix A is composed of several submatrices (A_{ij}), each fetched with a single polymorphic call. Matrix multiplication can be expressed easily in this form, so $C = A*B$ becomes $C_{ij} = \sum_k A_{ik} * B_{kj}$, where now '$*$' denotes the usual matrix multiplication for the submatrices. Now, each C_{ij} can be computed independently from the other and in parallel.

2 Implementation

The design of this implementation includes slight changes from the starting version.
They are discussed in the following subsections.
But first, some overall observations:

Lazy evaluation has been extended to include also addition operation in order to reach more cases of concurrent operations.

Where possible, dimensions of matrices are checked at **compile time**.

Mutexes were needed to protect some critical sections. Given the fact that different threads were accessing the same shared data, to synchronize their actions, mutual exclusiveness has been granted using **std::unique_lock**. It is a wrapper class for mutex, which follows **RAII idiom** (Resource Acquisition is Initialization). Thus, in case of thread failure, the loss of the lock does not cause all other threads to starve waiting for the mutex. Indeed, when the thread holding the lock terminates by launching an exception or simply by failing, the lock is automatically released.

When new threads are launched, they are wrapped in a try-catch statement and exceptions are handled through **Exception Dispatcher Idiom**.

Usage semantics for this implementation does not change from the starting implementation.

2.1 Matrix Addition

The class **Matrix Addition** has been added in order to address concurrent operations in cases in which addition operations can be performed asynchronously, *e.g.* $(A+B)*(C+D)$. This class is similar to matrix product. It has template parameters for the type of matrices content it holds, and for height and width of the resulting matrix.
If composed of static matrices than template parameters are going to carry the static information, otherwise, if one of the matrices is dynamic, the static information of Matrix Addition becomes useless and it's going to be discarded.

Resolve() method tries to optimize the operations inside a matrix addition in case there is a sequence of matrices to sum.

For example the sequence of operations $A + B + C + D$ will lead to a Matrix Addition holding these 4 matrices so when *Resolve()* will be called, it is going to launch a thread to solve $A + B$ and in parallel another thread for $C + D$.
Unlike matrix product, there is no need to decide a better order in which operations can be performed because all matrices must have the same size. For that reason pairs of matrices are picked sequentially and assigned to threads.

There might be cases of **critical section** confilcts, because threads should all access the same list containing matrices to be summed. For that reason there is a **unique lock** which is acquired from the main thread, which upon launching threads for cuncurrent operations assigns them the corresponding matrices. After launching all threads, it releases the lock and waits for all threads to join. All threads after performing the addition operation have to modify the list of matrices, which can cause the conflict, thus they have to wait for the unique lock before performing this delete operation.
Only one thread at a time can have this lock, otherwise concurrent deletions can cause inconsistencies in the list. Including the situation in which first thread launched by main thread completes first, thus invalidating the list, thus making main thread start a new thread on invalid matrices of the so modified list.

2.2 Matrix Product

In order to achieve concurrent multiplication, some changes have been applied to the starting matrix product class.

A new member variable is introduced, **std::list<bool> busy**. This list has the same length of the list of matrices composing the matrix product and it's used to keep track of which matrices are currently busy in some operation. Thus when we call *find_max()* to find the best couple of matrices to multiply, we avoid those which are already occupied in a concurrent operation.
After launching the thread we proceed by searching for the next best couple of matrices to multiply until there are no more addiacent matrices available.

There is a similar structure of **mutex** to protect the **critical section** to the one described for matrix addition in Sec. 2.1. Main thread holds the lock until all threads are launched, when the lock is released, threads go on with their computation. Main thread is responsible for setting *busy* entries to *True* when corresponding matrices are assigned to threads. Main thread also passes references to elements on lists of *matrices*, *sizes* and *busy* for corresponding matrices. Changes to these lists are under critical section, thus protected by the same mutex. A thread may access this section on the beginning, to insert new entry for the result matrix, and after multiplication operation is executed, to delete old matrices entries. The multiplication operation is out of the mutex, of course, otherwise it would block all other thread uselessly.

Sizes member variable has been chosen to be a **list**, differently from the starting implementation, because an insertion/deletion to a vector may cause a resize and thus invalidation of all pointers. Which means that threads having a reference to a certain position of the vector (or an iterator) may point to an invalid (or wrong) position after an alteration caused by another thread to that vector. Same reasoning holds for list *busy*.

The required algorithm for parallel matrix multiplication says that input matrices (say A, B) should be partitioned into square blocks, such that each block holds a fixed size number of elements (say *block_size* $bs = 100$, so $A_{ij} = bs$ x bs). Each block of the resulting matrix (say C_{ij}) can be computed in parallel and requires only submatrices A_{ik} and B_{kj}.
Thus $C_{ij} = \sum_k A_{ik} * B_{kj}$ where now instead of calling the polymorphic wrapper for each entry of A and B several times, we call it on submatrices A_{ij} and B_{ij}.

In Fig. 1 we can see an example in which the resulting matrix is composed of 16 blocks. Each block is managed by a different thread. We can notice that each block is independent and we are assured that only its corresponding thread is going to write on it. With this structure we lower the number of accesses to entries of both A and B. In particular, only gray coloured areas of A and B are needed for that particular thread handling block C_{22} as we can see in Fig.1.

Block size is an important constant which can influence the execution time of the operation. An analysis on its effect has been carried out in Sec. 3.3.

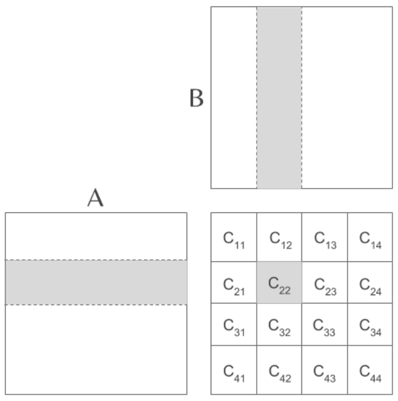


Figure 1: Graphical visualization of multiplication $C = A * B$. At this particular step block C_{22} is showed. We can notice that it depends only on *i-th* row of matrix A and *j-th* column of matrix B.

2.3 Get sub

In order to avoid the polymorphic call hierarchy for each entry of the matrix, the method **get_sub()** has been introduced. This method extracts a sub matrix (block) of given dimensions. By doing so, the number of polymorphic calls is reduced considerably.
Depending on the block size, if set equal to 100 as brought by the analysis in Sec. 3.3, it means **100 times less** polymorphic calls than the naive implementation.

In case of hierarchies composed of Windows or constant Diagonal matrixes, this method simply calls the base method by passing the adjusted parameters.
In case of a Transpose matrix, there were 2 choices:

- Make a single call to base matrix and arrange the result internally or
- Make ordered single calls to base matrix to get matrix entries one by one while filling the local resulting vector

Given the fact that the hierarchy of matrices may be deep, the first choice has been adopted. That is, the method **get_sub()** of base matrix is called with inverted parameters and the resulting vector has been adjusted to create the order which a transposed matrix has.

2.4 Concurrent operations

The main mechanism used to deal with other concurrent operations is operator overloading. In particular, the operator, using 2 matrices creates an instance of lazy structure (matrix_addition or matrix_product), that is:

```
matrix_addition<T, matrix_ref<T,LT>::H, matrix_ref<T,LT>::W>
operator + (const matrix_ref<T,LT>& lhs,
            const matrix_ref<U,RT>& rhs);
```

This is the example of matrix addition.
Successively, in case of additional matrices to sum, in order to mantain the structure and add more, we do:

```
matrix_addition<T, h, w>
operator + (matrix_addition<T,h,w>&& lhs,
            const matrix_ref<U,RType>& rhs);

matrix_product<T,h,matrix_ref<U,RType>::W>
operator * (matrix_product<T,h,w>&& lhs,
            const matrix_ref<U,RType>& rhs);
```

And then all possible combinations to specialize it for less common cases, such as:

```
(1) (A*B) + (C*D)
(2) (A+B) * (C+D)
(3)  A + B*C
(4)  A*B + C
```

The implementation for *(1)* and *(2)* makes the two sides execute concurrently making use of the **promise-future** paradigm. Results are used to create and return a matrix_addition and matrix_product respectively.

For *(3)* clearly the priority is for multiplication operator, so it first creates a matrix_product and than the addition forces the multiplication to solve in order to create a matrix_addition with A. This behaviour is catched through the following operator overloading

```
matrix_addition<T,h,w>
operator + (const matrix_ref<U,RType>& lhs,
            matrix_product<T,h,w>&& rhs);
```

But, in cases in which the multiplication appears on the left, as in *(4)*, we have to further specialise the behaviour by adding a mirrored version:

```
matrix_addition<T,h,w>
operator + (matrix_product<T,h,w>&& lhs,
            const matrix_ref<U,RType>& rhs);
```

We can consider also the cases of operations carried out with parenthesis, for example:

```
(5) (A+B) * C
(6)  A * (B+C)
```

which give priority to the addition operation as explicitly requested by the user through parenthesis usage. Again operator overloading is the technique to deal with this. And we can think of other possible combinations to catch all possible cases.

3 Time Analysis

Some analysis on execution time have been carried out to evaluate the gain on adding the parallelized algorithm discussed in Sec.2.2 and on block size variation.

Library **chrono.h** has been used for time span calculation, in particular *high_resolution_clock* provides the clock type with shortest tick period.

Every point plotted is calulated as a mean of 5 different runs, for a better precision.

Matrixes are filled in a simple way:

```
matrix<int> A(size, size);
for(int i=0; i!=size; ++i)
    for(int j=0; j!=size;
        A(i,j) = (i+1)*10+j+1;
auto B = A.transpose();

auto t1 = std::chrono::high_resolution_clock::now();
matrix<int> E = A*B; // or any other operation
auto t2 = std::chrono::high_resolution_clock::now();
```

and the execution time is calculated as a difference between t2 and t1.
Experiments have been carried out on a machine with CPU Intel i5-5300U 2.3 GHz and 8GB of RAM.

3.1 Parellelization Gain

First thing we are going to see is how the algorithm performs on different matrix sizes. For simplicity we take square matrices (N x N). All matrices involved in the multiplication are of the same type and same size.

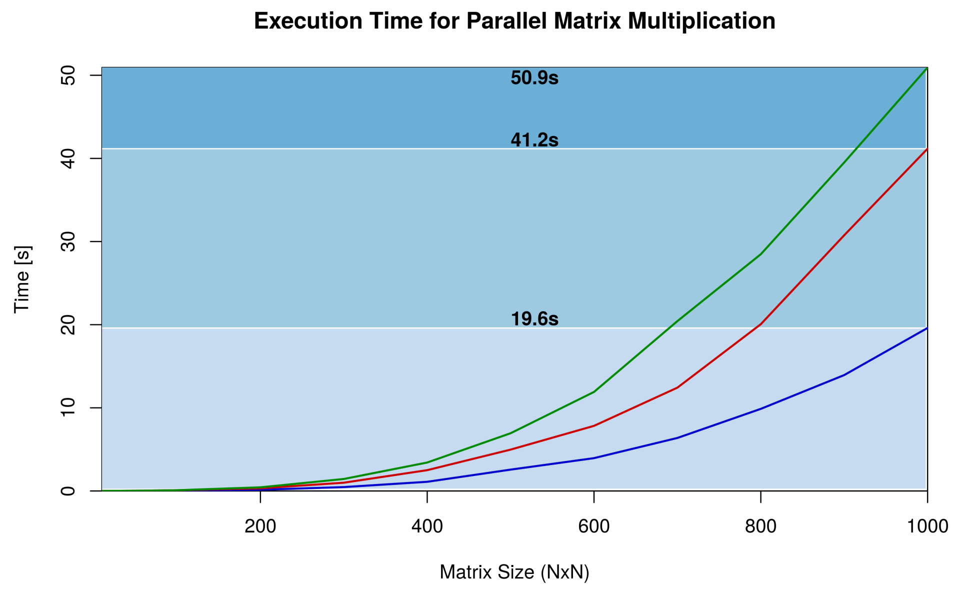
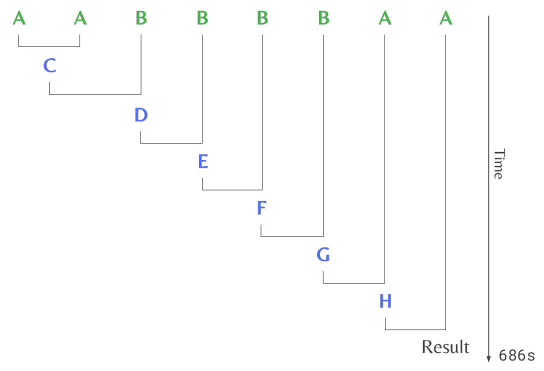
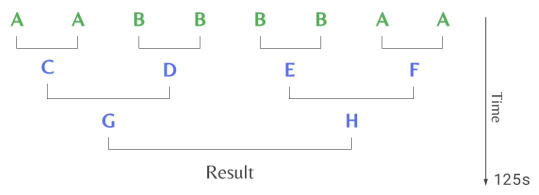


Figure 2: Execution time for multiplication AB in Blue, ABA in Red and $ABBA$ in Green. With Block Size = 100.

In Fig.2 we can see the behaviour for 3 different multiplication operations, in ascending complexity. In **Blue** we can see a simple multiplication $A * B$. Its execution time increases as the size of matrices involved increase. At the most complex case considered ($N = 1000$) it takes **19.6s** to execute the operation.
Adding another matrix to multiplicate (shown in **Red** in Fig.2) practically adds a second multiplication to execute and as we expect it doubles the execution time, indeed for $N = 1000$ it takes **41.2s**.
Now, the interesting part is the multiplication of 4 matrices (shown in **Green**) where without the usage of concurrent operations the expected time should be more than 3 times that of a single multiplication. But the parallelization of the starting four makes it convenient and shortens the overall execution time considerably. As we can notice from the figure, the execution time is **50.9s**.
Taking into consideration the case $N = 1000$, we can notice that the gap between blue line and red line is at least twice that between red and green lines, which suggests that the parallelizzation of operations brings a good improvement.



(a) Execution order without parallelization. The operation requires **686s**.



(b) Execution order with parallelization. The operation requires **125s**.

Figure 3: Execution order and time consumption of multiplication $A * A * B * B * B * B * A * A$ with matrix size 1000x1000.

Bringing in another experiment on a more complex operation, we can also graphically notice the difference between the implementation with concurrent operations enriched with parallel multiplication and the starting implementation without enhancements in Fig.3.
In the starting implementation (Fig.3a) operations are sequential, that is:

- First multiply the first couple of matrices
- The result of it is than multiplied with the third matrix
- The result of it is than multiplied with the fourth matrix
- ...and so on until we get the result

This brings a large consumption of execution time (**686 seconds**).
On the other hand (Fig.3b), by using concurrent operations, we can greatly shorten the execution time by executiong many multiplications concurrently, indeed the starting 8 matrices are taken in couples and multiplied concurrently. Same happens into the second level. This leads to **125 seconds** of execution time, which is a marked improvement.

3.2 Parallel Multiplication Algorithm

Another aspect to analyse is the improvement gained from the adoption of the parallel multiplication algorithm.
We take into consideration a simple operation $A * B$ which does not use the concurrent operations mechanism already analysed, but takes advantage only of the block-multiplication algorithm explained in Sec.2.2.

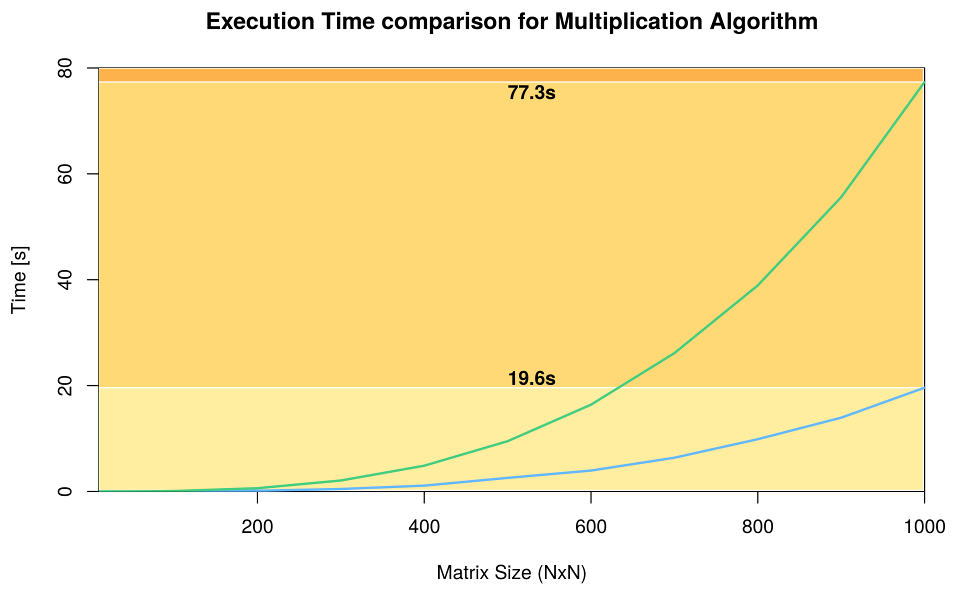


Figure 4: Simple multiplication operation $A * B$ to carry out information about the parallel multiplication algorithm. In **Blue** there is the execution with the parallel multiplication algorithm, while in **Green** the execution whithout.

In Fig. 4 we can see the plot of the multiplication with and without the parallel algorithm. We can notice that the difference between the two executions gets greater along with the dimension of matrices involved. At the largest dimension considered ($N = 1000$) the difference is more or less **57 seconds**.

The difference for small matrices is small because there is an overhead for thread creation and management which the parallel version has to manage, while in the sequential version it goes straight to operations. But for larger matrices the difference becomes larger because the parallel version can make a better use of phisical cores of the machine in which it is executed.

In this experiment the block size has been set to 100, and an analysis on its effect is carried out on the next section.

3.3 Block Size Effect

The parallel multiplication algorithm, as pointed out earlier, divides the resulting matrix into blocks of fixed size. The size of blocks may influence the performance.
An analysis on performance has been conducted on a simple matrix multiplication involving two square matrices of size 1000 x 1000, given the fact that the multiplication operation becomes expensive in terms of execution time starting from matrices with such size, as pointed out on the previous experiments. Thus, the parallel multiplication algorithm becomes more important on such sizes.

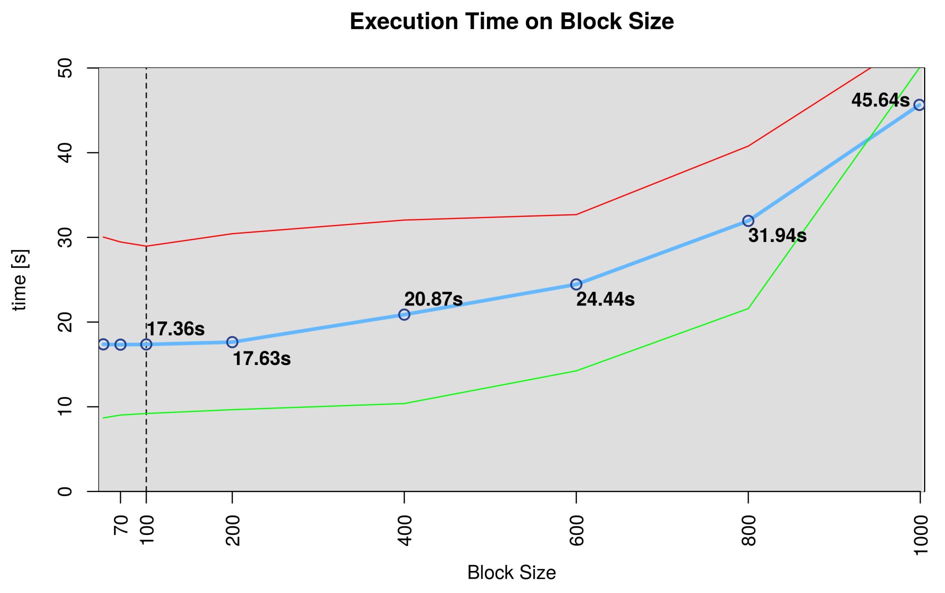


Figure 5: Execution time for multiplication $A * B$ on block size variation. Size of square matrices involved are: in **Green** 800x800; in **Blue** 1000x1000; in **Red** 1200x1200

In Fig.5 we can see how the execution time changes on block size changes.
The effect is not negligible, it can go from **17** to **45 seconds**, as we can notice on the main line plotted in Blue. On increasing sizes of blocks, the performance decreases.

Based on these experiments, a good block size appears to be 100. It is not a magic number, it's simply carried out on these empirical observations and for sure depends on threads available on the machine. It was used on these analyses given the fact that for dimensions up to 1000 it gave better performances. The empirical trend is confirmed by the other two lines showed in Fig. 5.

To overcome this variability a good idea might be to set a variable block size, for example taking $\sqrt{h_1}$, where h_1 is the height of the left-side matrix of the multiplication. This would create a variable number of blocks, and can be tuned to adapt to the available threads of the machine in use.

4 Limits

A limit of this implementation shows up when matrix operations involve matrices of different types. In those cases it doesn't build a lazy evaluation structure (matrix_addition or matrix_product) because the container of such classes can't hold matrices of variable types. Which means that it is not possible to choose the best order to carry the multiplications, or to parallelize operations.

SFINAE (Substitution Failure Is Not An Error) paradigm has been used to detect cases in which types are different and to distinguish static matrices from dynamic ones. For such cases a simple matrix addition/multiplication is carried out, just without lazy evaluation. Distinguishing the cases in which static matrices are involved provides the means to carry static informations anyway.

In cases in which a lazy structure was created before encountering a matrix of a different type, it is going to be solved as previously and what happens internally to that structure remains unchanged. Which means that operations before encountering the different type matrix are safe.

Moreover, in some cases, like those of concurrent operations seen in Sec. 2.4:

```
(1) (A*B) + (C*D)
(2) (A+B) * (C+D)
```

if left side is of different type from right side than, just as in the normal case, the two sides are evaluated concurrently using **promise future** pradigm and addition(1)/multiplication(2) is carried out lastly.

In general the effect of this limitation has been reduced as much as possible.